Czech technical university in Prague

Faculty of Mechanical Engineering

Department of Instrumentation and Control Engineering

Design of a testing program for an automatic control system of tram lines

Master's thesis

Kostiantyn Mykhailov

Master's program: Automation and Instrumentation Engineering

Program specialization: Automation and Industrial Informatics

Supervisor: Doc. Ing. Martin Novák Ph.D.

Consultant: Ing. Jan Kuptík

January 2023

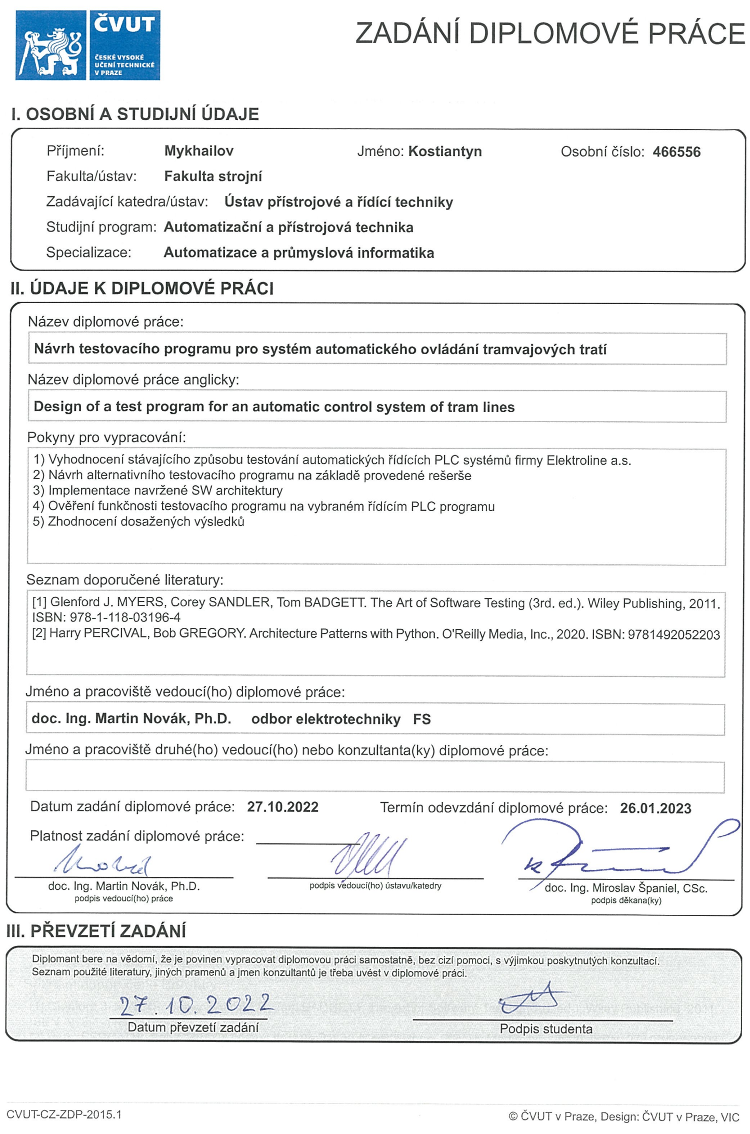
Supervisor:

Doc. Ing. Martin Novák Ph.D.
Department of Instrumentation and Control Engineering
Faculty of Mechanical Engineering
Czech technical university in Prague
Technická 4
160 00 Praha 6
Czech Republic

Consultant:

Ing. Jan Kuptík
Elektroline a.s.
Signaling System Development Department
K Ládví 1805/20
184 00 Praha 8
Czech Republic

Assignment



ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Mykhailov**	Jméno: **Kostiantyn**	Osobní číslo: **466556**
Fakulta/ústav:	**Fakulta strojní**		
Zadávající katedra/ústav:	**Ústav přístrojové a řídicí techniky**		
Studijní program:	**Automatizační a přístrojová technika**		
Specializace:	**Automatizace a průmyslová informatika**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh testovacího programu pro systém automatického ovládání tramvajových tratí

Název diplomové práce anglicky:

Design of a test program for an automatic control system of tram lines

Pokyny pro vypracování:

1) Vyhodnocení stávajícího způsobu testování automatických řídicích PLC systémů firmy Elektroline a.s.
2) Návrh alternativního testovacího programu na základě provedené rešerše
3) Implementace navržené SW architektury
4) Ověření funkčnosti testovacího programu na vybraném řídicím PLC programu
5) Zhodnocení dosažených výsledků

Seznam doporučené literatury:

[1] Glenford J. MYERS, Corey SANDLER, Tom BADGETT. The Art of Software Testing (3rd. ed.). Wiley Publishing, 2011. ISBN: 978-1-118-03196-4
[2] Harry PERCIVAL, Bob GREGORY. Architecture Patterns with Python. O'Reilly Media, Inc., 2020. ISBN: 9781492052203

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Martin Novák, Ph.D. odbor elektrotechniky FS

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **27.10.2022** Termín odevzdání diplomové práce: **26.01.2023**

Platnost zadání diplomové práce: ______

[illegible] [illegible] [illegible]

doc. Ing. Martin Novák, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

doc. Ing. Miroslav Španiel, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

27.10.2022
Datum převzetí zadání

[illegible]
Podpis studenta

CVUT-CZ-ZDP-2015.1

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Declaration of Authorship

Hereby I declare that this thesis is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during the elaboration of this work are properly cited and listed in complete reference to the due source.

Prague 2023 ....

Abstract

Předmětem této diplomové práce je návrh a vývoj testovacího programu pro systém automatického ovládání tramvajových tratí. Práce je tvořena ve spolupraci s českou společností Elektroline, a.s., jež se zabývá projektováním, výrobou, dodávkou a montáží elektrických i elektronických zařízení a vývojem řidičích systémů pro tramvajové i trolejbusové linky. V rámci této práce je vyhodnocen stávající způsob testování automatických řídících PLC systémů a následné navrhnut, implementován a otestován alternativní testovací program. Program je psán v programovacím jazyce Python s použitím principů objektně-orientovaného programování a asynchronního programování. Pomoci zvoleného komunikačního protokolu (současně OPC UA) program komunikuje s PLC, poveluje knihovnou modelů testovaných zařízení a funkcí a zároveň vyhodnocuje výsledky testů. Z uživatelského hlediska je program koncipován jako knihovna objektů, která umožňuje relativně rychle a jednoduše napsat skript pro otestování řídicího softwaru. Výsledkem testu je report ve formě tabulek uložených do souboru formátu HTML.

Klíčová slova: Testování softwaru, Asynchronní programování, Objektově orientované programování, Python, Asyncio, OPC UA

The subject of this thesis is the design and development of a testing program for an automatic control system of tram lines. The thesis has been created in cooperation with the Czech company Elektroline, a.s., which designs, produces, delivers, and installs electrical and electronic equipment and develops control systems for tram and trolleybus lines. In this work, the existing method of automatic PLC control systems testing is evaluated and subsequently, an alternative testing program is designed, implemented, and tested. The program is written in the Python programming language using the principles of object-oriented programming and asynchronous programming. Using the chosen communication protocol (currently OPC UA), the program communicates with the PLC, commands the library of models of the tested devices and functions, and evaluates the test results. From the user's point of view, the program is designed as a library of objects, which makes it relatively quick and easy to write a script to test the control software. The result of the test is a report in the form of tables saved in an HTML file.

Keywords: Software Testing, Asynchronous Programming, Object Oriented Programming, Python, Asyncio, OPC UA

Acknowledgments

I would like to thank my advisor, Doc. Ing. Martin Novák Ph.D., for guidance during the writing of this thesis. I am also grateful to my colleague Ing. Jan Kuptík for valuable feedback, suggestions, technical insights, and support throughout my work. I would also like to thank my family and loved ones for their support during my studies. Thank you all for helping me to reach this milestone.

List of Figures

1 Control system PLC hardware [4] 4

2 Railroad switch [2] 5

3 TSH 100 point machine [1] 6

4 General scheme of a track circuit [2] 7

5 PPI and SG signal lamps [2] 9

6 Tram-to-Wayside Communication Vetra [2] 11

7 Deployed control cabinet [11] 12

8 Tram depot plan in Sydney, Australia with marked zone areas [13] 14

9 Zone state machine diagram [13] 22

10 Route layout elements and hardware components [2] 23

11 Route state machine diagram [13] 25

12 PLC communication scheme [13] 31

13 Electrical schema of a track circuit simulator [11] 34

14 Simple test flowchart 35

15 File structure of a test project 39

16 Flowchart of the `SingleTest` class 70

17 Flowcharts of the `MultiTest` classes 72

18 Zone Z02 layout of the tram depot in Pilsen, Czechia [10] 81

19 Static track circuit testing scenario 84

20 Report for the `SetDetectorOutNOOutNC` test of detector 1TC21 85

21 Testing scenario for routes 2R01, 2R03 with `optimal_routing=True` . . . 89

22 `CheckIfPrevRoutePassed` test evaluation of route 2R03 90

23 Testing scenario for routes 2R01, 2R03 with `optimal_routing=False` . . 92

24 Testing scenario for routes 2R01, 2R08, 2R17, and 2R10
with `fastmode=False` and `truncate=False` 94

25 `CheckIfRouteFree 2R17` test evaluation
during the routes 2R01, 2R08, 2R17, and 2R10 testing scenario 95

Acronyms

AB All Blocked state. 21, 22, 24, 27, 28

ABC Abstract Base Class. 64

EI Emergency Inhibit state. 21, 28, 40

EYAS Elektroline Yard Automation System. 15, 21, 31

HMI Human-machine interface. 13, 15, 26, 29

I/O Input/Output. 3, 15, 16, 33

NORM Normal state. 21, 22, 27

NR Numeric Rail. 9

OOP object-oriented programming. 53

OPC UA OPC Unified Architecture. 30, 31, 33, 42, 44, 45, 49, 61, 63, 76–78

PLC Programmable logic controller. 1, 3, 12, 15–18, 20, 26, 32–34, 38, 47, 61, 62, 77

PPI Point Position Indicator. 9, 13, 21

SAR System Auto Recovery. ix, 18, 22

SG Stop-Go. 9, 21, 23, 24, 60, 75, 80

SIL Safety Integrity Level. 15

SPAD Signal Passed at Danger. ix, 19, 21, 22, 27

TWC Tram-to-wayside communication. 10, 11, 26

Contents

List of Figures **vii**

Acronyms **viii**

1 Introduction **1**
1.1 Motivation 1
1.2 Thesis Structure 2

2 System Hardware Overview **3**
2.1 PLC Hardware 3
2.2 Point Machines 5
2.3 Position Detectors 7
2.4 Signal Lamps 9
2.5 Communication Devices 10
2.6 Control Cabinet 12
2.7 Heating System 13

3 Control System Overview **14**
3.1 General Overview 14
3.2 Software Implementation 16
3.3 Program Structure 18
3.4 Zone Logic 20
3.4.1 States 21
3.4.2 System Auto Recovery (SAR) 22
3.5 Route Logic 23
3.5.1 States 24
3.5.2 Requesting and Request Memory 26
3.5.3 Build 26
3.5.4 Signal Passed at Danger (SPAD) 27
3.5.5 Passage 27
3.5.6 Cancellation 29
3.6 Communication 30

4 Analysis of the Current Testing Process **32**
4.1 Stages of Testing 33
4.1.1 Software Testing 33
4.1.2 Implementation and Wiring Testing 33
4.2 Testing Scenarios 35

4.3 Software Implementation 36
4.3.1 `El_Testing` library 37
4.3.2 Testing Files and Their Structure 39
4.3.3 `pytester` module 44
4.4 Conclusion 48

5 New Testing Program Implementation 49
5.1 Genaral Development Principles 49
5.1.1 Reconsidering the Testing Files 50
5.1.2 Modular Design 52
5.1.3 Concurrent Programming 54
5.2 `elements` package 57
5.2.1 `Element` class 58
5.2.2 `ElementNode` class 61
5.2.3 `ElementCommAdderss` class 63
5.3 `tests` package 64
5.3.1 `SingleTest` class 66
5.3.2 `MultiTest` class 71
5.3.3 `PassageGenerator` class 73
5.3.4 `TestMemory` class 75
5.4 `client` package 76
5.4.1 `ClientProtocol` and `OpcUAClient` classes 77
5.4.2 `NodeSubscriptionHandler` class 78

6 Evaluation of the New Testing Program on the Pilsen Tram Depot PLC Control System 80
6.1 Static Tests 82
6.1.1 Case 1: Test of the track circuits functionality 82
6.1.2 Case 2: Test of the point machines functionality 86
6.2 Dynamic Route Passages 87
6.2.1 Case 1: Two routes, fast mode, passage truncation 88
6.2.2 Case 2: Two routes, no truncation, enforced test fail 91
6.2.3 Case 3: Four routes, real-life duration, full passage length 93

7 Conclusion 96

References 98

1 Introduction

1.1 Motivation

Programmable logic controller (PLC) control program testing is a process used to ensure the reliability and effectiveness of the control program software for PLCs. Control program testing is particularly important in the context of trams, as these systems are responsible for the safe and efficient operation of the tram network. Testing PLC control programs involve simulating different scenarios and inputs to evaluate the performance and behavior of the program and identifying any potential issues or improvements that can be made.

There are various methods and techniques that can be used for PLC control program testing, including simulation, functional testing, and stress testing. Simulation involves creating a virtual environment to mimic the real-world conditions and inputs that the PLC control program will encounter, and evaluating how it performs in these scenarios. Functional testing involves testing the individual components and functions of the control program to ensure that they are working correctly and meeting the desired specifications. Stress testing involves subjecting the control program to extreme or abnormal conditions to evaluate its robustness and reliability.

Since PLC control program testing is an essential step in developing and implementing PLC control programs, it is important that such testing is easy and automated. This can help to reduce the time and resources required for testing and make it more feasible to conduct thorough and comprehensive evaluations of PLC control programs.

Automated testing allows for the rapid and efficient evaluation of PLC control programs, as it allows for the creation of virtual environments and scenarios that can be used to test the performance and behavior of the programs. This can help to identify potential issues and improvements more quickly and accurately and reduce the need for manual testing and analysis.

Easy testing is also important, as it can help to ensure that PLC control program testing is conducted on a regular and consistent basis. By making the testing process simple and straightforward, it is more likely that it will be conducted regularly and thoroughly, leading to improved reliability and effectiveness of the PLC control programs.

1.2 Thesis Structure

The text of the thesis is divided into several chapters. Chapter "System Hardware Overview" describes the physical elements of the system - point machines, position detectors, signal lamps, communication devices, control cabinet, and heating. Chapter "Control System Overview" is dedicated to the analysis of the software modules of the control system, both the ones that manage the physical elements and pure software ones - system, zone, gate, and routes. Chapter "Analysis of the Current Testing Process" examines the specifics of the current testing guidelines and procedures, as well as their software implementation. Chapter "New Testing Program Implementation" describes the development process and the structure of the new testing program. Chapter "Evaluation of the New Testing Program on the Pilsen Tram Depot PLC Control System" validates the correct function of the new software through the results of the testing of a chosen controlled area and the comparison between the old and new testing process in terms of flexibility of scenario design and testing speed. Chapter "Conclusion" discusses the result of the whole work, including the improvements it introduces to the testing process, potential limitations of the software, and the vector of future work.

In the text of the thesis, specific formatting is used to highlight the important terms. Key terms of a paragraph are written in `cursive`, and code-related terms (for example, source code, function and class names, etc.) are written in `typewriter` font. The parts of the text that contain general descriptions and more complex examples are named ⌜ **Sidebars** ⌟. These passages may be skipped if the reader is knowledgeable about the described topic. Finally, Elektroline, a.s. is referred to as "the company".

2 System Hardware Overview

Before embarking on designing a testing program, it is important to gain an essential understanding of the control system hardware. This includes the system's architecture, the specific components that make up the hardware, and how these components interact with one another. This knowledge may aid in analyzing the necessary testing procedures and understanding the specifics of the control software logic implementation.

On a hardware level, the company's automatic signaling control system consists of different physical elements, including programmable logical controllers (PLCs) and peripheral Input/Output (I/O) cards, point machines, position detectors, signal lamps, communication devices, heating systems, and other equipment.

2.1 PLC Hardware

The control system is based on product line X20 of B&R Industrial Automation (also known as B&R), an Austrian company that specializes in the development and production of hardware and software for industrial automation. It utilizes a range of distributed modules and I/O PLC hardware to control and monitor the movement of trams and other processes within the system. The coated variants of the modules are often used to protect the hardware against condensation and corrosive gases. The products' protection against condensation and corrosive gases is tested using BMW GS 95011-4 and EN 60068-2-60 standards, respectively [4].

The controller of the system is usually the X20cCP1584. It has a USB interface, Ethernet connectivity, and a CompactFlash slot. CompactFlash memory card contains the runtime system (the program which operates the CPU, communication, and peripheries), and application software (the control program). It serves as a hard drive for the PLC. The CPU also has a slot for X20 12-pin peripheral modules (for example, I/O modules or power supply modules), which can be used to expand the system's capabilities. For safety-critical applications, the system may also include the X20cSL8101 CPU unit equipped with B&R SafeLOGIC functionality, which is able to execute applications designed in B&R SafeDESIGNER and coordinate the safety-related communication of all modules involved in the application.

X20 I/O cards implement modular design and consist of three basic elements: a terminal block, an electronic module, and a bus module. Such design allows the system to have a solid combination of performance, flexibility, and seamless integration. Peripheral modules, that are commonly used in the control system, include input modules for digital (X20cDI9371) and analog (X20cAI4622) signals, output modules for digital (X20cDO9322) signals, safety input modules for digital signals (X20cSI9100), safety digital output (X20cSO41200) and relay (X20cSO2530) modules, temperature input module (X20cAT4222), RS422/485 interface module (X20cCS1030), power supply module (X20cPS9400), and others.

Regular, safety, and high-voltage modules used in the system look similar, but they can be distinguished by their color. Grey modules are used for standard I/O, yellow modules are used for safety I/O, and red modules are used for 240 VAC I/O.

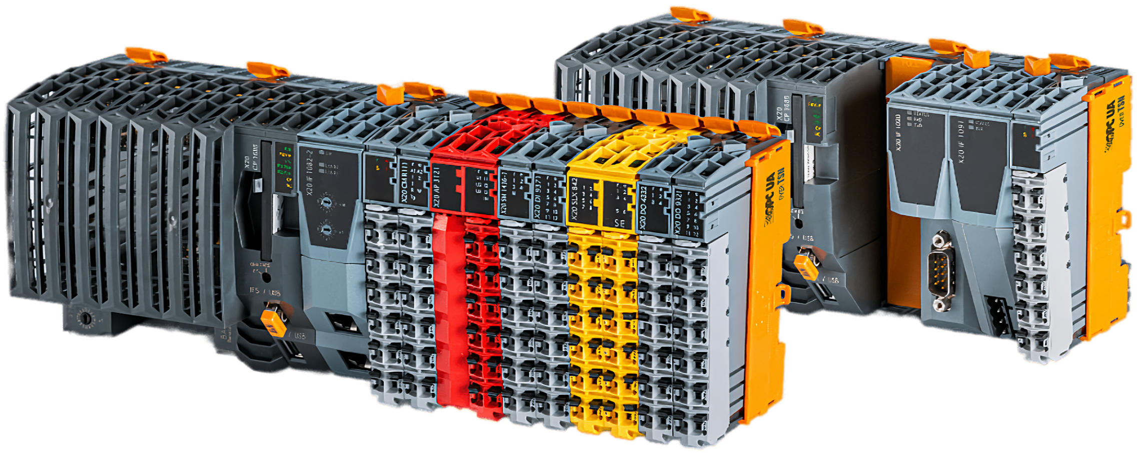


Figure 1: A composition of a PLC CPU, a safety CPU, and I/O cards [4]

2.2 Point Machines

A railroad switch, also known as a turnout, is a railway facility at the point where tracks diverge or converge. It may have various mechanical executions, but most of them incorporate a pair of linked tapering rails, known as points (also switch rails or point blades), lying between the diverging outer rails (the stock rails). Railroad switches are an important part of the infrastructure of a railway. They allow trains to change direction and move from one track to another, providing them access to different parts of the railway network.

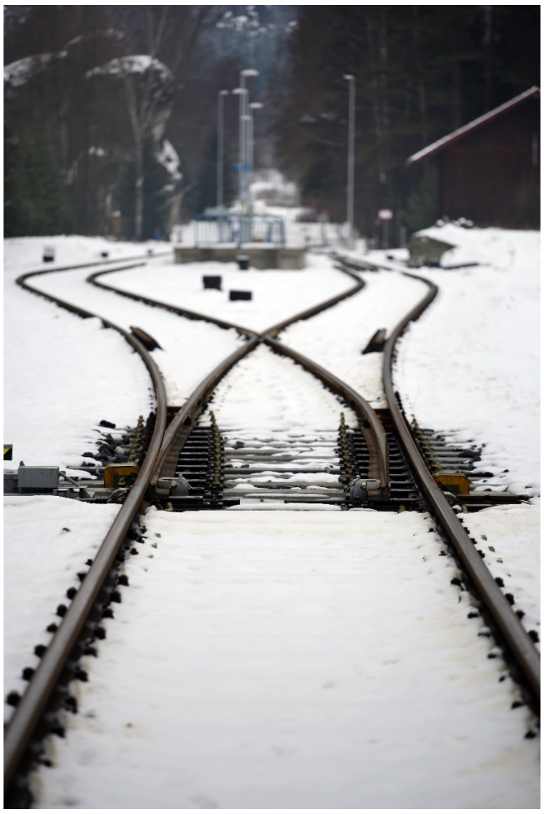


Figure 2: A railroad switch, also known as a turnout [2]

Railroad switches are operated using *point machines*, which can be either electrically or mechanically actuated. These machines ensure that the switch rails are properly aligned and that trains can safely move from one track to another. They are equipped with sensors and safety features to prevent accidents and ensure smooth operation, for example, locking mechanisms that prevent unauthorized or accidental movement of the switch rails. The company uses electrically and mechanically actuated point machines.

A mechanical point machine does not have an electrical locking function, nor can it be automatically controlled. Such point machines are usually trailable (at least from one direction), which means they can be moved mechanically directly by the wheels of a rail vehicle to guide it onto a different track. It is worth mentioning that trailing, while being useful for mechanical point machines, is unwanted for electrical point machines and leads to faster wearing and potential malfunctions. A mechanical point machine can be also equipped with a spring-back mechanism, designed to automatically return the switch blades to their normal, straight position after the rail vehicle passes over them. The maximum speed for mechanical point machines with trailing functionality is advised to be 15 km/h.

An electrical point machine can be driven by an electric motor or an electric motor in combination with a hydraulic system. In the latter case, the electric motor pressurizes the hydraulic motor, and switching is realized hydraulically by means of valves. The motor supply is 600 VDC and is controlled by the control system via power contactors. It is also possible to switch the point machine manually with a metal rod. An electrical point machine is equipped with a safety locking function, that ensures that it is not moved under a passing tram, and forced trailing detection (a loss of position of the point machine).

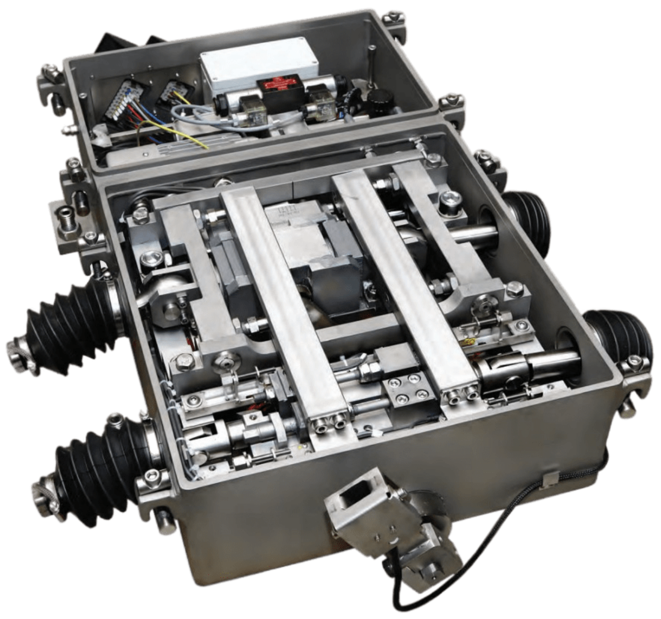


Figure 3: TSH100 - the most sophisticated point machine in the company's portfolio [1]

Both mechanical and electrical point machines are equipped with one, two, or three tongue position sensors on each side and the sensor of the manual lever rod presence. The sensors are connected via an input unit, which ensures the protection of the electronics. The point machine itself is placed in the ground box.

For maintenance purposes, the remote control of the turnouts can be overridden by a switch located on the side of the switchboard. When activated, the switch cuts power from the point machine and deactivates the remote control, causing the turnout to be locked and unable to be switched by any command from the signaling system. After maintenance is completed, the override switch is turned off to resume normal operation.

2.3 Position Detectors

A position detector is a type of equipment that is used to determine the location of a tram along the track. This information is used by the control system to monitor and control the movement of the tram, for instance, to ensure that it stays on its designated track. The company uses two main types of position detectors: *track circuits* and *pantograph detectors*. Both of them are used for tram position detection, mainly to prevent point machines from unsafe switching during the tram passage.

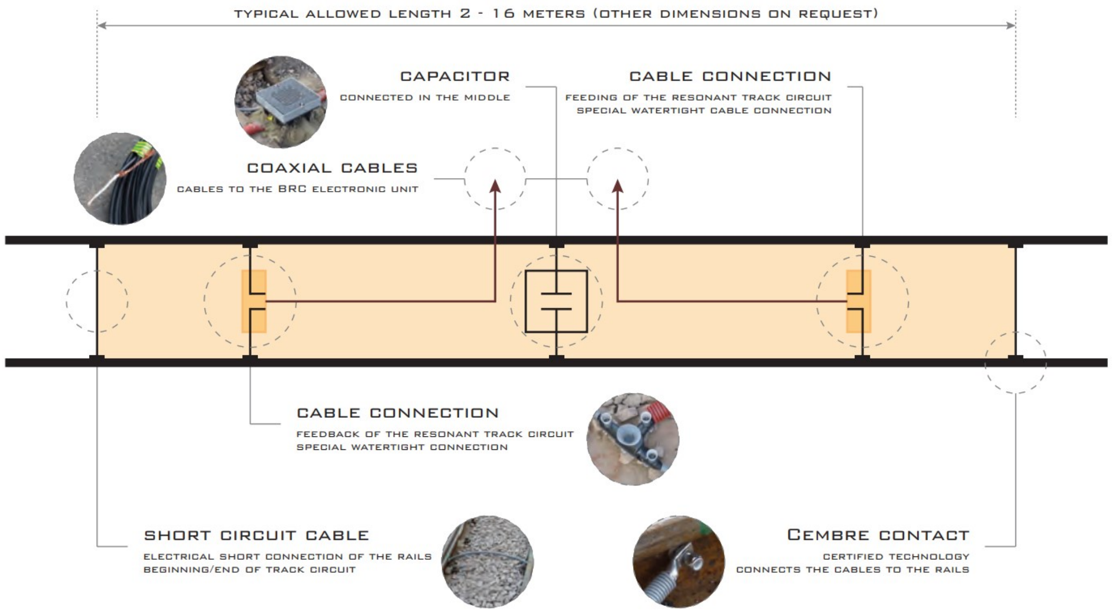

TYPICAL ALLOWED LENGTH 2 - 16 METERS (OTHER DIMENSIONS ON REQUEST)
CAPACITOR
CONNECTED IN THE MIDDLE
CABLE CONNECTION
FEEDING OF THE RESONANT TRACK CIRCUIT
SPECIAL WATERTIGHT CABLE CONNECTION
COAXIAL CABLES
CABLES TO THE BRC ELECTRONIC UNIT
CABLE CONNECTION
FEEDBACK OF THE RESONANT TRACK CIRCUIT
SPECIAL WATERTIGHT CONNECTION
SHORT CIRCUIT CABLE
ELECTRICAL SHORT CONNECTION OF THE RAILS
BEGINNING/END OF TRACK CIRCUIT
CEMBRE CONTACT
CERTIFIED TECHNOLOGY
CONNECTS THE CABLES TO THE RAILS

Figure 4: A general scheme of a track circuit [2]

Track circuits are resonant circuits that detect trams based on a short circuit caused by the vehicle's axle in combination with the iron mass of the vehicle floor. They are typically 6-18 meters in length and consist of two-track shunts (that define the size of the track circuit), a capacitor, and a special evaluation unit called BRC, developed by the company. The BRC unit is located in the control cabinet; it supplies power to the circuit and evaluates vehicle detection. Track circuits are tuned on-sight during the system hardware installation, and the tuning includes the selection of the correct capacitor value. This tuning allows track circuits to reliably detect trams but not other road vehicles.

Pantograph detectors, on the other hand, detect trams by detecting contact with the pantograph, a device on the roof of the tram that collects power from overhead wires. These detectors are mounted on the catenary wire, which carries the electrical power for the tram, and are designed to handle the high voltage used by trams. The voltage signal from this detector must be reduced before the signal goes into the control system.

In terms of their relative advantages and disadvantages, track circuits are generally considered to be more reliable and accurate than pantograph detectors. While track circuits are able to detect the precise position of the tram, pantograph detectors are only able to evaluate if a tram is passing through a secured area. This is because track circuits use a physical connection with the track to detect the presence of trains, whereas pantograph detectors rely on mechanical contact with the pantograph, which can be less reliable. Track circuits, if tuned properly, only detect trams and are not receptive to interference from other electrical devices. They can be used to detect the presence of trams in tunnels or other areas where pantograph detectors may not be effective. One of the main advantages of a pantograph detector is its simple installation. It can be integrated into an already existing system without the need for extensive groundwork, which is especially useful for zones in historic city parts. Also, they are not affected by the electromagnetic characteristics of the track. If the rail tracks are short-circuited inside the area of a track circuit (for example, by steel reinforced concrete structure between the tracks), such a track circuit can never work properly. This can make pantograph detectors more versatile and easier to install in certain situations.

2.4 Signal Lamps

Signal lamps are used to direct traffic in a tram system by providing information about the route, its status, and compatibility with other routes. The company uses several types of signaling lamps: *Stop-Go (SG) lamps*, *Point Position Indicator (PPI) lamps*, *Matrix signals*, *Numeric Rail (NR) lamps*, and occasionally some custom modifications.

SG lamps are located at so-called gates, which are the entrances to the controlled area, and provide information about the preparedness of the route. They display either a "STOP" or a "GO" symbol, indicating whether the route is clear and all point machines are in the correct position and are electrically blocked. The main signal gives the driver permission to drive and prevents trams from collisions. The default symbol of the main signal is the "STOP" symbol. Secondary lamps may also be located along with the main signalization lamps to provide additional information to the driver.

PPI lamps are located near switch points in a clearly visible location. They inform the driver about the position and locking status of the switch, allowing the driver to safely navigate the route. They may display symbols such as "switch point set to the left and locked" or "switch point in mid-position and NOT locked". This information is crucial for the driver to properly navigate the route and avoid collisions or other hazards.

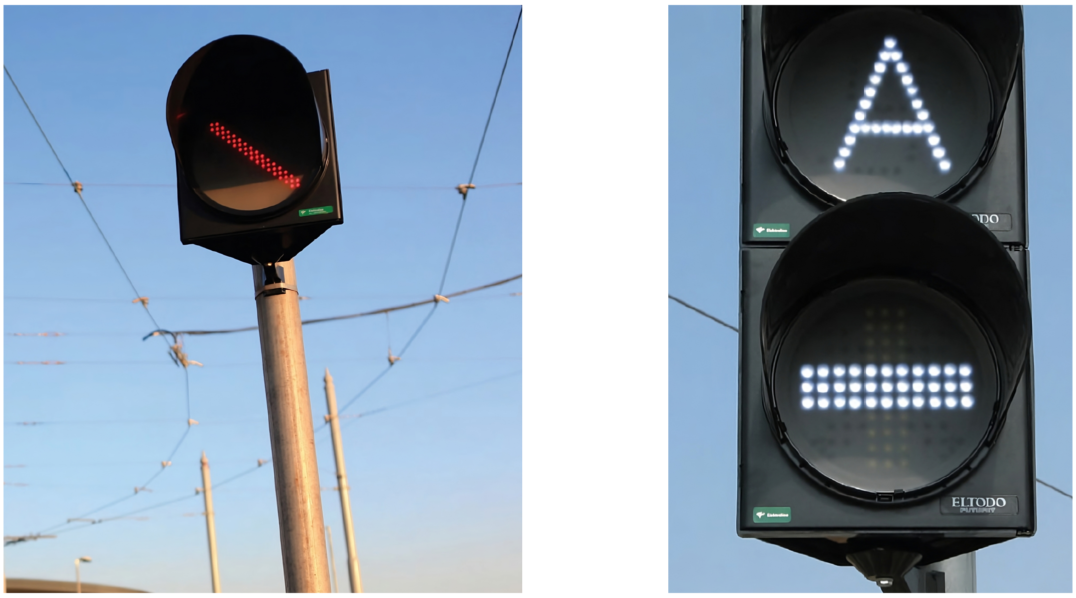


Figure 5: PPI signal lamp (left) and SG signal lamp (right) [2]

Some parts of the signaling lamp may be equipped with current sensors to determine whether the aspect of the lamp is on or off. During measurement, if a deviation from the calibrated value is detected, a warning or error is generated. The reference value of the current for the correct state of the signal is established through a calibration process. During calibration, the signals are turned on and the personnel confirms their correct function, and the actual current value is stored as a reference.

Matrix signal devices and NR lamps serve a supplementary role, providing additional information to the driver. Matrix lamps are more versatile and use a LED-based RGB matrix display to provide various information, such as the system operating mode, tram ID, requested route, upcoming route, warnings and errors, and more. NR lamps, on the other hand, provide additional information about the situation on a particular track. They are usually hung above a track in a depot in a position visible from the entrance to the depot, so the driver can see the state of the destination track upon entering the depot. They are 7-segment, two-digit color displays located above the entrance to each track, with the number identifying the track number and the color of the number indicating its meaning. As these are supplementary signals, they do not have any safety control functions. In the event of a failure, the system remains operational. Additionally, these signals are not backed up in case of a power failure, and will therefore be extinguished.

2.5 Communication Devices

Tram-to-wayside communication (TWC) is a type of communication between a tram and the area adjacent to the tracks, known as the wayside. This communication is typically used for operational and safety purposes, such as for signaling and for transmitting information about the tram's location and status.

The company uses wireless radio or cellular communication technologies, such as its own product called *Vetra*, or third-party communication devices such as *VECOM*, *SPIE*, and others. The Vetra system, for example, is a two-way communication system between the tram and the signaling system. It uses 2.4 GHz radio data transmission and is designed for trams traveling up to 80 km/h. A tram usually has antennas installed on the front and rear of the vehicle, and the rear car transmits a "rear antenna" flag. A coupled tram has two antennas on each car (the trailing car transmits the "rear car" flag). Ground Vetra antennas are installed primarily at the entrances to a zone to detect and transfer tram requests. It is also used to improve the reliability of the visualization and record events. When a tram passes over a ground antenna, communication with the Vetra antenna on the tram occurs, and the tram simultaneously sends and receives data messages to and from the signaling system.

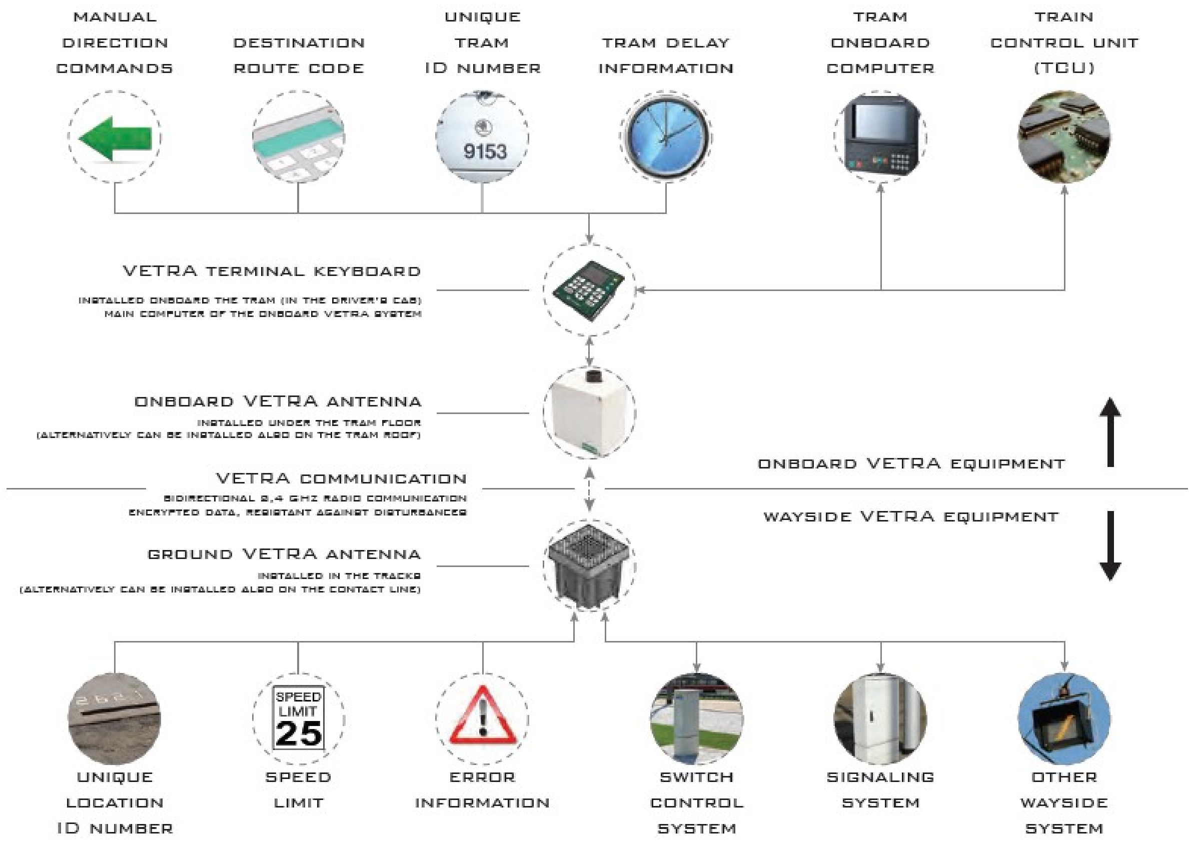

MANUAL DIRECTION COMMANDS
DESTINATION ROUTE CODE
UNIQUE TRAM ID NUMBER
9153
TRAM DELAY INFORMATION
TRAM ONBOARD COMPUTER
TRAIN CONTROL UNIT (TCU)
VETRA TERMINAL KEYBOARD
INSTALLED ONBOARD THE TRAM (IN THE DRIVER'S CAB)
MAIN COMPUTER OF THE ONBOARD VETRA SYSTEM
ONBOARD VETRA ANTENNA
INSTALLED UNDER THE TRAM FLOOR
(ALTERNATIVELY CAN BE INSTALLED ALSO ON THE TRAM ROOF)
VETRA COMMUNICATION
BIDIRECTIONAL 2,4 GHZ RADIO COMMUNICATION
ENCRYPTED DATA, RESISTANT AGAINST DISTURBANCES
ONBOARD VETRA EQUIPMENT
WAYSIDE VETRA EQUIPMENT
GROUND VETRA ANTENNA
INSTALLED IN THE TRACKS
(ALTERNATIVELY CAN BE INSTALLED ALSO ON THE CONTACT LINE)
UNIQUE LOCATION ID NUMBER
SPEED LIMIT 25
SPEED LIMIT
ERROR INFORMATION
SWITCH CONTROL SYSTEM
SIGNALING SYSTEM
OTHER WAYSIDE SYSTEM

Figure 6: Tram-to-Wayside Communication Vetra [2]

When the TWC communication system is not functioning properly or an incorrect route has been requested, the manual switch, also known as the *keyswitch*, serves as an alternative means of selecting or canceling the route. This switch can be found near the entrance gate on the pole of the SG lamp or nearby. It is housed in a sturdy, locked box and can only be unlocked and activated with a special key by the driver.

2.6 Control Cabinet

The *control cabinet* is an essential component of the control system, as it contains all of the control components that are necessary for the system to function properly. Typically, it is a composite housing that contains a variety of different devices, including a PLC, a BRC unit, a DC/DC converter, and an interface for communication transmitters.

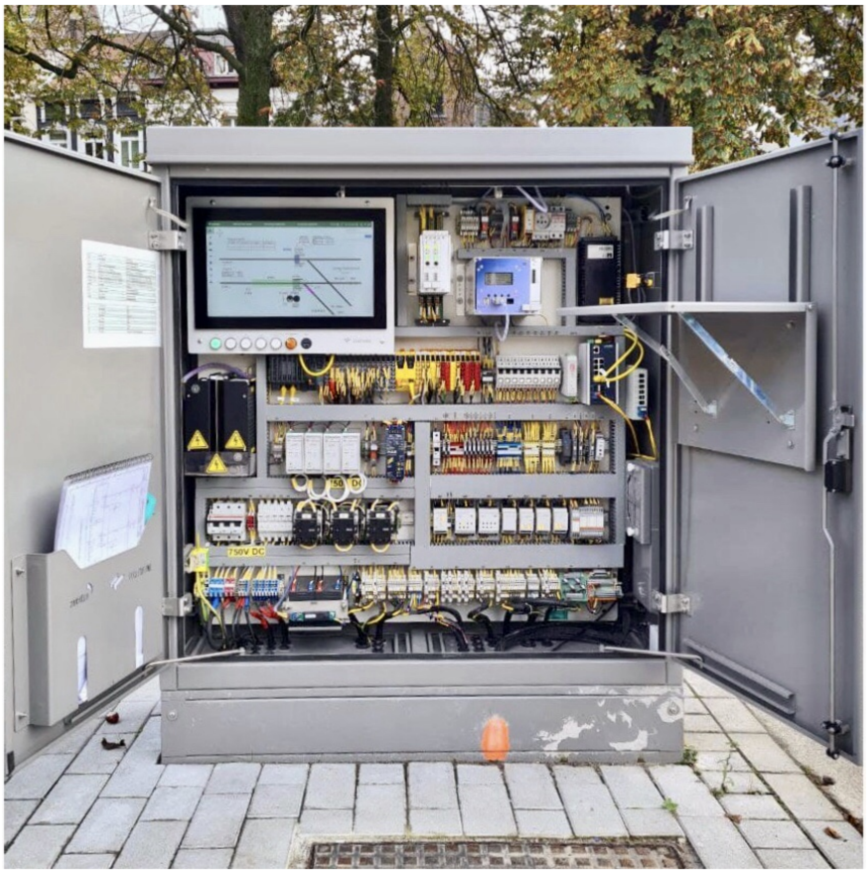


Figure 7: A deployed control cabinet [11]

A PLC is one of the key components of the control cabinet. It is responsible for managing the routing logic and controlling other equipment within the system (for instance, the heating system, which prevents the point machines from freezing), enabling the system to operate automatically, without the need for constant human intervention. The BRC unit, mentioned in section 2.3, is another important component of the control cabinet. It monitors and controls the operation of track circuits within the system. In addition to the PLC and BRC equipment, the control cabinet may also include evaluation units for pantograph detectors, outputs for signal lamps, and other devices.

The control cabinet is equipped with a touch panel with zone visualization, as well as touch buttons that allow for the switching between operational modes, lamp calibration, and emergency heating. Additionally, the control cabinet includes a remote communication modem, which allows for remote monitoring and control of the system.

The control cabinet is primarily powered from the overhead contact line of 600 VDC voltage, and a converter to 24 VDC is used to supply power to all its components. In case of a power failure, it is powered from the *UPS* power backup unit. The power supply backup is designed to keep the system operational for a preset time, except for point switching, supplementary signaling (such as PPI lamps), and heating. The system allows remote control and monitoring and detects and logs tram communication and passages. If the overhead line voltage is not restored within this time, the system will shut down to conserve battery power. Otherwise, after the overhead line voltage is restored, the system may remain blocked until it is switched until a specific action is performed or may switch to the normal operation state right away.

2.7 Heating System

Heating is not a single device, but a subsystem of devices that contains components to control the heating of point machines. It is a critical subsystem for the colder regions, as it prevents the point machines from freezing in cold weather. The heating system operates in several modes, including Permanently OFF, Permanently ON, Automatic mode, and Emergency heating. It includes a weather station and rail temperature sensors to provide up-to-date data on weather and temperature and uses this information to control the heating in automatic mode. The heating rods are rated at 600 VDC and are switched through power contactors with protective barriers against direct contact. In the absence of overhead line voltage, the heating rods do not function. The heating system can be controlled remotely or locally and includes error monitoring of each heating rod. All heating parameters can be set locally on the Human-machine interface (HMI) panel or remotely, including offsetting adjustment for automatic heater turn-on. There is also a mechanical button in the cabinet that can be used to turn on the Emergency heating in case the system is out of service. This button immediately turns on the heaters to prevent freezing.

3 Control System Overview

To design an effective testing program for control software, it's essential to have a basic understanding of the control software itself. The knowledge of the program logic and its implementation enables the creation of precise test cases that can effectively evaluate the software's performance and reliability. Furthermore, it assists in identifying potential areas of failure and designing test cases that specifically target these areas.

3.1 General Overview

A physical railroad layout (may it be a depot or a railroad section) is usually divided into smaller controlled areas, also called *zones*. Each zone is an independent traffic hub. The traffic throughout the zone is usually managed on different levels. The system itself is usually designed to fulfill the requirements of the client and the tender documentation.

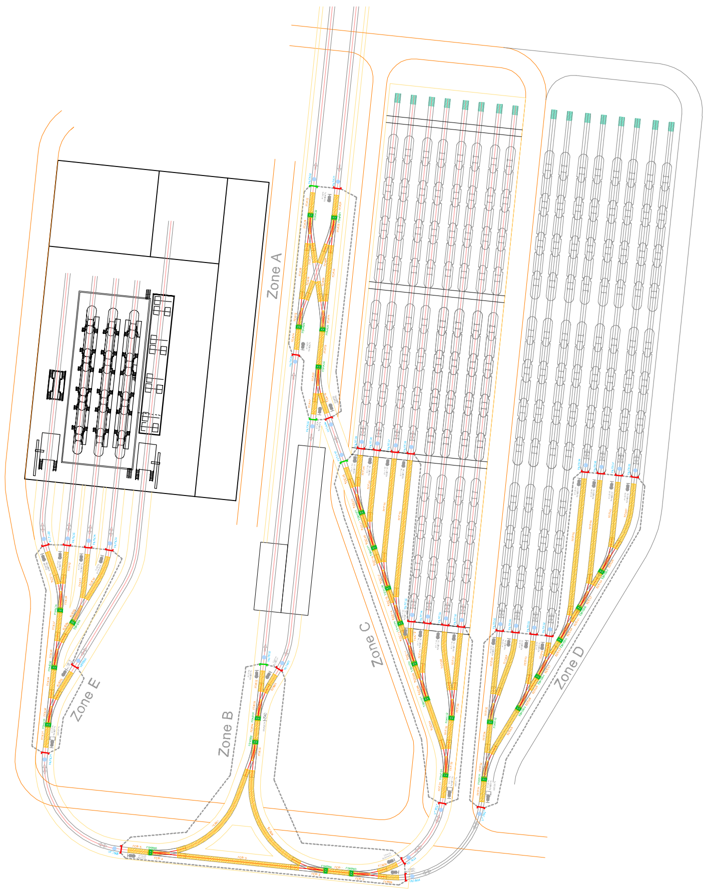


Figure 8: A tram depot plan in Sydney, Australia with marked separate zone areas [13]

Separate zones are controlled by the *signaling control system*, a PLC-based program. Each zone has one control PLC and a set of I/O modules, located in one or several control cabinets. If SIL2 or SIL3 safety is required (Safety Integrity Level (SIL), is a measurement of the safety performance of a safety function within a safety system based on the IEC 61508 standard), a zone is called a safety zone, and two PLCs may be used. A special safety PLC handles the routing and blocking logic of the zone, while a regular non-safety PLC is responsible for auxiliary functions (logging, visualization, etc.).

The main functions of the signaling control system are:

- Tram passage management (passage requests handling, routing logic, etc.);
- Verification of the availability of the point machines, switching their direction according to the request, and blocking them while they are reserved for a passage;
- Activation of the signal lamps to direct trams to their intended destinations;
- Tram detection with the position detectors;
- Communication with trams as they pass through;
- Handling of potentially hazardous and error situations;
- Heating system activation according to the weather conditions;
- Communication with external systems, i.e. vehicle washing facilities, trolley power supply, traffic signal controllers outside of the zone area, etc.

In case of any specific requests or unpredictable situations, the system can be controlled via an HMI from the dispatcher's room or via the on-sight push-buttons interface, if such is implemented. If zone control cabinets are not functional, the function of the control system may be limited.

It is worth noting that complex projects often require the implementation of a separate high-level program called Elektroline Yard Automation System (EYAS), which manages the global traffic in the controlled area. It monitors the entire location, implements the tram movement is also visualization on the dispatcher's control interface, communicates with the zones, and requests routes in the zones for tram passages. Unfortunately, a more detailed discussion of EYAS exceeds the scope of this text.

3.2 Software Implementation

New project development starts with the requirements and specifications analysis. The PLC development department gets several documents with relevant technical data, for example, zone layouts, project specifications, hardware specifications, I/O mapping, etc., among which there is a document called `SW definition table`. It is an MS Excel table containing the data about the software elements used in the project, including the element naming system, specification of the software submodules, safety, and notes.

A dedicated program reads these documents and creates an initial version of the control program using the modules from the module library `G3 core`. The code of the program is then carefully revised, as, firstly, the creator program may produce faulty software due to inconsistencies in the source files, and, secondly, most of the projects have unique requirements from the customer, which are addressed manually.

The signaling control system is implemented in the PLC as a set of separate tasks. The zone control system is realized as one task, while general system control is another task. These tasks are run in loops, a key feature of PLC systems.

Sidebar: PLC operation cycle

⌜ The control program stored in the PLC's memory is run in a continuous loop, with each iteration of the loop known as a "scan". Each scan consists of the following steps:

1. Input scanning: During the first phase of the operating cycle, input signals are sampled and the resulting information is stored in the input memory.
2. Program execution: Next, the program is executed using instances, with the PLC's multitasking capability allowing it to execute multiple instances simultaneously.
3. Output updating: After processing the data, the PLC sends these instructions to the output peripherals.
4. Communication and Diagnostics: If the PLC is part of a larger control system, it may communicate with other controllers or devices through a network.

This cyclical process enables the PLC to continually monitor and control the various systems and processes. ⌟

All the non-safety control program code is written in the C language. The safety control program is developed in the Function Block Diagram (FBD) language and consists of function blocks that are written in the Structured Text (ST) code.

Sidebar: PLC programming languages

⌜ PLC programming languages are used to create instructions to for the controller. These languages may be either text-based or graphical. The IEC/EN 61131-3 standard defines four main languages, one text-based and three graphical [5]. They are:

1. Structured Text (ST): ST is a high-level programming language that is used to create instructions for PLCs in a structured and organized manner. It is similar to the C programming language and is often used for safety-critical functions.

2. Function Block Diagram (FBD): FBD is a graphical programming language that is used to program PLCs using blocks and connections, that visually represent the function between input variables and output variables.

3. Ladder Diagram (LD): LD is a graphical programming language that is used to program PLCs using a ladder-like diagram. It is based on the way electrical circuits are drawn.

4. Sequential Function Chart (SFC): SFC is a graphical programming language that is used to program PLCs using a flowchart-like diagram. It may be used to program sequential control tasks.

PLC manufacturers may add other programming languages support on top of the languages defined by IEC/EN 61131-3 standard. For example, B&R PLCs support the ANSI C language by integrating the GNU compiler. C is a high-level programming language that may be used to program PLCs for non-safety critical functions. It is a powerful and flexible language that is widely used in the programming industry. ⌟

When the control software is functional, it is uploaded to a PLC. It is then debugged through an iterative testing process. This also involves the control cabinet wiring testing. Once thoroughly tested, the cabinet with the uploaded control software is transferred and installed on sight.

3.3 Program Structure

The control program implements a hierarchical modular structure.

The `System` module is the highest level in the hierarchy of functions within the system. It is responsible for managing the overall operation and maintenance of the PLC, including tasks such as diagnostics and system-level functions. This module includes various functions such as system and common functions (CRCs, DateTime), special convert functions (e.g. ASCII, DateTime to string), special string functions (e.g. trim, find/replace), filesystem, and logging functions. `System` module is mandatory for any system, and it is only called once for each PLC. It plays a vital role in ensuring the reliability and stability of the system.

In addition, the `SystemSafety` module is a specialized module used for the safety CPU module control in safety projects. It is also mandatory and is typically used in conjunction with the `System` module. `SystemSafety` is specifically designed to handle safety-critical tasks and functions within the system, such as safety CPU diagnostics, command authorization from the safety PLC in non-safety PLCs, and remote control of the SafeLogic controller (such as downloading the safety program from backup, acknowledging module, SafeKey, or program changes). This module does not have any submodules. All functions related to the PLC's management and operation of the safety system are contained within the module itself.

Next, the `Zone` module is responsible for zone control and determines the state and behavior of the logical elements within a zone. It is worth noting that a physical system element may be a part of several zones at once, or not be a part of any zone at all. A typical example would be a detector shared among multiple zones that overlap in a tram depot, when there is not enough space to install a detector for each zone. Another case is a detector situated outside of the controlled area, typically on a platform on a terminus. Such detectors are designed to be outside of the zone because they are not a part of any route in the zone and only provide information if the platform is occupied. The module also includes the *SAR* function, which allows the zone to automatically recover from a blocked state if certain conditions are met. The `Zone` module also handles global route requests and determines which requests should be processed with the highest priority, efficiently managing the traffic flow.

The `Gate` module manages all entry gates within a zone and handles route request allocation. The gate is the common point for routes and communication devices (also known as requestors) such as KeySwitch, Vetra, Vecom, etc. It collects the requests, serializes them, and sends them to the Zone for processing. The Zone reviews requests from all gates and marks the request with the highest priority. The gate, to which this request belongs, processes it and sends a BUILD command to the requested route. Gate also handles signals from all routes starting at the gate. For example, if a potentially dangerous event (so-called *SPAD*) is flagged by route, the gate may try to solve it or set the zone to the blocked state.

The `Route` module initiates the allocation of route elements for a tram passage, evaluates a successful passage, and flags events such as SPADs and errors. The module is largely invisible in the system, with the gate determining if it is open to requests and if any requests can be processed, as well as handling any illegal activity on routes starting at the gate and determining if the zone is occupied. Additionally, a global request from the zone is not directly connected to the route, but via the gate, which serves as the intermediary in the request processing.

Several modules operate the physical system elements. The `Detector` module provides general tram detection functionality using digital inputs. Detector functions set and reset the CLAIM property of all dependent route elements based on route requests and the detector state.

The `PointMachine` module manages point machines, either electrically or mechanically actuated. It can also raise a warning based on the flood sensor signals (if a point machine is equipped with such).

The `Signal` module is a container for symbols used in the operation of signal lamps. Lamp symbols are defined by combinations of shining and blinking symbols, with each symbol having specific conditions for shining and blinking. The module has commands to calibrate the signal lamp and turn it ON or OFF. The `Signal.Symbol` module is responsible for the operation of individual symbols within the `Signal` module. Any symbol variant may implement current measurement hardware, that must be calibrated before use. The matrix lamps have their own dedicated module called `Matrix`.

The `Requestor` module contains the functions that operate the communication devices. Each device has a corresponding control submodule: `Requestor.Vetra` for Vetra, `Requestor.VecomController` and `Requestor.VecomController.VecomLoop` for Vecom, `Requestor.DigitalRequestor` for the KeySwitch, etc.

The `Cabinet` module is responsible for various control and diagnostic functions within the control cabinet. Its submodules include `Cabinet.BRC` (controls the track circuit BRC unit), `Cabinet.MonitoringModule` (controls an optional cabinet diagnostics module X20CMR111), `Cabinet.ControlPanel`, `Cabinet.Fuse` and `Cabinet.Convertor` (for fuse and converter diagnostics respectively), and others.

The `Heating` module operates the heating subsystem. The parent module computes the heating power from the temperature inputs and precipitation sensor. The `Heating.Contactor` switches heating rods on and off, and `Heating.Contactor.Rod` provides the rods calibration and diagnostics. `Heating.Meteo` makes air and rail temperature measurements and determines the outside weather conditions.

Finally, there is the `GPIO` module that provides control over general-purpose inputs and outputs, and the `Communication` module, which contains functions to control all communication channels in PLC (serial, Modbus, TCP, CAN, etc.).

Several modules, including the `Zone` module, have a safety variant designed specifically for safety systems, which allows the module to read data from the safety system. In case of any general operational error, both these safety and non-safety modules are capable of setting the zone to the Emergency Inhibit state, effectively suppressing most of the control system functionality.

3.4 Zone Logic

The recapitulate, a zone is basically a defined area, controlled by one PLC (or one PLC and one safety PLC in the case of a safety zone). Among other functions, it handles global route requests and determines which requests should be given the highest priority in order to efficiently manage the traffic flow. The System Auto Recovery function allows the zone to automatically recover from a blocked state after an unauthorized tram passage if certain conditions are met. Additionally, the system can also be manually recovered by the operator.

3.4.1 States

The signaling control system operates in three different states, each with its own characteristics and limitations.

The *Normal state (NORM)* is the standard operating state of the system. All functions of the signaling system are available, and route requests are processed normally according to the operating mode of the system. The signaling system can operate in either automatic or manual mode. In automatic mode, routes are requested automatically (for instance, by the EYAS) and commands from the tram driver are ignored. In manual mode, alternative methods of requesting routes can be used (for instance, in case the EYAS is out of service).

The *All Blocked state (AB)* may be activated due to several conditions, such as a driver operational error or a noncritical system component error. Only some functions of the signaling system are available. All switch points in the zone are electrically blocked. The gate command memory is deleted, new route requests are not accepted, and routing logic and route building are suspended (a rare exclusion may be the SPAD route building in case of a potentially dangerous route event). Entrance into the zone is controlled by an operator. All entrance signals usually display the STOP symbol, and the matrix signals (if present) provide information on the system status.

The *Emergency Inhibit state (EI)* is activated when a critical hardware fault is detected in a system component. The system is shut down and requires service intervention. Almost no signaling system function is available. Similar to the AB state, route requests are not accepted, and all the switch points in the zone are electrically blocked. Entrance into the zone is controlled by an operator. All entrance SG signals and PPI signals are usually off. Alternatively, SG lamps may be defined to display the STOP symbol. However, the Emergency Inhibit state may be caused by an error in the STOP signal itself. In this situation, all signals are to be signaling the EI state with STOP signs, but the signal with the error will not be able to display it, resulting in an undefined system state. The matrix signals (if present) provide information on the system status. The driver must follow the internal traffic rules. The system allows remote monitoring and troubleshooting and signal calibration.

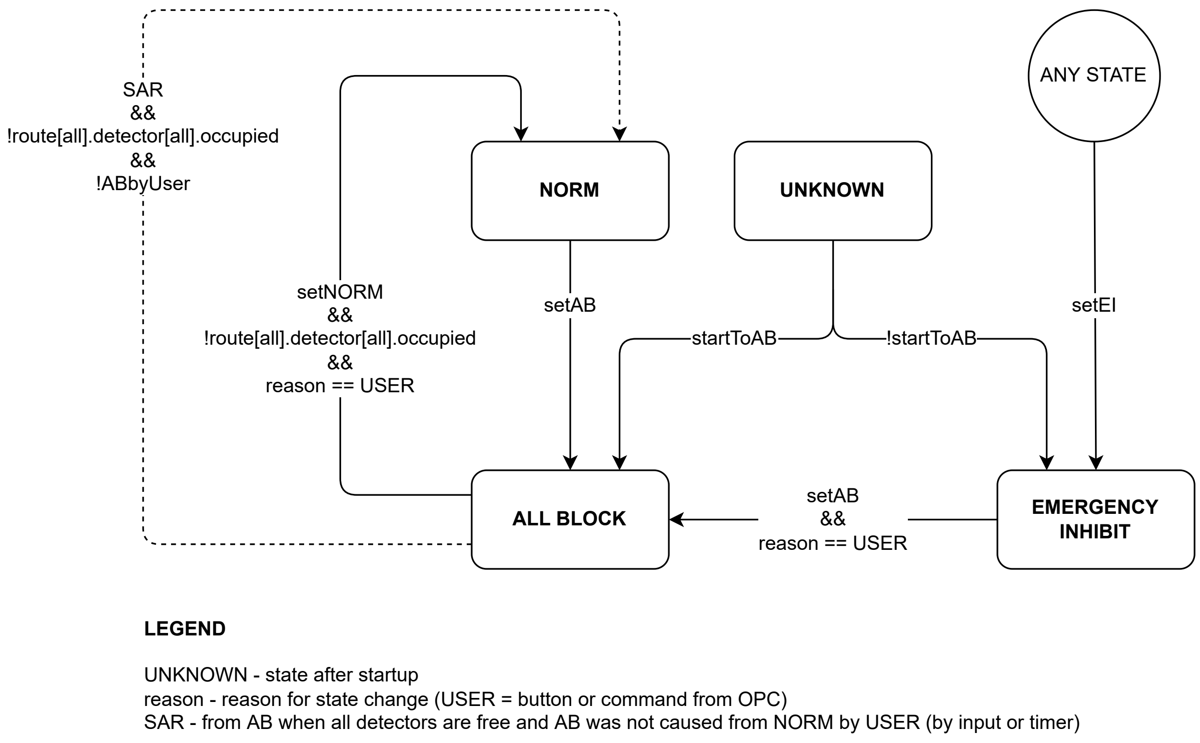

SAR
&&
!route[all].detector[all].occupied
&&
!ABbyUser
NORM
UNKNOWN
ANY STATE
setNORM
&&
!route[all].detector[all].occupied
&&
reason == USER
setAB
startToAB
!startToAB
setEI
ALL BLOCK
setAB
&&
reason == USER
EMERGENCY INHIBIT
LEGEND
UNKNOWN - state after startup
reason - reason for state change (USER = button or command from OPC)
SAR - from AB when all detectors are free and AB was not caused from NORM by USER (by input or timer)

Figure 9: Zone state machine diagram [13]

3.4.2 SAR

The automatic System Auto Recovery function is a feature of the signaling system that is designed to automatically reset the system from the All Blocked state to the Normal state in the event of an operational error. It is only available if the system is switched to AB state in specific ways, such as after a potentially dangerous route event (SPAD) has occurred, for example, an unexpected change of the passage direction, or start the passage before the GO symbol has been displayed. It is not available if the system has been switched to AB state manually or if any tram is present in the zone.

The SAR function may be executed by a time delay and by the correct passage of a chosen route (the latter can be implemented on the customer's request and is usually used in systems where it might not be clear to the driver whether the zone is functioning normally or is blocked). If all position detectors in the zone remain unoccupied for a defined period of time (known as the T_{SAR} timeout), the system automatically switches to the NORM state. However, if any detector is occupied during this time, the SAR function is not activated, and the system remains in the AB state.

The SAR function can only be implemented in zones that are fully covered with the position detectors, ensuring that there is no area where a tram may be present without being detected. Otherwise, if a tram was present within the zone, and the switch from the All Blocked state to the Normal state was activated (either by the SAR function or an operator), a potential safety risk would arise. The system would be unaware of the presence of the tram, and any point machine may be switched as a result. The driver of the tram may also be unaware that the zone has been unblocked, further increasing the risk of an accident. It is important to ensure that all necessary precautions are taken when using the SAR function to avoid potential hazards.

3.5 Route Logic

A route is a specific path that a vehicle follows to reach its desired destination. It is defined by its starting point (entry gate equipped with a SG signal lamp), an ordered array of elements (detectors, point machines, and crossings), and the position of the turnouts required to build it. Gates themselves are the entrance points to the zone and may lead to multiple routes. A typical route layout and its hardware components are visualized on the fig. 10.

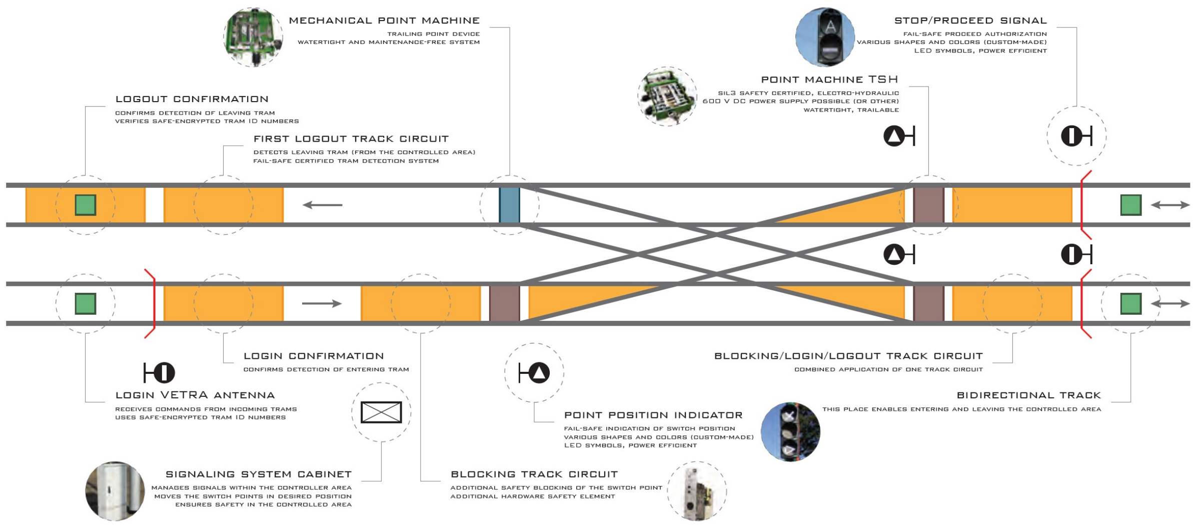

MECHANICAL POINT MACHINE
TRAILING POINT DEVICE
WATERTIGHT AND MAINTENANCE-FREE SYSTEM
STOP/PROCEED SIGNAL
FAIL-SAFE PROCEED AUTHORIZATION
VARIOUS SHAPES AND COLORS (CUSTOM-MADE)
LED SYMBOLS, POWER EFFICIENT
POINT MACHINE TSH
SIL3 SAFETY CERTIFIED, ELECTRO-HYDRAULIC
600 V DC POWER SUPPLY POSSIBLE (OR OTHER)
WATERTIGHT, TRAILABLE
LOGOUT CONFIRMATION
CONFIRMS DETECTION OF LEAVING TRAM
VERIFIES SAFE-ENCRYPTED TRAM ID NUMBERS
FIRST LOGOUT TRACK CIRCUIT
DETECTS LEAVING TRAM (FROM THE CONTROLLED AREA)
FAIL-SAFE CERTIFIED TRAM DETECTION SYSTEM
LOGIN CONFIRMATION
CONFIRMS DETECTION OF ENTERING TRAM
LOGIN VETRA ANTENNA
RECEIVES COMMANDS FROM INCOMING TRAMS
USES SAFE-ENCRYPTED TRAM ID NUMBERS
BLOCKING/LOGIN/LOGOUT TRACK CIRCUIT
COMBINED APPLICATION OF ONE TRACK CIRCUIT
BIDIRECTIONAL TRACK
THIS PLACE ENABLES ENTERING AND LEAVING THE CONTROLLED AREA
POINT POSITION INDICATOR
FAIL-SAFE INDICATION OF SWITCH POSITION
VARIOUS SHAPES AND COLORS (CUSTOM-MADE)
LED SYMBOLS, POWER EFFICIENT
SIGNALING SYSTEM CABINET
MANAGES SIGNALS WITHIN THE CONTROLLER AREA
MOVES THE SWITCH POINTS IN DESIRED POSITION
ENSURES SAFETY IN THE CONTROLLED AREA
BLOCKING TRACK CIRCUIT
ADDITIONAL SAFETY BLOCKING OF THE SWITCH POINT
ADDITIONAL HARDWARE SAFETY ELEMENT

Figure 10: Route layout elements and hardware components [2]

A *route request* is a request for a specific route to be created. This request can be made automatically or manually, depending on the system and the circumstances, and is assigned to the gate where the route starts. Routes are usually requested by a requestor (Vetra, Vecom, etc.). Each requestor is paired with a gate. A gate may have multiple requestors paired to it, and the signals from all requestors are summarized. A route is considered to be *compatible* with another route if both routes can be active at the same time without the risk of a tram collision within the zone. If there is a such risk, and both routes cannot be active at the same time, they are considered to be *colliding routes*.

All elements within a route have a property called `CLAIM`. The concept of `CLAIM` is the following: if an element is claimed for a specific route, its `CLAIM` is set to the route number. If the element is unclaimed and available for use by any route, its `CLAIM` is set to zero. Position detectors reset their `CLAIM` when the tram leaves the detector (on the falling edge of the occupation). All other elements, on the other hand, reset their `CLAIM` when the next element in the array loses its `CLAIM`.

It's important to note that the first and last element in any route must be a detector. The order of elements in a route is usually determined by the signaling layout, but it can be customized to meet the specific needs of a project.

3.5.1 States

A route has several different states that describe its current status.

The *FREE* state is the default route state. The route is not claimed, and all the route elements are not occupied. The route can be requested in this state.

The *BUILD* state indicates that the route has been reserved and is being created. Turnouts are being set in the correct direction, the entrance signal prohibits entry.

The *READY* state implies that the route is ready for the tram to enter. Turnouts are in position and blocked, and the entry SG lamp displays the "GO" symbol.

The *OCCUPIED* state marks that the tram is actively traveling through the route. It is activated when the first position detector of the route is occupied and ends when the last position detector in the route of released.

The *ERROR* state signalizes that a route error has been detected (for instance, a route passage error). In case this occurs, the signaling system switches to the AB state.

The *RELEASE* state can only be set by a CANCEL command from the READY state. In certain special cases, however, it is possible to set RELEASE from OCCUPIED if all previously occupied detectors are freed. This is typically only possible in situations where the route is fully covered by detectors and non-standard movements (such as reversing) are allowed on the route.

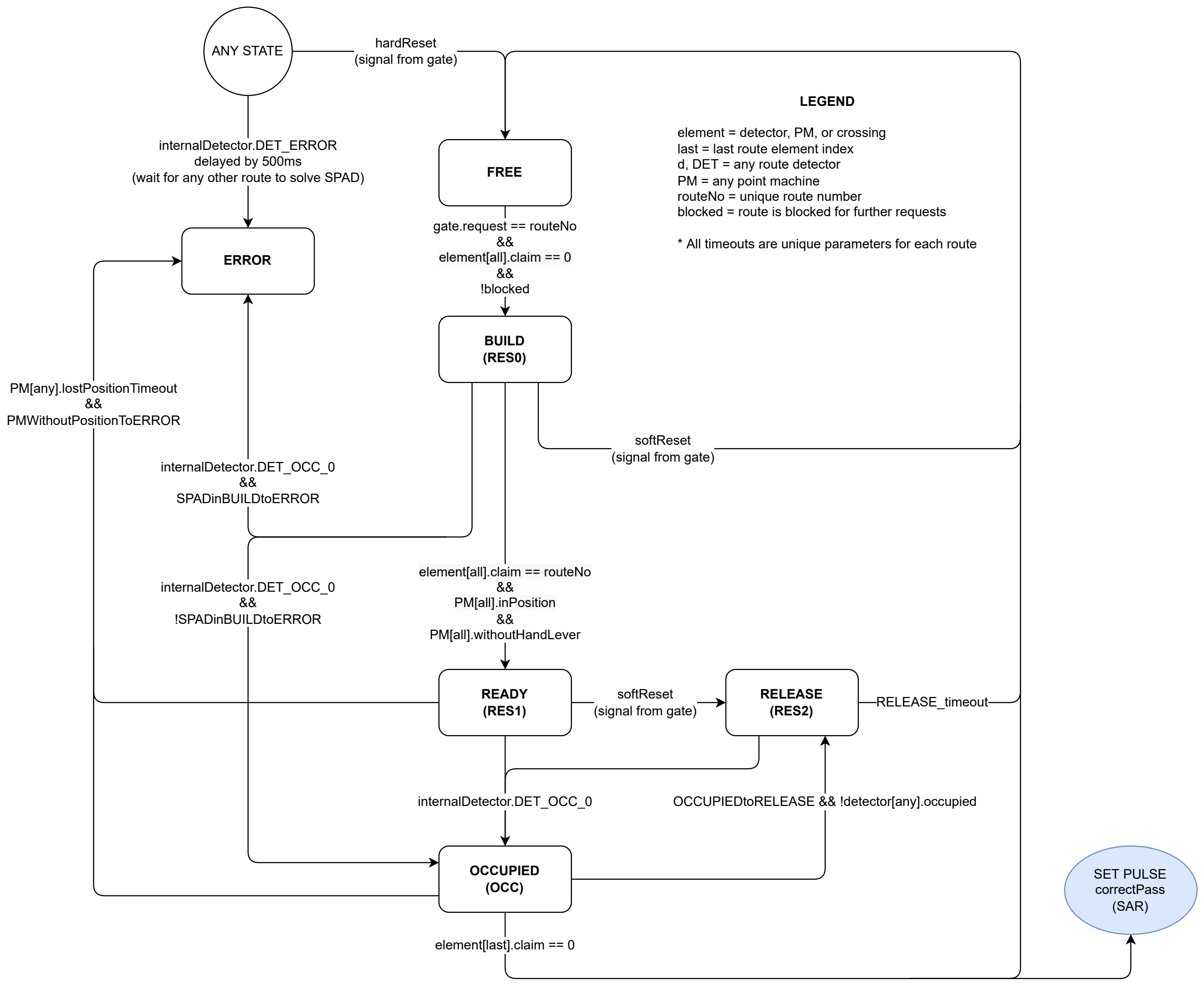

ANY STATE
hardReset (signal from gate)
internalDetector.DET_ERROR delayed by 500ms (wait for any other route to solve SPAD)
ERROR
FREE
gate.request == routeNo && element[all].claim == 0 && !blocked
BUILD (RES0)
LEGEND
element = detector, PM, or crossing
last = last route element index
d, DET = any route detector
PM = any point machine
routeNo = unique route number
blocked = route is blocked for further requests
* All timeouts are unique parameters for each route
PM[any].lostPositionTimeout && PMWithoutPositionToERROR
softReset (signal from gate)
internalDetector.DET_OCC_0 && SPADinBUILDtoERROR
internalDetector.DET_OCC_0 && !SPADinBUILDtoERROR
element[all].claim == routeNo && PM[all].inPosition && PM[all].withoutHandLever
READY (RES1)
softReset (signal from gate)
RELEASE (RES2)
RELEASE_timeout
internalDetector.DET_OCC_0
OCCUPIEDtoRELEASE && !detector[any].occupied
OCCUPIED (OCC)
SET PULSE correctPass (SAR)
element[last].claim == 0

Figure 11: The route state machine diagram [13]

States of a route can be summarised by three general operational phases: the blocking phase (occurs when a tram enters the controlled area and includes the route request, the BUILD state, and the READY state), the occupation phase (occurs when a tram passes through the zone and includes the OCCUPIED state), and the release phase (occurs when a tram leaves the zone and includes the RELEASE state and the FREE state).

3.5.2 Requesting and Request Memory

Normally, the TWC (Tram-To-Wayside) communication system requests the route once it receives a signal from an approaching tram. An alternative way to request a route is from the traffic control center or via the HMI located in the control cabinet). Once the route request is received, the signaling system evaluates whether it is possible to build a route right away by checking the status of each route element. If all needed elements are available, the route is considered compatible and can be built.

If the request cannot be resolved immediately, it can be stored in the gate request memory. It helps to ensure that trams can pass through smoothly and efficiently, even if there are multiple trams approaching at the same time. Such memory is implemented for each entry gate. Requests are added to the memory as soon as a tram is identified at the communication device login antenna, and its destination code or ID is recorded. Requests are then executed as soon as possible to allow the tram to pass through the zone. A request is removed from the memory the moment the tram occupies the first position detector of the route. In manual mode, the route request memory should be used to request a route for an approaching tram in advance, so that the route can be prepared in time for the tram's arrival.

It is worth noting, during each scan cycle of the PLC, the zone processes only one global request. After this request is processed, the zone reevaluates the other requests from all the gates again (in the next scan cycle) and decides which route can be processed next - the new claims are taken into consideration there. This ensures that requests for colliding routes cannot be evaluated in one scan cycle.

3.5.3 Build

To build a route, the system reserves the necessary elements for that route and prevents other routes from being built in parallel if these routes are colliding (colliding routes cannot be built simultaneously due to the global requesting algorithm described in section 3.5.2 above). The route checks the `CLAIM` properties of all its elements and evaluates if the summarized claim is zero. If it is, the route is switched to BUILD and a claim (route number) is sent to all its elements.

During the building, the entrance signal lamp displays the "STOP" symbol and the turnouts are set to their desired positions. If a turnout does not reach its desired end position (either stuck in an intermediate position or not switched at all), the route cannot be successfully built and remains in the BUILD state. The driver is alerted of the incorrectly set turnout (for example, through a warning text on the matrix lamp) and needs to manually adjust it using the alignment bar. Once all the turnouts are correctly set in their end positions, the route status changes from BUILD to READY. The route is then ready for use, and the entrance signal lamp displays the "GO" symbol, allowing the tram to enter the zone.

3.5.4 SPAD

A "Signal Passed at Danger" (SPAD) event occurs when a tram enters a zone without permission while the route is not yet ready for passage and the entry lamp displays the "STOP" symbol. When this happens, the route sends the information to its entry gate, which determines whether the SPAD can be resolved. In some cases, customers may prefer for the zone to switch to the AB state immediately after a SPAD event, regardless of whether it can be resolved. The dispatcher is also notified of the incident.

If SPAD is solvable, the signaling system attempts to construct a route for the tram to proceed. It takes into consideration the current position of the point machines, the availability of individual position detectors, and other relevant factors. If, based on these conditions, a route can be established, it is constructed immediately, and the signaling system remains in the NORM operating state. On the contrary, if the SPAD cannot be resolved through route construction, the signaling system switches to the All Blocked (AB) state.

3.5.5 Passage

To detect the presence of a tram as soon as it enters the zone after the "GO" signal, a position detector is placed after each entry gate. When this detector is occupied, the corresponding route, prepared for its passage, changes its status from READY to OCCUPIED. It may also happen that the tram enters the zone in the RELEASE state. This case is discussed in paragraph 3.5.6 below.

After the start of the passage, the route elements are sequentially released. When a tram passes through a position detector, occupying and releasing it, and subsequently departs, the detector is marked as available and can be used to reserve other routes. All dependent route elements, such as point machines or route cross sections, are also released. If a certain position detector and its dependent route elements are part of multiple route layouts at the same time, this logic allows new routes to be built before the tram has passed the previous route and left the zone, leading to an increased traffic flow through the zone of the zone. It is worth noting that the sequential release function is defined by the ordering of the route elements array. If all non-detection elements are placed before the last element in the array, then the sequential unblocking function is void - all non-detection elements will be unclaimed by the last detector.

It is also worth mentioning the point machine locking function that prevents a turnout from being switched when a tram is passing. This is a software function implemented in the PLC control program. The locking function is triggered when at least one position detector corresponding to the turnout is occupied, the route through the turnout is active, or the control system is in the All Blocked (AB) or Emergency Inhibit (EI) state. In the case of a safety zone, the locking function is implemented as a safety function, and its control is handled by the Safety PLC. The control logic is usually the same in safety and non-safety modules to maintain consistency in the company's control software.

The signaling system continuously checks the correctness of the tram's passage through the zone. While the prepared route is not passed correctly, and the tram is not detected by any detector, the route remains in the OCCUPIED state, and the system continues waiting for the passage to finish. A detector, placed before each exit gate, detects the tram's exit from the zone. When this detector is released (along with all preceding ones), the route switches to FREE, and the tram is unregistered.

It is not possible to change the direction of the tram in an active route within the zone. If the tram changes direction and reoccupies an already freed detector, the signaling system evaluates this as an incorrect passage, and the route enters the ERROR state. If the position of any point machine is lost for more than a defined period of time, an operational error is raised, too. These situations lead to the signaling system switching to the All Blocked (AB) state.

The systems designed by the company from scratch are usually fully covered with position detectors. Unfortunately, it is not always the case if the company only provides the control system for an existing physical layout. Dense detector covering ensures that the tram is always detected by at least one detector as it passes through.

3.5.6 Cancellation

Routes can be canceled for various reasons. If a route is only in the command memory and has not yet started being built, it can be directly deleted from the memory by the operator, rather than be canceled. If a route is in the BUILD state or the READY state, and a tram has not entered yet, it can be canceled either by the dispatcher or the operator or by canceling directly from the switchboard on the HMI panel. However, if a tram has started the passage, the route can only be canceled by switching the entire zone to the AB state.

If a route is canceled in the BUILD state, when the entrance has not yet been permitted and the "stop" symbol is still displayed on the entry lamp, it changes its state directly to FREE. If it is canceled in the READY state, it first enters the RELEASE state, and the cancellation timer, known as the "RLS timeout", starts. During RELEASE, the "STOP" symbol is displayed on the entrance signal lamp. The timeout is in place to hold a reserved route if the cancellation command is issued too late for the approaching tram to stop. If a tram enters the route within the time limit, the route is not canceled and goes into the OCCUPIED state. Otherwise, it switches to the FREE state. The cancellation timeout (RLS timeout) is set individually for each route and is calculated as the sum of the braking time required for the tram to stop from its maximum initial speed (rounded up to the nearest second) and the driver reaction time (2 seconds). The braking is considered a uniformly decelerated motion with a normal deceleration rate.

If the route were to switch from the READY state to the FREE state immediately without the RELEASE state being activated, some point machine on the route could be switched in between the cancellation of the route and the entry of the approaching tram, it is possible that derailment could occur. To avoid this risk, it is necessary to activate the RELEASE state prior to switching the route from the READY to the FREE state.

3.6 Communication

The communication between the control system and other actors of the global control system is realized through the OPC UA communication protocol.

OPC Unified Architecture (OPC UA) is an open-source machine-to-machine communication protocol that is designed to provide interoperability between different devices and systems in the automation and control industry. It is developed by the OPC Foundation and is defined by the IEC 62541 standard [6]. The protocol is based on a client-server architecture, where clients request data from servers and servers provide the requested data to the clients.

A specific piece of information or functionality within a server is represented by a logical object called a *node*. Nodes are organized in a hierarchical tree structure within the server's address space. Each node has a unique identifier and a set of properties that describe its characteristics and behavior. There are several different types of nodes in OPC UA, including variables, methods, and objects, which can be used to represent different types of information and functionality.

Nodes can be accessed and manipulated using OPC UA client applications, which can connect to an OPC UA server and request data or invoke methods on specific nodes. This allows devices and software programs to exchange data and control each other. An important detail is that OPC UA uses a publish-subscribe model, where clients can subscribe to data from servers and receive updates when the data changes. This allows for efficient communication, as clients only receive updates when the data they are interested in changes.

One of the key technical features of OPC UA is its ability to support a wide range of data types, including numerical, string, and Boolean data, as well as more complex data types such as structures and arrays. It also allows users to define custom data types and structures that can be used in communication between devices.

In terms of security, OPC UA uses a combination of encryption and authentication to provide secure communication between devices. It supports both symmetric and asymmetric encryption algorithms and includes mechanisms for the secure authentication of clients and servers.

Another communication protocol, SHV RPC, is widely used in the global control system for tasks like system visualization, EYAS communication, etc. This protocol is developed by the company and is based on the RPC communication protocol. Currently, the implementation of the direct PLC signaling system support of the SHV protocol is in progress. Similar to OPC UA, it uses a client-server request-response model. Both RPC and OPC UA also implement a concept of subscriptions.

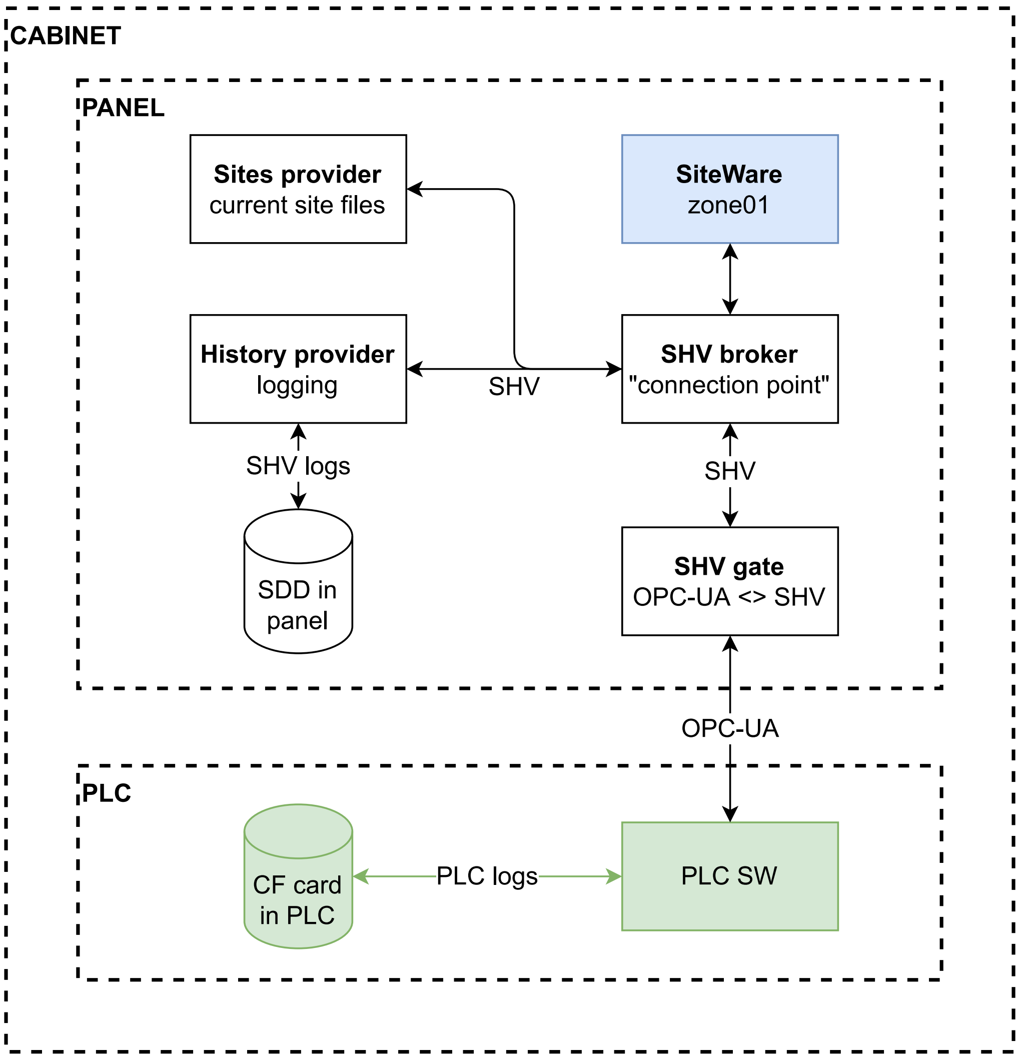


Figure 12: The diagram illustrates how PLC communicates with the control panel in the cabinet. The communication in the global control system, which the panel is a part of, is realized via the company's custom SHV communication protocol. SHV gate module converts the OPC data packet from the PLC to SHV data packet. [13]

4 Analysis of the Current Testing Process

Conducting an analysis of the current testing process is an important step in the development of a new PLC code testing program. It helps to establish a baseline understanding of the existing system and provides valuable insights for further development. By conducting an in-depth review, one can both identify best practices or approaches that could be incorporated into the new program, and spot challenges or problems that the new program should aim to address.

In order to thoroughly analyze the current implementation of the testing process, it is necessary to consider a number of factors. First, it is essential to examine the testing process as a whole. This includes considering the various types of tests that are used to evaluate the system's functionality and reliability, as well as reviewing any standards or guidelines that must be followed. It may be also necessary to consult with subject matter experts who can provide additional insights.

The steps involved in the testing process must be examined, too. This includes any data input or preparation that is required, the execution of the test itself, and the analysis of the results. Also, the files used in the current testing program must be reviewed. This includes any input or output files, as well as any intermediate data or results that are generated during the testing process.

Then, the testing software structure and design must be studied. This includes a general testing program code review as well as a deeper examination of its software modules. It would be also helpful to look into the implementation of the system elements digital twins' library and review any accompanying documentation or instructions. After that, the maintainability, scalability, and flexibility of the current software can be evaluated.

Finally, the ease of use and effectiveness of the current testing program must be considered. This includes any challenges or limitations that may impact its ability to accurately and efficiently test code. It may be helpful to gather feedback from users of the current program in order to fully understand their experiences and challenges. It is also necessary to assess any specific requirements or needs that may exist in regard to the test environment.

With a thorough analysis of the current testing solution, the new program can be designed to meet the needs of the user and to effectively and efficiently test PLC code.

4.1 Stages of Testing

A new PLC control system is usually tested in two main stages. Initially, software testing is conducted by the PLC developers. After that, implementation and wiring testing is performed by the testing lab personnel.

4.1.1 Software Testing

Software testing involves simulating the control system in a virtual environment. Besides a control task, which contains the control program, PLC also runs a test task consisting of digital twins of the physical control system elements. These tasks communicate via a common PLC memory data structure, with the inputs and outputs of both control and test function blocks being virtually connected. Connection to the safety program is achieved through B&R SafeLogic data exchange. The command signals are sent to the system elements by a higher-level test program via a chosen communication protocol (currently OPC UA).

Sidebar: Digital twins

A digital twin is a digital replica of a physical object or system. It can be created using either a measurement-based system identification or mathematical modeling of the object's characteristics and behavior (or a combination of both), and it is used to simulate the object's performance and predict its response to various conditions.

The concept of the digital twin originated in the manufacturing industry, where it was used to design and test new products before they were built. Today, digital twins are being used in a wide range of fields including transportation. One of the key benefits of a digital twin is that it allows testing and optimizing the performance of a physical object or a system without having to build it in the real world. This can save time and money, and it can also help to reduce the risk of failure or downtime.

4.1.2 Implementation and Wiring Testing

Once the cabinet hardware (a PLC, peripheral I/O cards, sensors, etc.) is prepared, the control software is uploaded to the cabinet PLC and the whole control system is tested again. This step, known as implementation and wiring testing, is similar to software testing, with two main differences. First, the test environment is now uploaded to a separate test PLC. Second, the actual working conditions (i.e the inputs and outputs of the digital twins) are simulated by the real hardware models of the physical elements rather than by the digital twin functions. Even though the control PLC, the test PLC, and the models are physically connected by wires, the communication with the system elements is very similar to the software testing one.

The hardware models are electrical circuits that mimic their real-world counterparts: track circuits, point machines, lamp signals, route requestors, etc. For example, the position detector simulator mimics the outputs of a real track circuit. Physically, this simulator is an RLC circuit with parameters corresponding to the real device with the possibility of shortening (a short circuit is realized as a potential-free relay contact). Its input command is a control command for the track circuit occupation, and the output signal is the above-mentioned relay contact. This simulator enables the simulation of the occupation and the release of the track circuit. Common test cases that involve its usage are verification of the track circuit functioning and validation of the correct system logic behavior during a tram passage, where the simulator is a part of the whole zone simulation system [11].

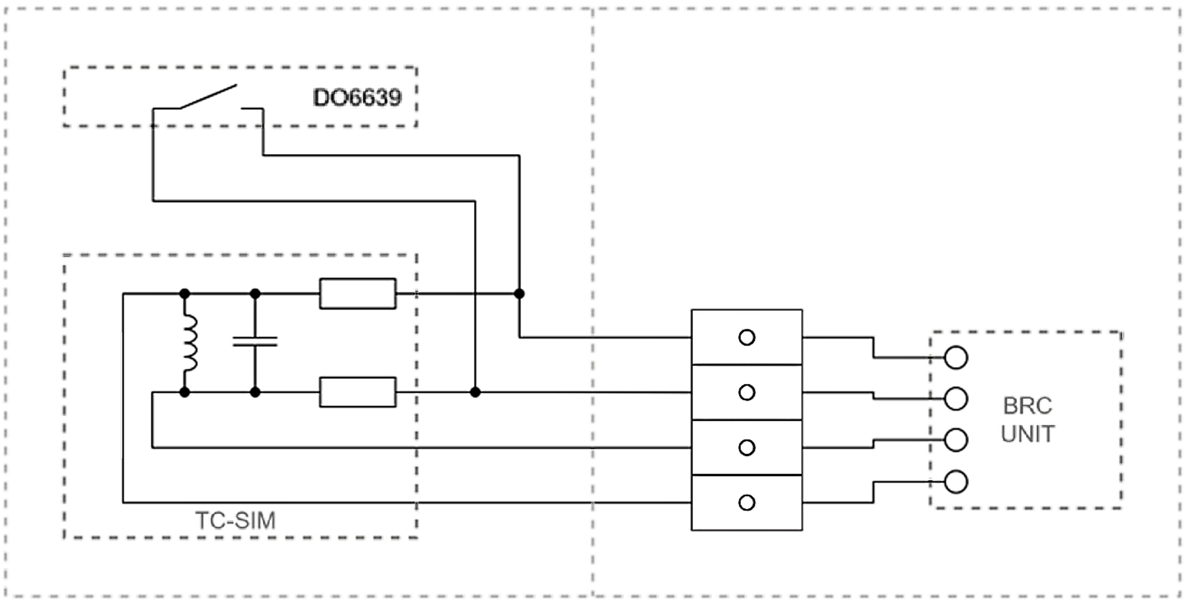


Figure 13: Electrical schema of a track circuit simulator. The simulator is connected to the BRC unit in the control cabinet via the connection terminal. The output relay of the simulator is B&R X20DO6639 240 VAC / 30 VDC relay module [11].

4.2 Testing Scenarios

A simple software test typically involves providing input to the software and then observing the results to determine whether the software is functioning as expected. Such a test can be thought of as having three main parts: an input command, expected results, and actual results. After the input is provided to the software, a tester acquires the actual results and compares them with the expected results. Expected results may include desired results, undesired results, and timeouts. Undesired results are outcomes that should not occur as a result of the test steps being executed, such as errors or exceptions. Timeouts are defined intervals within which the test steps should be completed. If the actual results match the desired results, the test is considered to have passed. If the actual results match the undesired results, or test steps do not complete within the defined timeout, the test is considered to have failed.

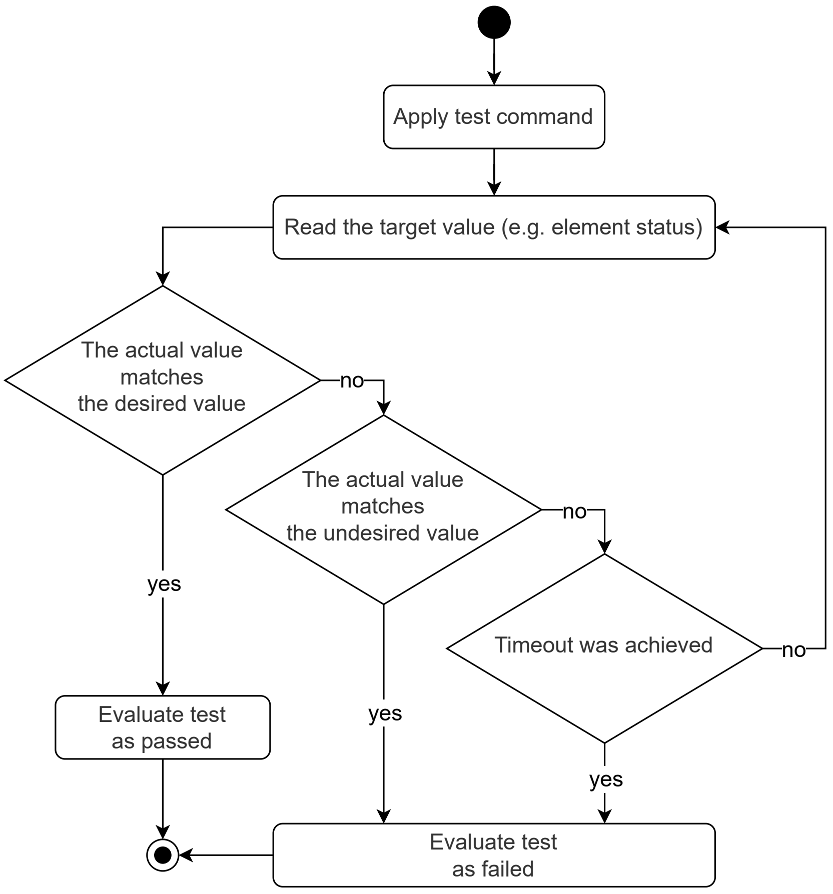

Apply test command
Read the target value (e.g. element status)
The actual value matches the desired value
no
yes
The actual value matches the undesired value
no
yes
Timeout was achieved
no
yes
Evaluate test as passed
Evaluate test as failed

Figure 14: A simplified flowchart of a test

From a technical point of view, the majority of the performed tests can be classified as system tests. It is almost always necessary to simulate multiple system elements at the same time in order to properly test the software. For example, in order to test a signal lamp, it is necessary to simulate the lamp and also the control panel, and in order to test the control panel, it is also necessary to simulate the zone. That is why all the system elements are simulated together, and the test environment mirrors the entire controlled system. Initial tests of the core control or test function blocks could be classified as unit tests. Such testing occurs in case these functions are newly implemented or have been modified. Such tests are out of the scope of this text.

A test scenario is a combination of simple tests described above. Technically, even a single test could qualify as a scenario. However, scenarios are usually more complex and involve several smaller tests, run either sequentially or in parallel.

The terminology used in the company divides the test scenarios into several groups, including static tests and dynamic tests. Static tests mostly cover single system elements functions (for instance, point machines switching), while dynamic tests simulate a passage of a single tram or multiple trams. These scenarios model different situations that may occur in a real-world system, for example, activation of signal lamps, route requesting, route building, SPADs, etc.

4.3 Software Implementation

Both software and implementation and wiring testing are performed by using a Python program `pytester`, which manages the testing process, and a set of helper Python scripts for data acquisition and test files generation. The program is only capable of communication via the OPC UA protocol. The digital twins of the physical system elements are stored in the `El_Testing` library, which is a part of the PLC testing software. The testing process is semi-automated, meaning that human supervision is needed to manage the global steps (e.g. test scenarios creation, test and control element communication paths generation, `pytester` setting, etc.), but the testing itself, once started, is autonomous.

4.3.1 `El_Testing` library

`El_Testing` library contains the function blocks that emulate the physical system elements. It includes three modules: `TST_Gen` module, `TST_Comm` module, and `TST_Periph` module. Each module contains files with the function blocks' source code and the definitions of the OPC communication data types in a form of C structs. Every function block has a set of variables and an OPCType as an interface for interaction with it [12].

Sidebar: C structs

⌜ In C, a `struct` (short for "structure") is a composite data type that groups together values of different types. It is similar to a class in object-oriented programming, but does not have any methods and does not support inheritance. Structs are useful for grouping together related data and creating complex data types. ⌟

The test zone is created by a dedicated Python script. The script parses the files of the control system to gather data about the system elements and generates a counterpart test zone structure. It creates files with instances of the function blocks from `El_Testing` library and their OPCTypes definitions, which are then stored in the test PLC memory. Any function block is able to transmit and receive messages from any OPC UA system.

The `TST_Gen` module This module includes general-purpose testing functions, that are able to simulate digital and analog inputs and outputs of the system devices.

The testing input functions are counterparts of the control output functions. For example, `fTestGeneral_DigitalInput` is a counterpart of the control functions of the cabinet internal lighting and cooling fan, control panel LEDs, GPIO outputs, etc. `fTestGeneral_AnalogInput` mirrors the analog input signals.

Similarly, the testing output functions are counterparts of the control input functions. `fTestGeneral_DigitalOutput` pairs with the control functions of the control panel non-safety buttons, electrical fuses inputs, UPS inputs, GPIO inputs, and others. `fTestGeneral_DigitalOutputCompl` serves as a digital twin for control input functions with complementary inputs, namely control panel safety buttons or both safety and non-safety position detectors. Finally, `fTestGeneral_AnalogInput` accompanies control analog input functions such as cabinet power input and catenary voltage input.

The `TST_Comm` module This module includes functions for testing all universal communication control functions. Its functions `fTestComm_Serial` (running in 100ms cycle) and `fTestComm_CommSerial` (running in 10ms cycle) mimic the serial communication devices, and `fTestComm_CAN` (running in 100ms cycle) and `fTestComm_CommCAN` (running in 10ms cycle) are used specifically for SHV CAN testing.

Universal functions `fTestComm_Serial` and `fTestComm_CAN` are mainly used for the transmission of messages in the form of raw HEX strings that represent JSON strings. The remaining functions, `fTestComm_CommSerial` and `fTestComm_CommCAN`, may be employed in other functions, in instances where the communication protocol is known. Such functions convert input data into raw messages, serialize said messages, and subsequently transfer them to serial communication modules utilizing the `fTestComm_CommSerial` and `fTestComm_CommCAN` functions, which operate on a 10ms cycle.

The `TST_Periph` module This module is a collection of functions designed for testing various system peripherals.

`fTestPointMachine_PME` function block is a digital twin of a point machine. It may be set to have one, two, or three tongue position sensors on each side and the sensor of the manual lever rod presence. It enables setting the position of the point machine with a settable switch time. By default, the switch time is 2500 ms, but it can be adjusted as needed to suit the specific requirements of the railway system.

`fTestSignal_Symbol` function block is used for testing symbols (both with and without current measurement capability). It allows setting a symbol to the warning and to the error states, which corresponds to situations when the measured current value on the symbol is below 85% or 75% of the reference value, respectively. It also enables setting a current value that corresponds with the situation when the symbol is off.

`fTestCabinet_Cabinet` function block represents the control cabinet and its components. For example, `fTestCabinet_Panel` provides a software simulation of the control panel in the cabinet, including its hardware buttons. These functions are only used during the software testing since the implementation and wiring testing assumes that the real cabinet hardware has been prepared, and the control software has been uploaded to the cabinet PLC.

`fTestHeating_Rod` function block simulates a heating rod (both with and without current measurement capability). It is can simulate various warning and error states with different abnormal current values. `fTestHeating_MeteoELL` simulates a meteo station with temperature and precipitation sensor inputs. Although it is primarily used for software testing, it could be used during implementation testing, too.

`fTestRequestor_Vetra` function block serves as a Vetra device digital twin and represents its data and configuration. It can be set to be used both for software testing and for implementation testing (for the latter, a 10ms communication function `fTestComm_CommSerial` and its `TestCommSerialType` OPCType are responsible for the message exchange process). `fTestRequestor_Vecom` simulates a Vecom device can be used in a similar manner to the`fTestRequestor_Vetra` function block.

4.3.2 Testing Files and Their Structure

Test files for a specific zone can be divided into two main groups: *test scenarios* and *configuration files.* Scenarios are used to provide input data for testing programs and contain input values as well as the expected output values for those inputs. The testing program reads the test file, runs the program being tested with the input values from the file, and compares the output of the program to the expected output values from the file. Configuration files are used to store settings for a particular installation of the test program. They contain key-value pairs that represent configurations and are often human-readable and easily editable.

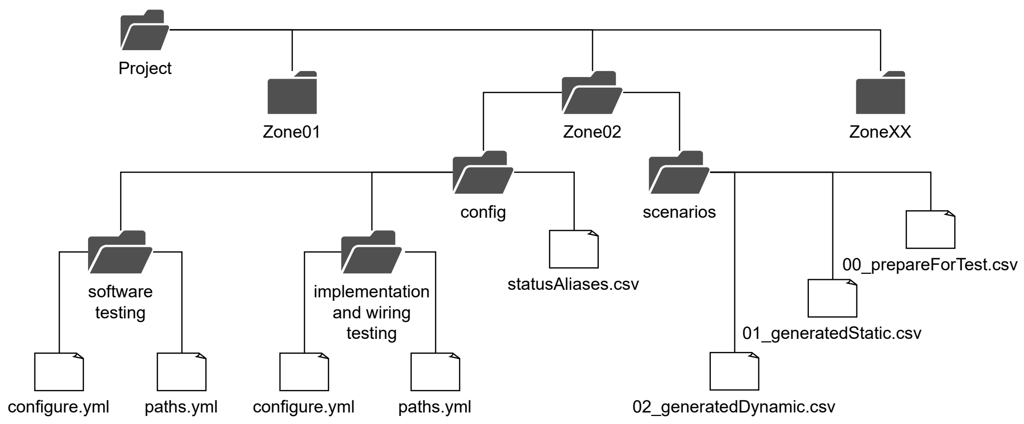


Figure 15: File structure of a test project

`scenarios` files Scenarios files are `CSV` tables of testing actions and reactions, organized into rows and columns. Each row of a scenario represents a simple test described in section 4.2. The first column contains a testing action that is taken, and the second and third columns contain the expected value if the action is successful and if the action fails, respectively. The third column may also contain a test timeout. It is worth noting, that multiple success conditions are bound with a logical `AND`, while multiple fail conditions are bound with a logical `OR`. Each test scenario is marked by a header line starting with a `#` symbol. This line contains the index of the test scenario as well as a brief explanation of its purpose.

The testing starts with a resetting sequence: the `System.resetAll` command is applied. This command resets all the digital twins of the control system elements. To ensure the control `Zone` is in the correct state after the previous test has ended, it is also possible to run `Zone.cmdSetAB` and `Zone.cmdSetNORM` commands.

`00_prepareForTest.csv` file contains a set of commands that are used to calibrate the signal lamps. This calibration process is typically only performed once, following the compilation and upload of the control program to a PLC. It is important to complete this prerequisite step, as failing to do so may result in the Zone remaining in the EI state. Once the calibration has been successfully completed, the signal lamps are able to function as intended, and further testing may be performed.

`01_generatedStatic.csv` file contains static testing scenarios. These include testing of the position detectors' functions (`occupy` and `free` commands), testing of the point machines' functions (`setTrue` and `setFalse` commands for the tongue detection sensors), and testing of lamps and lamp symbols (`setWarning` and `setError` for symbols with the correct measurement capability).

`01_generatedDynamic.csv` file contains dynamic testing scenarios. These are testing sequences for single tram passages through a controlled zone and combinations of two simultaneous tram passages. The passages are requested by directly inputting the request JSON string into the memory of the entry gate of the requested route. If the passage is permitted, the positions of the point machines are checked. They should be the same as the required positions defined in the route layout. Then, the sequence of position detectors' occupations and releases is tested. For a successful passage, all the detectors in the route layout must be sequentially occupied and freed, as if the tram has passed through them. The fail conditions of these tests include the route status switching to ERROR. In the case of two simultaneous tram passages, a colliding route must not be created and built, too.

Sidebar: CSV file format

⌜ CSV (Comma Separated Values) is a simple file format for storing tabular data. It consists of a series of rows, with each row representing a different record, and each record consisting of a series of fields separated by commas.

The simplicity of the CSV format makes it a popular choice for storing and exchanging data, as it can be read and written by a wide variety of programs and is easy to understand. It is particularly well-suited for data that is organized into a table, with rows representing records and columns representing fields.

One of the key features of CSV is that it is a plain text format, which means that it can be viewed and edited using a simple text editor, which makes it easy to work with. However, it does have some limitations, such as the inability to represent more complex data structures or to include formatting information. ⌟

The generation of scenario files is achieved through the implementation of a set of Python scripts. In order to function properly, these scripts require data from the `SW definition table` and the `sw_names` table, which contains additional data about the system elements. In addition to this, project-specific information such as the `project_name` and `zone_name`, file paths, and various other configurations are stored in a separate configuration file called `task_config.yml`. Both `sw_names` and `task_config.yml` are also utilized in the production of the control program.

Despite being designed with a high level of formatting and usability, as evidenced by the inclusion of extensive comments and usage instructions on the company's internal wiki, it is possible that these scripts may not possess the necessary versatility to generate a wide range of scenarios. This potential lack of versatility could restrict their effectiveness in certain contexts.

config files Configuration files include a testing configuration file as well as a file with the element OPC paths and a file with the aliases for different status bits of specific elements. While the latter file is the same for software testing and implementation and wiring testing, the testing configuration file and the paths file are generated twice for each kind of testing.

paths.csv file is a table of OPC UA addresses of the system elements, organized into rows and columns. Each row represents a different system element and information about its paths. The first column Module contains the type of element module (such as Gate or Detector.TrackCircuit), the second column Name carries the element's name (such as 2G01 or 2TC01). The third column Path-Test and the fourth column Path-Control contain paths to the element in the test and control environments, respectively. The path files are created alongside the scenario files. They are mostly script-generated. However, since the OPC UA addresses of the elements tend to be the same across different projects, the automation of their generation is a reasonable choice. Additionally, due to their structure, these files can be easily modified as needed.

statusAliases.csv file is a table of element status variables and indexes of the corresponding bits in the status. Its first column contains the type of the module to which the status belongs, the second column holds a descriptive alias for a variable, the third column carries information about specific bits that represent the variable in the binary status dword, and the fourth column contains the value that means that the alias variable is set to True. For example, the line Detector.TrackCircuit.status;free;0;0 means that a detector of type Detector.TrackCircuit is free if the most-left bit with index 0 of the detector's status is 0.

configure.yml file is a configuration file. It contains several parameters that can be used to customize the behavior of the testing software. The first section of the file defines the parameters for a test PLC. This includes the URL of the PLC, a username and password to access it, and a zone name. The second section defines similar parameters for a control PLC. The File and Folder section specifies the locations of various files used by the testing software, such as a file containing test scenarios and a file with status aliases. It also specifies a folder where the results of the tests should be saved. The Configuration section contains several parameters that can be used to customize the behavior of the testing software. For example, the `max_time` parameter specifies a default timeout in seconds for a simple one-line test, and the `pause_between_rows` parameter specifies the amount of time in seconds to pause between each row of a test scenario. Other parameters in this section include options for saving console output to a text file, specifying which test case to run, and waiting for user confirmation before starting the tests. Overall, `configure.yml` file is well-structured and provides comprehensive control of the testing program parameters.

Sidebar: YAML file format

⌜ YAML (YAML Ain't Markup Language) is a human-readable data serialization language that is often used for configuration files. It is designed to be easy to read and write and is particularly well-suited for storing data in a hierarchical format. In YAML, data is organized into a series of key-value pairs, with each level of indentation representing a deeper level in the hierarchy.

YAML is similar to other data serialization languages such as JSON, XML, and TOML. Like YAML, these languages are designed to make it easy to store and exchange structured data. However, each of these languages has its own unique features and trade-offs. JSON is lightweight and easy to parse, and XML is more flexible, but both of them are also more verbose and thus can be more difficult to read and write. TOML is simple and easy to understand but supports a limited set of data types. Also, its support has been recently added to Python's standard library. With all that said, YAML is a good choice for a config file, especially for smaller projects. ⌟

The main disadvantage of the test files structure described above is its complexity and maintainability. Due to the fact that the testing scenarios files are generated by a script, the testing process lacks flexibility. Also, the paths files are only compatible with the glsopcua communication protocol, compromising the usability of the testing program in the future.

4.3.3 `pytester` module

`pytester` module is the core of the current automated testing process. It is a Python program that contains a set of functions that interact with each other with the PLC server in order to perform the testing and the `OpcuaClient` class and an OPC subscription handler class, that manage to communication process with the PLCs.

Sidebar: Python modules, packages, and libraries

⌜ A *module* is a single Python file (a file with the `.py` extension) that contains code, including definitions of functions, classes, and variables.

A *package* is a directory containing Python modules. A package may contain subpackages, which are packages within packages. Packages allow for a logical organization of modules and provide a namespace for the definitions contained within them.

A *library* is a collection of packages and modules that contain code designed for a specific purpose, such as scientific computing, data analysis, and more. Technically, the library is similar to the package. However, it is often assumed that while a package is a collection of modules, a library is a collection of packages. The Python Standard Library is a library of packages and modules included in the Python distribution.

Modules, packages, and libraries can be imported into other Python code, allowing the definitions and statements contained in them to be used in the importing program. This allows developers to reuse and organize code, and to leverage the work of others in order to build and maintain complex software systems more efficiently. ⌟

`OpcuaClient` and `Handler` Server-client OPC UA communication is built by using the `opcua` package. It is an open-source third-party pure Python synchronous implementation of the OPC UA specification. It allows for the creation of OPC UA clients and servers in Python, as well as the ability to connect to existing OPC UA servers and read or write data, and assures secure communication via encryption and authentication.

The OPC client of `pytester` is represented by the `OpcuaClient` class, a subclass of the `Client` class from the `opcua` package. It is used for managing the connection between the testing program and the glsplc glsopcua server. An instance of `OpcuaClient` is initialized with three arguments: URL, user, and password. URL is the PLC's TCP/IP address, and the user and password are used as credentials for logging in. It is worth noting that there is another class `OpcuaClient_async` with similar functionality, but it does not appear to be used later.

Client-server communication is provided through several `OpcuaClient` methods. The `connect` and `disconnect` methods establish and terminate the connection, respectively. The `writeValue` method writes a value to an OPC UA node specified by its node id. It first determines the data type of the node and then converts the value argument to the appropriate data type. It then writes the value to the node using the `Node` class's `set_value` method. The `readValue` method reads the value at an OPC UA node specified by its node id. The `createSubscription` method is used to create a new subscription on the server. A subscription represents a set of monitored items that the client is interested in receiving notifications for.

The `createSubscription` method expects a handler object as one of the input arguments. This object must implement a `datachange_notification` method, which is used to specify the action that should be taken when a notification is received from the server for any of the monitored items. It is called with a list of `DataValue` objects, one for each monitored item that has changed. The `Handler` class is the implementation of the `pytester` module to provide the handler objects.

Sidebar: OPC UA node subscriptions and data change notifications

⌜ In the OPC Unified Architecture protocol, a subscription is a mechanism that allows a client to subscribe to data change events from a server node. When a client subscribes to a server node, it receives updates for all data changes on that node, such as value changes and events [6].

When a client creates a subscription, the server assigns a unique identifier called a "subscription handle" to the subscription. This handle is used by the client to identify and manage the subscription. The client can use the subscription handle to add or remove monitored items, modify the subscription parameters, or delete the subscription.

When the client wants to subscribe to a node, it sends a request to the server to create a subscription, passing in parameters such as the publishing interval, the lifetime count, and the maximum keep-alive count. The server then creates the subscription, assigns a subscription handle, and returns it to the client.

Once the client has a subscription handle, it can then use it to add or remove monitored items to the subscription. A monitored item is a specific node or attribute of a node that the client wants to subscribe to. Each monitored item also has its own unique handle, called the monitored item handle, which the client can use to manage the monitored item. ⌟

Overall, `OpcuaClient` appears to be well-organized and easy to read, with clear method names and concise code. However, the `writeValue` method could be made more readable by using a dictionary to map `VariantType` values to Python data types, rather than using a series of if statements (if the prior data conversion is necessary in the first place). There are also several methods that are commented out or are not used. It might be better to remove them to avoid confusion.

As for the `Handler` class, there are a few issues with the current implementation that could be improved upon. First, a class instance is initialized with the `value` and `sub` parameters, but the latter does not seem to be used in the class. Second, the `datachange_notification` method has a try-except block that catches all exceptions but does not do anything with the exception other than print a message. This is not very helpful for debugging and does not provide any information about what went wrong. It would be better to include specific handling for specific exceptions, that may occur, or to re-raise the exception so that it can be caught and handled elsewhere. Finally, the `datachange_notification` method is using a global variable `act_values` to store the current values of the monitored nodes. This may not be the best practice, as it can lead to unexpected behavior if multiple instances of the `Handler` class are used simultaneously. Instead, it would be better to store the values in an instance variable of the `Handler` class, so that each instance has its own separate set of values.

For both `OpcuaClient` and `Handler`, it would be helpful to have more thorough error handling and logging throughout the code. The methods either only directly log an occurred exception or do not log at all.

`pytester` `pytester` is a Python program that manages the testing process.

Unfortunately, a detailed review of the `pytester` code exceeds the scope of this text. Upon examining the program, it is clear that it is quite lengthy and complex. Its procedural nature and lack of proper structuring and commenting make it difficult to understand and analyze. The code is stored within a single Python file and does not follow the PEP 8 style guide. It implements numerous functions that are tightly bound to each other. However, some functions seem to not be used. Furthermore, the absence of technical code implementation documentation only compounds these difficulties.

It is worth explaining, however, the overall working process of the program. A procedural approach has been taken for its creation, meaning that the program is represented as a sequence of steps or procedures that are to be followed in a particular order. First, it loads the configuration files and the scenarios files. Then, it establishes the server connection with the PLC and creates subscriptions for the system elements. After that, the testing begins. The program runs the test scenarios sequentially, line by line. It applies the testing commands and evaluates the test results. The results are stored and used for the crating of the test report files. These files are essentially extended versions of the test scenarios files with additional columns indicating the results and some additional data, such as the time it took to perform the test.

The testing process can be automatically triggered in a CI/CD pipeline by using a specific commit message tag. A dedicated script starts the testing process and uploads the results to a cloud server, providing the development team with easy access to the results for tracking and analysis of the software's performance. Additionally, the use of a cloud server enables remote access to the test results, allowing teams working remotely to stay up-to-date with the progress of the testing process.

It is important to note that the testing performed by the program is executed in a *synchronous* manner, with only one line of a testing scenario able to run at a time. While this may be the default behavior, it is not always necessary and some scenarios may be able to run concurrently without compromising the credibility or reliability of the testing process. On the other hand, synchronous code has one major drawback in comparison with *asynchronous* code, particularly in the context of network applications: speed. The time required to make a network request can be a significant portion of the overall time required to perform a test scenario, and this time is not utilized to execute other testing scenarios while waiting for a server response. As a result, the sequential approach can significantly reduce the speed of testing.

4.4 Conclusion

Based on the findings, it can be stated that the current testing program is able to perform the testing in accordance with the relevant guidelines and requirements. However, there are multiple areas where its realization or performance is suboptimal.

Starting with the elements' digital twins library, its implementation fits the testing purposes and is sufficient. The code is consistent and is written in a way that complements the control system perfectly. It does not require any modification.

The testing process allows for performing different testing sequences that in general assess the functionality of the system quite well. However, due to the fact that the generation of the scenarios files is script-based, it may be difficult to implement new testing scenarios, since it would require writing a new generating script. Moreover, the tram passage tests are only limited to a single tram passage case and a two trams passage case, while simultaneous passages of three or more trams are not supported. Additionally, the passage testing sequence does not check the correct function of the signaling lamps, which may be considered a downside, too. Overall, the testing program provides limited functionality for the testing. In terms of performance, the current solution is suboptimal, too, mainly due to its synchronous implementation. It could benefit from the asynchronous approach.

The files involved in the testing process, such as scenarios files, paths files, and the status aliases file, are well-structured and easy to read and modify. However, with a different approach to the program development process, it would be possible to reduce the number of these files by embedding the data that is not supposed to be changed often directly into the code. Also, maintenance of the scripts that generate these files may be problematic with time.

To conclude, the time and resources required to effectively modify the program may be substantial, especially given its current state. Unfortunately, the maintainability, scalability, and flexibility of the current software are rather poor. As such, it seems to be more efficient to develop a new program that incorporates best practices and efficient implementation from the outset.

5 New Testing Program Implementation

During the development of the new program, several key objectives have been considered in order to enhance the functionality and usability of the testing process These goals have been inspired by the software testing practices described in [**myers2011art**].

The primary goal is to increase the flexibility of the testing process by providing the user with comprehensive testing scenario creation and customization tools. This would allow testing the correct function of the system hardware, described in Chapter 2, as well as verifying specific control algorithms, described in Chapter 3.

Additionally, it is important to maintain a balance between the flexibility of the program and its ease of use. The goal is not only to increase the range of custom testing scenarios, but also to provide a clear program interface for their generation, reduce the number of test files, and improve the overall user experience.

Another objective is to increase the speed of the testing process by implementing asynchronous programming. This approach would enable the program to perform multiple tasks simultaneously, which can greatly improve the testing process efficiency. Additionally, it allows for the seamless initiation of parallel tests.

Lastly, the aim of the new implementation is to enable the support of other communication protocols, namely the company's SHV RPC, that may replace the OPC UA protocol in the future, and ensure the long-term maintainability of the program. This would allow the program to adapt to changing technological requirements.

5.1 Genaral Development Principles

To meet the goals of the project, the new program has been designed using several principles. First, the program code has been developed in the object-oriented programming paradigm and has been split into separate modules with several interface-like classes to ensure compatibility. Second, the asynchronous nature of the code has been implemented through the Python `asyncio` and `asyncua` libraries. Lastly, numerous programming patterns, such as the observer pattern or the factory pattern, as well as Python decorators, have been implemented to improve the structure of the program. These patterns are discussed in the next sections.

5.1.1 Reconsidering the Testing Files

The previous chapters highlighted the utilization of the `SW definition` table as the primary source of information for the testing program. This table is parsed by various scripts to generate the testing files, such as configuration files and testing scenarios.

The new approach introduces a unified intermediary data file, `configZones.json`, which serves as a bridge between the `SW definition` table and the testing program. This file is used for the `zone` object generation, which contains data about all the system elements, including their names, relations with other elements, and server addresses. Specifics of the `zone` creation and usage, as well as the generation of the test scenarios, are discussed in the following sections.

The JSON format is a widely adopted data interchange format due to its human-readable nature, lightweight structure, and ease of parsing. Additionally, it is closely supported by Python through the built-in `json` library [8]. The utilization of a single JSON file as the intermediary brings several advantages to the testing program.

First, the adoption of a unified data file eliminates the need for the program and its associated scripts to support different file formats and file structures. The reduction of the number of testing files decreases the potential for errors and simplifies the program structure, leading to better maintainability and sustainability of the code. Additionally, `configZones` file can be utilized by other programs, such as the code generation program, thereby promoting code reusability. Second, `configZones` file can be used for precise tracking of changes within the `SW definition` table. This improves the traceability of any issues that may arise due to uncoordinated table updates.

`configZones` file is created using `swdef_table_reader`, a module specifically designed to parse `SW definition` table. The table as a whole is represented by the `SWDefTable` class, while its separate sheets are represented by the nested class `Sheet`. It is worth noting that each sheet of the `SW definition` table represents a separate zone. `SWDefTable` class supports parsing of both local tables and remote tables, stored on Google Drive. `Sheet` class introduces the `get_cell_value` method for easy table cell data access, as well as methods allowing to represent the table data as a Python dictionary or write it to a specified JSON file through the utilization of the custom `SWDefSheetToDictConverter` class.

To maintain consistency in file formats, the program configuration file and the test results file have been redesigned as JSON files, too. The number of parameters in the updated configuration file has been reduced to the URLs, usernames, and passwords of the PLCs and the names of the control and test zones, which simplifies its management.

The new test results file consists of three main parts: the structure of the test, its description, and the messages logged during the test evaluation. Each section consists of the key-data structure, where the key is the unique id number of a test. The `"details"` section contains information about each individual test, including its name and description, as well as details about the evaluation process: the action being taken, the monitored value and its address, the expected and error values, and evaluation parameters such as timeout and retry settings. The `"structure"` section contains information about the organization of the tests, such as which tests are part of a larger testing scenario and the order in which they are run. The `"log"` section contains a record of events that occurred during the testing, such as the start of a test and the checking of values, with the keys for the events being timestamps corresponding to when they occurred. The structure of the file is illustrated below.

```
{
    "details": {
        "2361002583952 (main)": {...}, # the test scenario
        "2361002735824": {...}, # subtest #1
        "2361002097808": {...}, # subtest #2
        "2361002091792": {...}, # subtest #3
    },
    "structure": {
        "2361002583952 (main)": {
            ...
            "tests": [2361002735824, 2361002097808, 2361002091792]
        },
        "2361002735824": {...}
        "2361002097808": {...}
        "2361002091792": {...}
    },
    "log": {
        "2361002583952 (main)": {
            "1674215728.070 (0)": {...} # test scenario started
            "1674215728.681 (1)": {...} # test scenario passed
        },
        "2361002735824": {
            "1674215728.072 (0)": {...] # test started running
            "1674215728.210 (1)": {...} # datachange event
            "1674215728.210 (2)": {...} # value check (did not match)
            "1674215728.319 (3)": {...} # another datachange event
            "1674215728.383 (4)": {...} # value check (matched)
            "1674215728.383 (5)": {...} # test passed
        },
        "2361002097808": {...},
        "2361002091792": {...},
    }
}
```

5.1.2 Modular Design

Modular design is a software design technique that aims to divide a software system into smaller, independent, and interchangeable parts called modules. Each module has specific functionality and can be developed, tested, and maintained separately. The goal of modular design is to achieve a high level of cohesion within each module and a low level of coupling between the modules.

The new program contains several Python packages: the `elements` package, which implements the system elements, including their server addresses generation, node functionality, and element-specific data and logic, the `tests` package, which implements the testing logic and tools, the `client` package, which is responsible for the server-client communication and handles the event-based actions, and several smaller modules, such as the `swdef_table_reader` module. Each package contains a number of modules.

The modular design has several benefits and makes the codebase more readable. Modules can be reused in multiple parts of the program, reducing the need to write and maintain duplicate code. They can be also added, removed, or replaced with other modules, allowing the program to adapt to changing requirements. They can be easily updated or extended with new features individually, too, without affecting other parts of the program, making it easier to maintain and update the program as a whole.

To fully implement the modularity of the program code, the object-oriented programming (OOP) approach has been chosen as the foundation of the program architecture. Each of the above-mentioned packages implements an abstract class or a protocol class, which shapes the functionality of every specific program class. For example, the `Element` class is a base class for every system element class, and the `Test` class is a base class for every testing scenario.

Sidebar: Object-Oriented Programming approach and its benefits

⌜ Object-oriented programming (OOP) is a widely used programming paradigm that emphasizes the use of objects and their interactions in the application design. At its core, OOP is based on the concept of an object, which is the collection of data with associated behaviors [14]. Such an approach brings several benefits to the code of a program:

- *Encapsulation* of data and behavior within objects, or *classes*, makes the code more modular and easier to maintain. Since the implementation details of an object are hidden from the outside code, and only its interface is exposed, code changes in one part of the program are less likely to affect other parts. Also, this makes the codebase more readable and easier to understand [15].
- *Inheritance* allows creating new classes that inherit the properties and methods of existing classes. This can greatly reduce the amount of code that needs to be written and maintained, thus improving development efficiency.
- *Composition* allows objects to be composed of other objects. Composition is a way to reuse code by creating new objects that have a "has-a" relationship with other objects, which provides additional flexibility and makes code testing easier.
- *Polymorphism* enables a single method or interface to operate on multiple types of objects. In other words, a single method call can be used to perform different actions depending on the type of object it is called on, which allows for more flexibility in the code. ⌟

5.1.3 Concurrent Programming

Concurrent programming design is used to handle multiple tasks and operations simultaneously, allowing for improved performance and more efficient use of resources. There are several concurrent programming libraries in Python, including `threading`, `multiprocessing`, and `asyncio`. The last one provides the best speed up for programs that perform a lot of I/O-bound tasks [3] and thus has been chosen for the new program implementation.

Sidebar: Python `asyncio` library

⌜ The Python `asyncio` library is a built-in library that offers an asynchronous execution model. It allows developers to write concurrent code using the `async/await` syntax and provides a variety of tools and mechanisms to handle concurrency, such as event loops, coroutines, and tasks. It is worth noting that `asyncio` library does not allow to implement true parallelism, but rather uses a single-threaded, event-driven, and non-blocking model to switch between tasks as they become available [7].

Event loop is the central component of the `asyncio` library. It runs in the background and manages the execution of coroutines, scheduling them and handling the communication between them. The event loop can be thought of as the "brain" of the asynchronous execution model, managing the flow of control and ensuring that all tasks are executed in the correct order.

Coroutines are a special type of function that can be paused and resumed, allowing other tasks to run in the meantime. They are used to represent tasks that can run concurrently, such as server requests, database queries, and other IO-bound operations. Such functions, also called *asynchronous functions*, are defined with the `async` keyword. *Tasks* are a higher-level abstraction built on top of coroutines; they may be used to schedule coroutines to run on the event loop concurrently with other tasks. Both coroutines and tasks can be *awaited*. The `await` keyword is used inside an asynchronous function to "pause" its execution and wait for an asynchronous operation to complete. Once the operation is complete, the function continues its execution from the point where it has been paused. For example, while one server request is waiting for a response, other operations can continue to execute.

The code example below showcases the `asyncio` library usage and the performance improvements it may bring to the program. It imitates the testing process of a detector occupation function (the procedural implementation does not reflect the actual testing program code and serves an illustration purpose).

```
async def occupy_detector(detector): # async keyword is used to define an
    asynchronous function (a coroutine)
    await detector.test.write_value("set_true", True) # await keyword enables
        to call other asynchronous functions while the expression is waiting
        for its result

async def get_status(element):
    return await element.control.read_value("status") # await expressions can
        be treated the same as regular expressions; for example, here the
        value that has been awaited is returned by the function

def is_free(detector_status: int) -> bool:
    """a regular synchronous function to evaluate if the detector is free
        based on the passed status value"""
    if status == 0:
        return True
    return False

async def test(detector, allow_fail: bool = False):
    await occupy_detector(detector) # await for the occupy_detector()
        execution
    status = await get_status(detector) # await for the get_status()
        execution and store the status variable
    if is_free(status): # evaluate the status
        print(f"occupation test of {detector}: fail")
        if allow_fail is True: # if the allow_fail argument is False, raise
            TestException, which will terminate the tests of other detectors
            raise TestException(f"Test of {detector} failed")
    else:
        print(f"occupation test of {detector}: success")
```

```
async def main():
    detectors = [Detector(), Detector()] # genetare a list of detectors
    try:
        with asyncio.TaskGroup() as tg: # create a task group that runs the
            asynchronous tasks concurrently
            for detector in detectors:
                test = test(detector, allow_fail=True) #initialize a test
                task = asyncio.wait_for(test, timeout=5) # create an
                    asynchronous task with a timeout
                tg.create_task() # add the task to the task group
        print("Finished")
    except* TestException: # if an exception is raised in any task of the
        task group, all the running tasks get canceled anyway.
        print("Aborted due to a TestException raise")
    except* asyncio.TimeoutError:
        print("Aborted due to timeout occurred")

if __name__ == "__main__":
    asyncio.run(main())
```

In this example, `test` is an asynchronous function that calls `occupy_detector` and `get_status` functions (that are also asynchronous) and checks the status of a detector using the `is_free` function. If the detector is not free, the function reports about the test fail, and, if the `allow_fail` parameter is set to `False`, raises a `TestException`. Otherwise, if the detector is occupied, it reports a successful test. Each `await` keyword in these functions suspends the execution of the surrounding coroutine (in other words, of this function) until the result of the awaited action is returned and passes function control back to the event loop. The `main` function generates a list of detector objects and creates test tasks using the `asyncio.TaskGroup` context manager. Each task is calling the `test` function for a generated detector and has a timeout of 5 seconds. If any of the functions raise an exception, for example, a `TestException` or a `asyncio.TimeoutError`, it can be handled with a corresponding exception group. The concept of exception groups is described in detail in [9]. However, even with exception handling, the execution of the rest of the running tasks is canceled.

According to the code flow, the testing sequence for a detector is the following:

1. Occupy the detector: make a server write request for the value `True` to be written at the remote test node `set_true` of the detector.
2. Read the detector status: make a server read request of the value at the remote control node `status` of the detector.
3. Evaluate the received value locally: if the detector has not switched to the occupied state, and the `allow_fail` parameter is set to `False`, raise an exception; exit the function otherwise.

The first two steps of the test evaluation implement client-server communication, which, by their nature, introduces a period of time when the program is waiting for an external network request to complete. If the detectors were tested sequentially, the program would loop through each test in a synchronous manner and would not proceed to the next test until the previous one has been completed.

Instead of this, the program is testing multiple detectors simultaneously, running the tasks concurrently, so while one task is waiting for the detector to be occupied, another task can be testing another detector. Such workflow can be seen as parallel execution of server requests. As a result, the program handles multiple requests at once, thus greatly increasing its performance and speed. A similar testing scenario is evaluated and discussed in sec. 6.1.1. ⌟

5.2 `elements` package

The `elements` package contains several modules that implement the system elements' functionality, including their hierarchy in the control and test systems, server node references, and communication addresses. The central module of the package is the `element` module, which contains base classes and protocols that represent a general system element. Other modules contain classes that imitate specific elements, such as detectors, point machines, gates, routes, signals, and signal symbols. These classes are inherited from the `Element` base class. The package also contains the `zone_creator` module with utility classes for convenient automated `zone` generation and configuration.

5.2.1 Element class

The Element base class of the element module is a general representation of a single system element. It has several attributes, including the element name, its varname (an attribute derived from the name, used in the PLC program), the module, which represents the element type, an optional parent element, and an optional list of dependent child elements. Each element also has a control node attribute and a test node attribute. If the element is not represented in the test library, its test node is set to None.

The name and the varname attributes, as well as some other attributes, are updated via special setter functions. Since these attributes take part in the element's node address creation, setters also update the address of the control and test node objects.

In addition to the Element class, the package contains several modules with its child classes: Zone, Crossing, Detector, PointMachine, Gate, Route, Signal, SignalSymbol, and SignalSymbolMeasured. These classes define some of the Element initialization attributes, such as module, and also introduce new, element-specific attributes. They also specify the ElementNode classes or subclasses for the initialization of element nodes and instantiate element server address creators via the ElementAddressFactory class, eliminating the need for the paths files.

Each element class includes a nested enumerator class used for the server values masking. Each enum contains a set of symbolic names and associated integer masks. Every instance of a specific element class has access to the enums of this class. This way of parsing the dword status is more intuitive and replaces the statusAliases file. Below there is an example of the detector's ControlStatusBitMask class.

```python
class ControlStatusBitMask(enum.Enum):
    OCCUPIED = 1 << 0 # 1
    CLAIMED = 1 << 1 # 2
    ERROR = 1 << 29 # 536870912

# mask all bits except the first lower bit
control_state = 0b10101010101010101010101010101
masked_state = control_state & ControlStatusBitMask.OCCUPIED
print(bin(masked_state)) # prints '0b1'
```

In this example, the & operator is used to perform bitwise masking on an integer value. We use the values of the ControlStatusBitMask enum as masks to select specific bits in the integer value. To mask the first lower bit of control_state, we use the value of ControlStatusBitMask.OCCUPIED as the mask. When we perform the bitwise & operation, we preserve the value of the lower first bit of control_state while setting the rest of the bits to zero.

It is worth taking a closer look at the Zone class. Upon initialization, the class is provided with several dictionaries that contain the element names as keys and the corresponding element objects as values. Once the Zone object is created, any element of the system can be easily accessed through the elements property, which returns a flat dictionary containing all the system elements. Here is an example of the detectors dictionary stored in the zone instance:

```
print(zone.detectors) # prints {'2TC01': detector_tc 2TC01, '2TC02':
    detector_tc 2TC02, ..., '2TC18': detector_tc 2TC18}
print(zone.detectors['2TC01']) # prints "detector_tc 2TC01"
print(zone.detectors['2TC19']) # raises a KeyError
print(zone.detectors.get('2TC01')) # prints "detector_tc 2TC01"
print(zone.detectors.get('2TC19', None)) # prints "None"
```

In this example, the detectors attribute of the zone contains a dictionary that provides a mapping of the detector names to the detector element instances. The example also illustrates the querying of the dictionary contents, both directly with the key and via the get method, which returns a defined default value rather than raising a KeyError if the key is not present in the dictionary.

To further facilitate the management of the elements, the Zone class includes two methods, attach_control_client and attach_test_client, which enable the simultaneous attachment of a client for server communication to all elements within the zone.

Although it is possible to create every system element and instantiate the zone manually, it would be much more convenient to automate this process. This could be accomplished with help of the ZoneCreator class.

ZoneCreator takes the configZones file as input and is able to generate system elements for a specific zone. The elements are organized into dictionaries for easy access. It is possible to provide an external address file as an alternative source for the server addresses of the elements, rather than using the addresses generated by the element address factories. Via the create_zone method, a complete zone creation with an initialized element structure can be achieved in one call.

An important detail is that when the dictionaries containing the elements are created by ZoneCreator, the elements mostly hold a string name reference to other elements that are connected with them rather than their actual instances. The bind method of the ZoneCreator class serves the purpose of binding the elements together by replacing the names of the connected elements with the actual element instances. This allows for easy access to the connected elements without having to look them up by name every time they are needed. From the technical realization point of view, this method takes in any number of dictionaries of elements as arguments and combines them into a single dictionary. Then, it iterates over all attributes of each element, gets the attribute value, and if this value is a string, it tries to look up the connected element in the dictionary. If it is found, it sets the attribute to the connected element instance. If the attribute value is a list, tuple, or set, it does the same thing but for each element.

As an example, consider the case of gates and SG lamps. A Gate class instance is initialized with a parameter that remarks the assigned SG lamp, which, however, may not have been initialized yet, so the Gate constructor is only given the lamp string name. Once the Signal instance is created, bind method may be used to bind the gate and its corresponding lamp together by replacing the lamp name string with the actual Signal instance. This allows for easy access to the lamp without having to look it up by name every time it is needed.

Another element class worth mentioning is the Route object. The specialty of this class is the data structure called layout. It is a list of the Element objects - point machines, position detectors, and crossings - that together form the route arrangement. Each object of this arrangement is called a route element and is wrapped by the RouteElement class. A RouteElement object stores the base element in the core attribute, and it can thus access its attributes, and holds a reference to the route it is a part of and its position coordinates in the route layout. RouteDetector, RoutePointmachine, and RouteCrossing classes are the subclasses of the RouteElement class. These classes specify additional attributes, for instance, the required position of a point machine for the route build.

5.2.2 ElementNode class

The element nodes are dedicated objects that are a part of the Element object composition. While the Element itself is more of a data container, the ElementNode object implements its functionality in a form of subscription, validation, read, and write commands. Element nodes are meant to resemble remote OPC UA nodes stored in the PLC. Each node is a part of a node tree structure, and each node can be a parent node to a number of other nodes.

The ElementNode class takes a client attribute, which is an instance of a communication client class that follows ClientProtocol, and an address attribute, which is an instance of an address class that follows the ElementCommAddress protocol. The client handles the communication process for the node and the address generates the node address depending on the used communication protocol. It is worth noting that the client attribute can be passed either to the class constructor during the initialization or via the attach_client method. The latter attaches a ClientProtocol client to the ElementNode instance at any time, allowing the nodes (and thus the elements) to be created and manipulated for other purposes than communication with the server. The ElementNode object also stores the mapping of the names of its child nodes to their server values in the child_nodes attribute. Initially, the values are set to None; they are updated with every data change event on the server (the concept of such events has been discussed in the sidebar of sec. 4.3.3).

ElementNode's methods subscribe_child, read_value, and write_value are the base methods for interacting with the node and its child nodes on the PLC server. These methods get the address of a given child node and then reuse the corresponding methods of the client. Such an element-based approach seems to be more intuitive and object-centered in comparison with the direct usage of the client's subscription, read, and write methods.

ElementNode is an abstract base class, and two other classes - ElementControlNode and ElementTestNode, that represent control nodes and test nodes, respectively - inherit from it. While the latter is technically the same as the base class and does not define any additional methods or attributes, ElementControlNode introduces a set of methods for the element status acquisition (given the fact that any element node in the control structure has a status child node). Similar to the Element class, besides the ElementControlNode and ElementTestNode classes, the package also contains element-specific child classes of the ElementNode class.

Some elements directly use ElementControlNode or ElementTestNode. For example, detectors and point machines use ElementControlNode for their control nodes. It is worth noting that the above-mentioned fact does not necessarily imply that the remote control nodes of such elements do not have any additional functionality; it rather means that this functionality is not required and thus is not implemented.

A good example of an extended node would be DetectorTestNode. While the control node of a detector remains unchanged, its test node defines methods set_true, set_false, and reset. Their names are rather self-explanatory. These methods call the write_value of ElementNode with the corresponding parameter name (or, in other words, the child node name) and True as the value that is written.

In the example below, a detector is set to the occupied state with the set_true method. This method calls the write_value method with parameters setTrue and True. The main advantage of these element-specific node methods is intuitiveness and ease and of use, since they do not require knowing the node name in the PLC server.

```
async def occupy_detector(detector: Detector):
    print(detector.control.status) # prints the current status value
    await detector.test.set_true() # apply the command to occupy
    print(detector.control.status) # prints the new status value (1)

if __name__ == "__main__":
    detector = zone.detectors,get("2TC01") # get the "detector_tc 2TC01"
        object
    asyncio.run(occupy_detector(detector))
```

5.2.3 `ElementCommAdderss` class

While the tree structure of the nodes and their commands do not depend on the communication protocol, the technical aspects of the communication and node address generation do. That is why the node remote address creation is separated from its functionality to a different class - `ElementCommAddress`. Each `ElementNode` instance takes the address generator in a form of a `ElementCommAddress`-like instance during its initialization.

The `ElementCommAddress` class itself is a protocol class. It defines three methods: `get_address`, `get_child_address`, and `set_address`. `get_address` returns the node address of the element, `get_child_address` returns the child node address for a given child name, and `set_address` allows the user to manually set the relative address for the element. Any object, which is meant to be used as a node address generator (for example, the `SHV` address generator class, which has not been implemented yet), should follow this protocol and provide the implementation of the above-mentioned methods in order to be natively compatible with the rest of the program.

The OPC UA implementation of the `ElementCommAddress` protocol is realized in the `ElementOPCAddress` class. The instances of this class are initialized with a relative OPC address of an element in the form of a string and generate a full OPC node address in a form of an `asyncua.NodeId` object. As prescribed by the protocol class, it has three methods: `get_address`, `get_child_address`, and `set_address`.

The `ElementAddressFactory` class is used to create the address generators in an easier and more automated manner. It is designed to be able to work for both OPC UA and SHV communication protocols, but since the `ElementSHVAddress` has not been implemented yet, its functionality is currently limited to `ElementOPCAddress` instances creation. The factory is initialized with four arguments that represent general templates of relative control and test addresses of an element node for both communication protocols. These templates contain placeholders, where a value is later inserted, for instance, via the in-build Python `format` function. Once initialized, the address factory can be "called" with a set of parameters to format the template address string and create a tuple of `ElementOPCAddress` instances, one for the control node and one for the test node (if the element has such). These two address generators may be then automatically assigned to the corresponding nodes of a specific element.

5.3 `tests` package

The `tests` package provides a comprehensive set of tools for the intuitive and flexible creation of test scenarios. The central component of the package is the base `Test` class and its various subclasses. Also, it includes a number of supporting classes such as `TestDescription`, `TestRunLogger` and the custom data structure `TestMemory`, as well as the `PassageGenerator` class, which is capable of generating complex testing scenarios for one or multiple simultaneous tram passages.

The `Test` class is an abstract base class, meaning that it is not intended to be used directly. It defines a set of methods that a subclass should implement in order to be considered a valid test, such as the `run` method or the `log`, `details`, and `structure` properties. Additionally, it ensures that every test instance has settable `status`, `parent`, and `description` attributes and initializes instances of the `TestRunLogger` and `TestMemory` classes for the test.

The `log` property should return a dictionary containing log information, such as the start and end times of the test, the expected and actual results, and any error messages that may have occurred during the test. This information can be used for debugging and analyzing test results.

The `details` property should return a dictionary containing detailed information about the test, such as its name, expected results, error values, evaluation details, and custom conditions. This information can be used to understand the configuration and behavior of the tests.

The `structure` property should return a dictionary containing information about the structure of the test, including its name, nest level, and child tests. This information is particularly useful when working with multitests, and can be used to understand the organization and hierarchy of the tests.

Sidebar: Python Abstract Base Class (ABC)

⌜ In Python, Abstract Base Classes (ABCs) provide a way to define a set of methods and properties that a concrete class, which inherits from the base class, must implement in order to be considered a valid implementation of this base class. These methods are typically decorated with the `@abstractmethod` decorator.

The use of ABCs allows for a clear separation of the interface (the methods and properties defined in the ABC) from the implementation (the specific implementation of those methods and properties in the concrete classes). This can make it easier to understand the structure of the code, as well as allow for more flexibility in terms of the implementation of the concrete classes. It's also worth noting that an ABC can also include implementations of certain methods, and the concrete classes don't have to implement those, but they have the ability to override it [14].

```python
class Test(ABC):
    ...
    @abstractmethod
    async def run(self):
        ...

class SingleTest(Test):
    ...
    async def run(self):
        async with asyncio.TaskGroup() as tg:
            tg.create_task(asyncio.wait_for(self._test(), timeout =
                self.timeout))
            tg.create_task(self._background_wait())

class SequentialMultiTest(Test):
    ...
    async def run(self):
        for test in self.tests:
            await test.run()

class SequentialMultiTest(Test):
    ...
    async def run(self):
        async with asyncio.TaskGroup() as tg:
            for test in self.tests:
                tg.create_task(test.run())
```

In this example, the abstract base class `Test` class defines an `@abstractmethod` named `run`, that the subclasses of Test, such as `SingleTest`, `SequentialMultiTest`, and `ParallelMultiTest`, are required to implement. Such design allows any combination of `SingleTest` and `MultiTest` objects to be used in a test scenario. ⌟

5.3.1 `SingleTest` class

The `SingleTest` class represents a simple system element test. It resembles a line of the `pytester` testing scenario but allows for a more flexible test design.

Initialization The test is created with an `Element` object, which allows accessing its data and node functionality (for example, making server read and write requests). It is also provided with the expected value, which indicates a successful test, and optionally an error value, which indicates a failed test. These values may be numbers, strings, lists of values, or specific objects, for example, an instance of a `RouteDetector`.

During the initialization, an instance of the `SingleTest` class also may receive several settings and evaluation details parameters, including:

- `require_datachange`: if set, the test waits for the data change notification from the tracked node before proceeding to the server value read and evaluation.
- `custom_tracked_node_address`: specifies the node address which is tracked by the test. By default, it tracks the status node of the assigned element.
- `allow_retry`: if set, allows the test to perform more value evaluations in case the first evaluation has failed.
- `mask`: if specified, this integer parameter masks the received server value. The masking is performed as a bitwise AND operation (as described in sec. 5.2.1).
- `check_mode`: specifies the comparison logic between the actual value and success and error values. Set to the equality check by default, it may also be set to determine if the value is less or equal, greater or equal, or is among a list of values. It is also possible to define a custom checker method.
- `assured_duration`: a minimum amount of time that the test runs for.
- `timeout`: a maximum amount of time that the test runs for.

It is also possible to provide the test with a `description` object, which includes a dictionary structure with explanations of the test purpose, values descriptions, etc. If it is not provided, the test constructor attempts to use a test description factory, gathering data from the code implementation and other sources, and creates the description itself.

Besides the `SingleTest` class, `testlib` module also contains a number of predefined test classes, that inherit from the base class. These child classes define some of the `SingleTest` initialization attributes to suit specific testing objectives, which may make them a more convenient choice for the user. Currently, the following tests are presented in the library: `ValidateAddress`, `ResetSystem`, `SetZoneToAB`, `SetZoneToNORM`, `RequestRoute`, `CheckIfRouteReady`, `CheckIfRouteOccupied`, `CheckIfRouteFree`, `CheckIfPrevRoutePassed`, `CheckIfGoSymbolShining`, `CheckIfPMEInPosition`, `ActivateDetector`, `DeactivateDetector`, and `CheckIfDetectorClaimed`.

Methods tagging with `CustomTags` The `SingleTest` class is designed to run a set of actions defined in the `command` method and then acquire and check it against *one* specific value. For example, during the `SetZoneToNORM` test, a `zone.control.cmd_set_norm()` command is applied, and the test checks if the `zone.control.get_status()` method returns a value of `1`, which signalizes that the `zone` is in the `NORM` state. However, sometimes additional evaluations are required to properly perform the test. Additionally, some actions should be performed only if specific conditions are met, for example, if the test has failed.

The `start_condition`, `custom_warning`, `custom_error`, and `custom_action` decorators are implemented to mark methods of a `SingleTest` class or its subclasses to be performed in those situations. These decorators add a special `tag` attribute that contains the event tag and a `desc` attribute that contains the description of the function. These tags allow for the test class to identify and categorize specific methods by their intended purpose based on the tag value. The tagged methods are collected during a `SingleTest` class initialization and are run depending on events; their results are used to determine the outcome of the test. It is worth noting that the methods marked as a custom condition or custom event must return `bool`, and custom action methods must return `None` (in other words, return nothing).

Sidebar: Python decorators

⌜ In Python, a decorator is a design pattern that allows adding functionality to a function or a class, thus modifying its behavior, without changing its code. It does this by wrapping the function or class in another function and returning the wrapped function or class. Decorators are often used to add logging, debugging, or input validation to functions or classes without changing their implementation.

```
class CheckIfPrevRoutePassed(SingleTest):
    """check if the route has begun setting (changed state to BUILD) after
        the specified detector has been released"""
        ...
    @start_condition(description="Route has switched to the BUILD state")
    async def route_in_build(self) -> bool:
        """route is in the BUILD state"""
            ...

    @custom_warning(description='Route set later than it should have been')
        async def route_set_late(self):
        """more detectors have been released before the route has switched to
            BUILD than necessary"""
            ...

    @custom_error(description="Route has switched to the ERROR state")
    async def route_in_error(self) -> bool:
        """route is in the ERROR state"""
            ...
```

In the example above, `CheckIfPrevRoutePassed` class uses the `start_condition` decorator to tag the `route_in_build` method as a start condition, which modifies the behavior of the test so that it only starts if `route_in_build` has returned `True`. Similarly, being tagged with the `custom_warning` and `custom_error` decorators, the `route_set_let` and `route_in_error` methods make the test register the warning condition or evaluate itself as failed, respectively, if they return `True`. ⌟

Evaluation The test starts running when its `run` method is called. It runs asynchronously, and its execution does not block the execution of the main program. Upon the call, the following sequence of actions occurs:

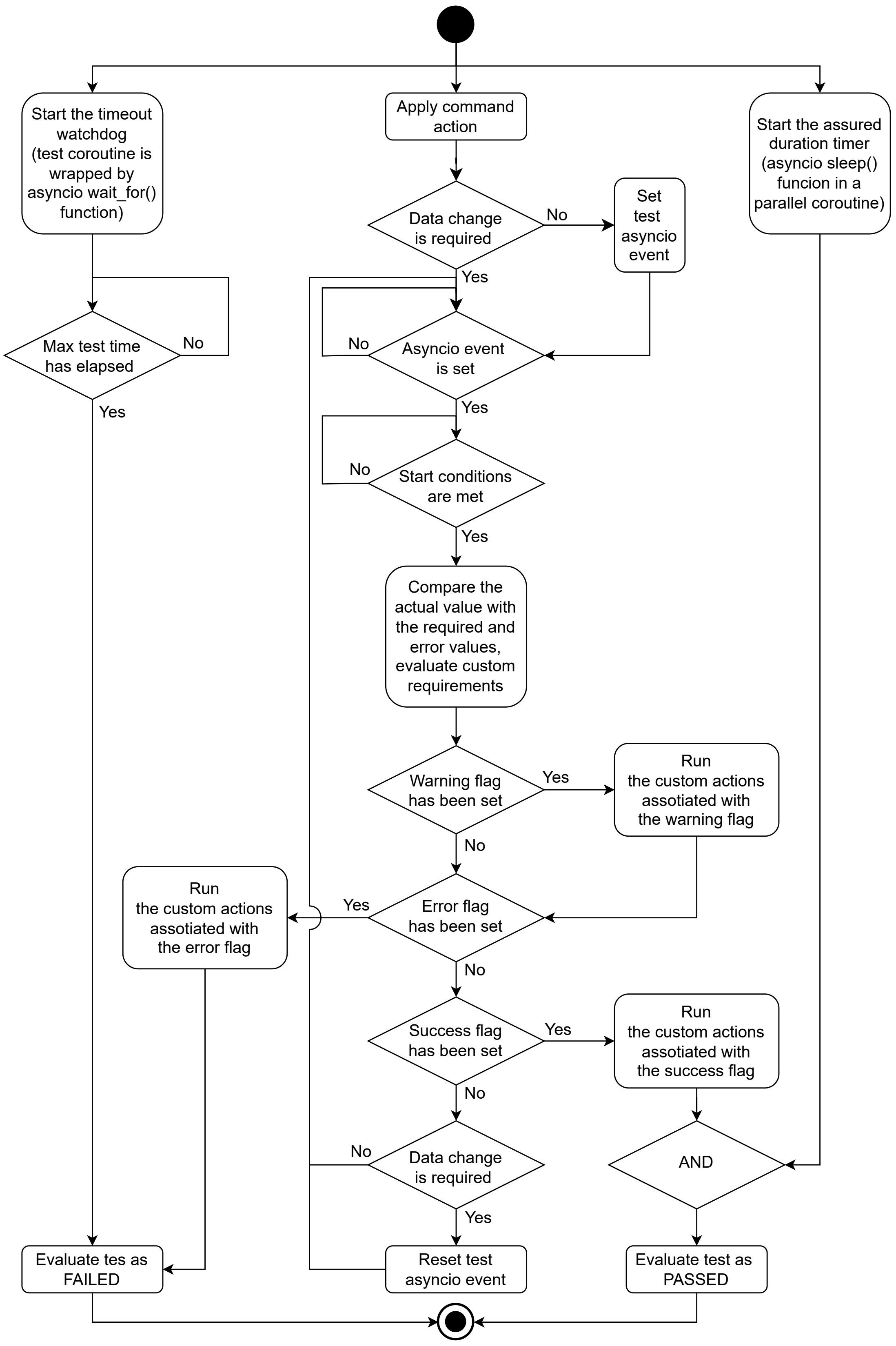
1. The test applies the `command` function. If the `require_datachange` flag is set to `True`, it then waits for the data change notification from the node address it is following; otherwise, it proceeds. The test also checks if the custom start conditions are met by evaluating all the functions that are decorated with a `start_condition` decorator. If any of the functions' returns is false, the test repeats the evaluation after a period of time.

2. The test acquires the monitored object value with the `get_value` method. For example, it may either do a server value request for a specified node address or read from the internal test memory. The value is then checked by a chosen value-checking method to match the specified conditions. The methods marked as custom conditions, if any, are evaluated, too.

3. If the current value of the monitored node matches the expected value or conditions, the test is considered a success, and the custom success actions are executed. If, on the contrary, the monitored node matches the error value, or a timeout occurs, the test is considered as failed, and the custom error actions are executed. Also, the test checks for custom warning conditions, if any, and executes the custom warning actions if the conditions are met.

The applied testing process is illustrated in detail in figure 16.

Each occurred event, may it be a test start, a value check, a custom action, or a test fail, is logged by the test logger. Each test has its own logger, which is an `TestRunLogger` object. Each log message must have a specified event type and an arbitrary number of key-data pairs. The logger also adds a timestamp to the message and stores it in a private dictionary structure.



Start the timeout watchdog (test coroutine is wrapped by asyncio wait_for() function)
Apply command action
Start the assured duration timer (asyncio sleep() funcion in a parallel coroutine)
Data change is required
No
Set test asyncio event
Yes
Max test time has elapsed
No
Yes
No
Asyncio event is set
Yes
No
Start conditions are met
Yes
Compare the actual value with the required and error values, evaluate custom requirements
Warning flag has been set
Yes
Run the custom actions assotiated with the warning flag
No
Run the custom actions assotiated with the error flag
Yes
Error flag has been set
No
Success flag has been set
Yes
Run the custom actions assotiated with the success flag
No
No
Data change is required
AND
Yes
Evaluate tes as FAILED
Reset test asyncio event
Evaluate test as PASSED

Figure 16: A `SingleTest` flowchart

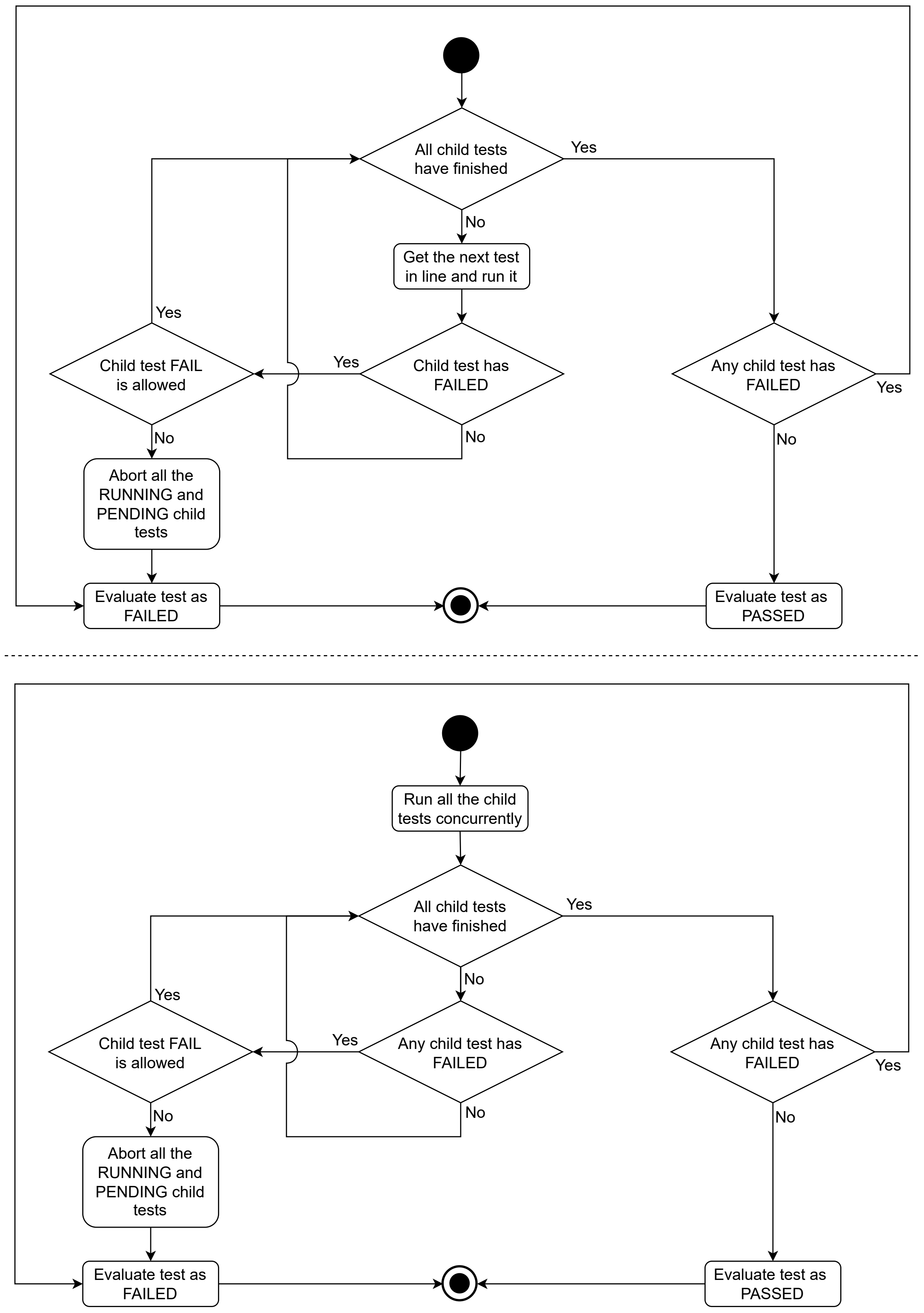
5.3.2 `MultiTest` class

The `MultiTest` class is a wrapper class for other tests, both the `SingleTest` objects and other `MultiTest` objects, that is able to run them all together. It organizes and manages multiple related tests and helps in structuring and performing complex test suites. Upon creation, the class constructor groups the tests, assigning them a parent reference and a common memory structure (the latter is described in section 5.3.4).

`log`, `details`, and `structure` properties of the `MultiTest` object not only return wrapper object data but also embed data from its child tests. This is achieved through the use of private recursive helper methods. These methods, as well as the method for the common memory structure setup, investigate the type of the child object. If the child object is also of type `MultiTest`, the same method is called on this object, and if the child object is of type `SingleTest`, the returned dictionary is updated with its data. The dictionary has a simple, flat structure, meaning all the keys are at the top level, without any nested dictionaries or lists, which makes accessing test data easier.

Another common parameter for all `MultiTest` objects is the `allow_fail` attribute, which determines the behavior of the wrapper if one of its child tests fails. If `allow_fail` is `True`, the test does not interrupt the other child tests in case of a failure. Such behavior is mainly intended for testing scenarios where the tests do not depend on each other, for instance, position detectors' occupation functional testing. On the other hand, if the child tests rely on the success of the previous tests in the testing scenario, for example, during the tram passage, there is no point in continuing the scenario if a failure occurs, and the tests that have not been evaluated yet are aborted.

`MultiTest` itself is an abstract base class; it provides common functionality for sequential and concurrent test classes, `SequentialMultiTest` and `ParallelMultiTest`, while the actual test run logic is implemented in the child classes. The logic depends on the type of testing process. In the case of `SequentialMultiTest`, the child tests are run one by one in the order they were passed; `SequentialMultiTest` runs all the child tests concurrently. The implementation of the `run` methods of these classes has been illustrated at the beginning of the chapter in the abstract base classes discussion. The testing process of both sequential and parallel testing scenarios is also illustrated in detail on figure 17.



All child tests have finished
Yes
No
Get the next test in line and run it
Yes
Child test FAIL is allowed
Yes
Child test has FAILED
Any child test has FAILED
Yes
No
No
No
Abort all the RUNNING and PENDING child tests
Evaluate test as FAILED
Evaluate test as PASSED
Run all the child tests concurrently
All child tests have finished
Yes
No
Yes
Child test FAIL is allowed
Yes
Any child test has FAILED
Any child test has FAILED
Yes
No
No
No
Abort all the RUNNING and PENDING child tests
Evaluate test as FAILED
Evaluate test as PASSED

Figure 17: A `SequentialMultiTest` (top) and `ParallelMultiTest` (bottom) flowcharts

5.3.3 `PassageGenerator` class

The `PassageGenerator` class is designed to create test scenarios for the passage of trams through a specific route or set of routes. Upon initialization, it assigns a tram for each route in the zone and creates route-tram pairs; it also provides an option to set a custom tram with modified parameters for a chosen route. The trams are instances of the `Tram` class, which represent a physical vehicle with speed, length, and position parameters traveling through the zone.

The class implements the `__call__` method, which is a special *dunder* method (the word "dunder" is short for "double underscore") that allows an object to be invoked as a function. Using this method, the generator can be run with route names as parameters to generate a test scenario for the provided route combination. Additionally, it is possible to specify the `truncate`, `fastmode`, and `optimal_routing` parameters to adjust the scenario generation process.

The `truncate` parameter determines the number of the position detectors that are to be occupied and released during the scenario. When set to `True`, the scenario terminates upon the occupation of the first detector in the layout. Conversely, if set to `False`, the scenario concludes with the release of the last route detector and a check to verify if the route has transitioned to the FREE state. In scenarios involving multiple routes, the test scenario is generated based on the route detectors that release the last point machine common among any two or more routes.

The `fastmode` parameter determines whether the test scenario should accurately reflect the time it takes for a tram to pass through a route. When set to `False`, the generator inserts time delays between each detector occupation and release, simulating the actual time it takes for a tram to travel between the detectors. Conversely, when set to `True`, the tests are applied immediately one after another, resulting in a faster but less accurate simulation.

The `optimal_routing` parameter determines the control routing algorithm used in the testing scenario. When the parameter is set to `True`, the detector, which must be released before the next route in the queue is set, is determined according to the routing algorithm described in sec. 3.5.5. Otherwise, if set to `False`, the tracked detector is the last detector of the current route.

The generator algorithm creates the passage test scenario in several steps.

1. make a route request for every route. The requests are made by the FIFO principle, so the first active route is the route that has sent the request first.
2. determine the next route in the queue to be set. The currently active routes and the waiting routes are stored in two lists. In a loop, perform the following calculation for every pair of an active route and a waiting route:
   (a) find the last layout element of the active route which is also present in the waiting route layout. If the `optimal_routing` flag is set, the algorithm finds the common element by iterating over the list of elements of the active route's layout in reverse order and checking if the element's name is also present in the waiting route's layout. If it finds a match, it then looks for the first position detector after this element and returns it. Conversely, if the routes do not have a common element, they are not colliding and can be built simultaneously.
   (b) calculate the time it would take for the tram of the active route to pass through the common element (i.e. time to the moment when the end of the tram leaves the detector area). After this time, all the layout elements of the waiting route are free, and the route may be passed though.
   (c) add a delay to the calculated time. The delay reflects the fact that the active route from the route pair may not be have been set simultaneously with the first route but after a period of time. In other words, even though the waiting route may have a small waiting time relative to this active route, the active route itself may have been built with a significant delay relative to the route that has been built before it. This delay propagates to the overall waiting time for the waiting route before it may be built.
   (d) sort the waiting routes and pick the one with the smallest cumulative waiting time. Add this route and its associated delay to the list of active routes.
   (e) perform the calculation loop until the list of waiting routes is empty.
3. cut the route elements that are not common with any other route layout from the passage scenario if the `truncate` flag is set.

The result of the generation is a `SequentialMultiTest` object that wraps the system reset test, sequential route passage request tests, the tests that check if every route has been set properly (the route detectors have been claimed, the point machines are in the correct position, the GO symbols are displayed on the SG lamps, etc.), the defined sequences of the detectors occupation and release tests, that imitate the actual tram passage, and the tests that verify that all the routes are set in the right time.

5.3.4 `TestMemory` class

There are cases when different tests need to share local data with each other. For instance, the key aspect of the multiple tram passages testing is following the released detectors during the first route passage and evaluating if the next routes have not started building sooner than a certain detector has been released. However, due to the design of the `SingleTest` class, acquiring such data from the server might be cumbersome. This leads to a need for introducing a shared data structure for the tests, which would provide both flexibility and a high degree of control over the stored data.

The `TestMemory` class is a shared memory structure designed to be used by multiple instances of all subclasses of the `Test` class, including `SingleTest`, `SequentialMultiTest`, and `ParallelMultiTest`. It provides a set of methods that allow for the creation, reading, writing, and clearing of keys and values in the memory store. The class is designed to be flexible and able to meet various demands for data storage.

One of the key features of the `TestMemory` class is the `create_key` method. This method allows the user to create a key-value structure in the memory store, with options for specifying the required data type of the value, storing previous values, ensuring that the values are unique, and protecting the value from being cleared. This enables a high degree of flexibility in terms of data storage needs.

A value associated with the given key can be accessed with the `read` method, and a given key can be assigned with one or more values with the `write` method (if the value data type does not match the data type of the key, a `TypeError` is raised). A key can be also cleared with the `clear_key` method, with the option to force clear a protected key. Additionally, the `clear` method allows for the clearing of all keys in the memory store, with the option to exclude certain keys and/or force clear protected keys.

Returning to the highlighted problem of following released detectors, a shared data structure with the ability to store a list of `RouteDetector` instances would give the tests access to the detector release history. Such structure is defined in the memory space for the `"DetectorReleaseLog"` key with the parameters `datatype=RouteDetector, store=True`. With access to the shared memory, every `DeactivateDetector` test can log a detector release to the `"DetectorReleaseLog"` structure, and `CheckIfPrevRoutePassed` test can acquire this data.

```python
class DeactivateDetector(SingleTest):
    """release the specified detector and check if its status has switched to
       FREE"""
    ...
    @custom_action(event_type=CustomTags.ON_SUCCESS)
    def register_release(self):
        self.memory.write("DetectorReleaseLog", self.element)


class CheckIfPrevRoutePassed(SingleTest):
    """check if the route has begun setting (changed state to BUILD) after
       the specified detector has been released"""
    ...
    def get_value(self):
        return self.memory.read("DetectorReleaseLog")
```

5.4 `client` package

A client is a kind of software application that accesses a service or resource provided by another software application or device, known as a server. This can include tasks such as retrieving and displaying information, sending and receiving data, and executing specific commands or functions. The client and server communicate over a network, with the client initiating requests and the server responding to those requests.

The `client` package contains modules defining the general client class protocol, as well as the OPC UA client implementation.

5.4.1 `ClientProtocol` and `OpcUAClient` classes

It is possible to use different clients depending on the preferred communication protocol. To ensure future compatibility, any client implementation should follow the `ClientProtocol` class. It defines a set of methods (async functions) that a client should have in order to communicate with the PLC server. These methods include connecting and disconnecting to the server, reading and writing values from a server node, and subscribing and unsubscribing to a server node to receive data change events.

Sidebar: Python `Protocol` classes

⌜ In Python, protocols are informal interfaces that define a set of methods an object should implement in order to be considered "compatible" with a particular protocol. Protocol classes do not usually provide any implementation for these methods themselves, nor are they meant to be used directly.

The concept of protocols in this example is related to the idea of "duck typing" in Python, which refers to the ability of an object to be used in a certain way based on its methods and attributes, rather than its specific class implementation [14]. Such design allows for a more flexible and extensible design, as it allows objects from different classes to be used interchangeably if they implement the same protocol.

The concept of protocols is similar to that of Abstract Base Classes (ABCs) in Python. Both protocols and ABCs define a set of methods and attributes that a class should have in order to be considered a certain type of object. The main difference is that protocols are not enforced by the Python runtime and rely on the developer to ensure that the correct methods are implemented. On the other hand, ABCs are enforced by the runtime and raises an error if a class that inherits from it does not implement the required methods. ⌟

The `OpcUAClient` class, a subclass of the `Client` class from the `asyncua` Python library, is specifically designed to facilitate communication with OPC UA servers. It inherits some of the design aspects from the client class of the `pytester` module, discussed in Chapter 4. It is initialized with the server URL, user name, and password as arguments and implements the interface defined by the `ClientProtocol` class.

One of the key features of the `OpcUAClient` class is the improved node subscription process, which aims at the efficiency increase and easier management of the monitored items by decreasing the number of performed subscriptions. Upon receiving the first subscription request, the class initializes a subscription and stores the corresponding subscription handle in a private attribute. Subsequent subscription requests utilize the initial subscription, adding new monitored items to it as necessary. Additionally, duplicate subscriptions for the same node are avoided in a similar manner. When a new data change subscription request is received for a node, the subscription method proceeds to create a new monitored item on the subscription based on the provided node address and saves the returned monitored item handle to a private class dictionary. The following attempts to subscribe to the same node would result in the immediate subscription method return without performing any further actions. The implemented subscription process is inspired by the *singleton* pattern, which is a software design pattern that ensures that a class has only one instance with a global access point [14].

5.4.2 `NodeSubscriptionHandler` class

Besides the node address, the node subscription method expects a second argument: a *handler*. A handler object defines custom logic and behavior in response to certain events that occur within the OPC UA server.

To qualify as a handler, an object must implement the *callback methods* that are called when such events or data change notifications associated with the subscribed nodes are received [16]. These methods receive specific input arguments such as the node address and the current node value in a certain order. They may evaluate the data, perform any necessary actions, and can be used to update the client application. For example, the `datachange_notification` method is responsible for handling the data change events related to the subscribed nodes. It can be prototyped as:

```
def datachange_notification(self, node: Node, val, data):
    pass
```

This method is implemented in the `OPCNodeSubscriptionHandler` class. Additionally, this class introduces a number of instruments to make the node subscription process easier and to automatically update the element nodes' values stored locally.

First, the class offers a set of registration methods. The `register_element_nodes` method is used to register the nodes of an `Element` object. It takes an `Element` object as an input and, for each node in the `child_nodes` dictionary of the `Element`'s control and test instances, it saves the node address as a key and its name and the containing `child_nodes` dictionary as a value in a private class dictionary. Similarly, the `register_test` method is used to register the tracked node address of a `Test` object. It maps the test's tracked node address to a list of tests that follow this node. These data structures are later used for group node subscriptions and value updates.

Second, the class includes subscription methods. The `subscribe_all_nodes` method subscribes to all the node addresses that are associated with the system elements. On the other hand, the `subscribe_testes_nodes` method only subscribes to the node addresses that are associated with the registered tests. This allows the user to choose the registration method that best suits their needs. If they want to run various test scenarios, subscribing to all the nodes right away may be the preferred method. However, if they are only interested in running a specific test scenario, they have the option to subscribe only to the nodes that refer to that specific scenario.

Third, the class implements the `__aenter__` and `__aexit__` methods, which allow using the handler as an asynchronous context manager. `OPCNodeSubscriptionHandler` is initialized with a `zone` object, control and test `client` objects, and a `subscribe_at_start` flag. When the handler is called as a context manager, it automatically attaches the clients to the element nodes, registers the nodes, and, if the `subscribe_at_start` flag is set, initiates the subscription process with the `subscribe_all_nodes` method.

Fourth, the class has two private methods that automatically update the element nodes' local values and notifies the tests and the data change notification, enabling the enable event-based testing. When a data change notification is received, is processed as defined by the `datachange_notification` method. Upon the call, the handler uses the `_set_element_attr_value` method to update the elements' data, and the `_set_test_events` method to set the `asyncio` event of each test that is awaiting for it. Such an approach allows running the tests more efficiently, since they do not poll the server for the value update, but instead wait for the server to notify them that the value has been changed.

6 Evaluation of the New Testing Program on the Pilsen Tram Depot PLC Control System

To showcase the new program and to verify if it achieves the defined goals, the testing software has been applied to the testing of the control system for one of the company's recent projects, the Pilsen tram depot.

The depot is divided into ten separate zones (Z1 - Z10), each of which is responsible for the tram traffic control in the assigned area. The tram operation is also controlled on the neighboring Slovanská street, from which the trams enter and exit the depot. The PMDP tram fleet has different types of trams: one-way and two-way, single-car, and coupled sets. The system must be functional for all these vehicle types and must detect every vehicle correctly [10].

Zone Z02 has been picked for the testing process. It consists of 18 intersecting routes with given layouts. Each route is equipped with track circuit position detectors, electrical point machines, and a SG lamp. The communication between the system and the trams is realized via the Vetra system. The basic identification feature of all trams is the tram ID, which is read and evaluated at each Vetra receiver.

The development of the control system for the depot has already been finished, and the system has been tested with the current testing tool, `pytester`. To compare the results of the previous testing with the process of testing with the new program, several testing scenarios have been replicated, including the following:

- A static test of the position detector control and test modules functionality
- A static test of the point machine control and test modules functionality
- A simultaneous passage of two colliding routes.

Also, other scenarios, that are only possible to implement with the new software, have been performed:

- A simultaneous passage of two colliding routes - the passages cover the full length of the route, their duration reflects the real physical movement of a tram, and the optimal routing algorithm is disabled.
- A simultaneous passage of four colliding routes.

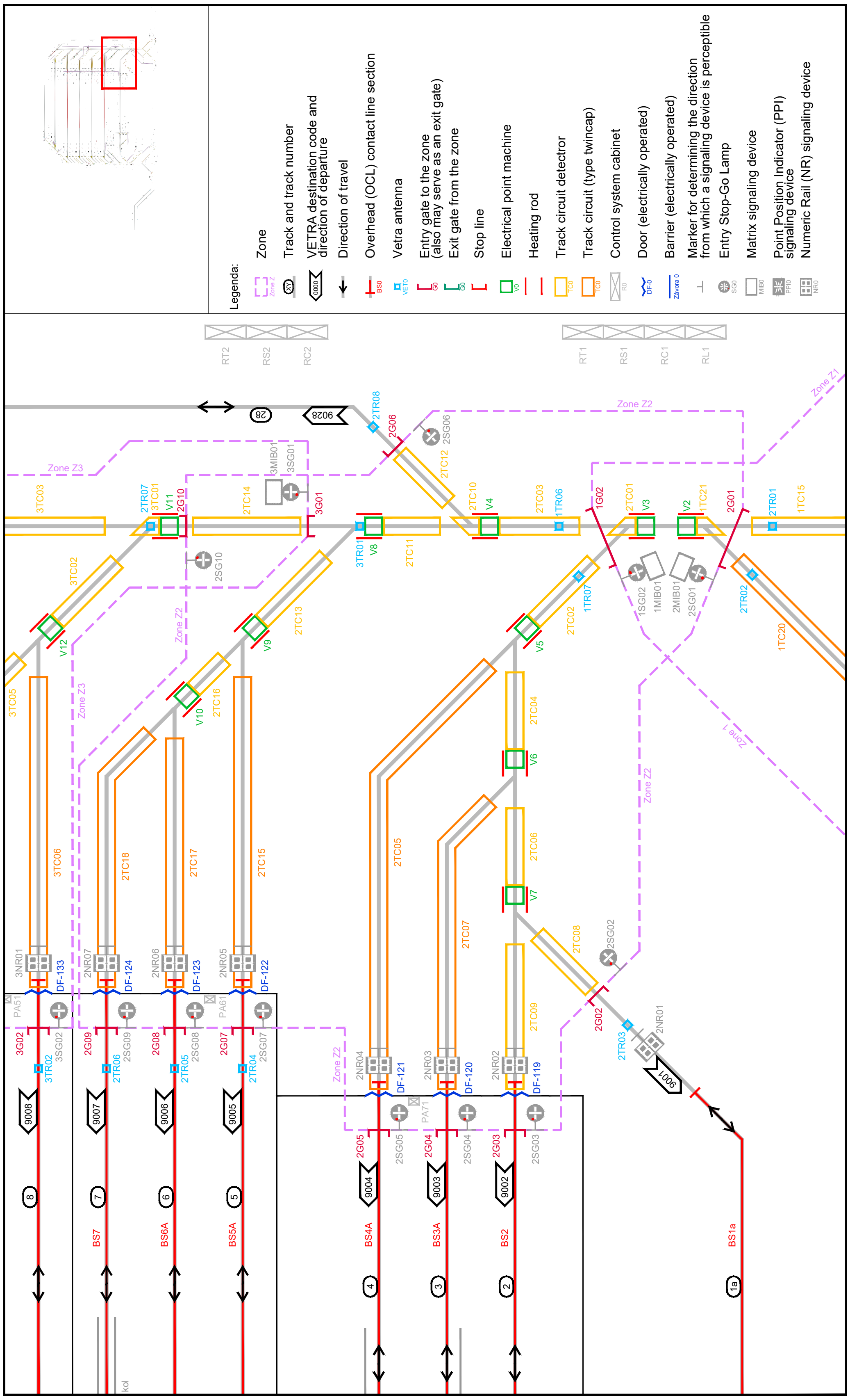

Legenda:
Zone
Track and track number
VETRA destination code and direction of departure
Direction of travel
Overhead (OCL) contact line section
Vetra antenna
Entry gate to the zone (also may serve as an exit gate)
Exit gate from the zone
Stop line
Electrical point machine
Heating rod
Track circuit detector
Track circuit (type twincap)
Control system cabinet
Door (electrically operated)
Barrier (electrically operated)
Marker for determining the direction from which a signaling device is perceptible
Entry Stop-Go Lamp
Matrix signaling device
Point Position Indicator (PPI) signaling device
Numeric Rail (NR) signaling device
Zone Z1
Zone Z2
Zone Z3
RT2
RS2
RC2
RT1
RS1
RC1
RL1
9028
28
9001
9002
9003
9004
9005
9006
9007
9008
BS1a
BS2
BS3A
BS4A
BS5A
BS6A
BS7

Figure 18: Zone Z02 layout of the tram depot in Pilsen, Czechia [10]

Every test scenario is generated with the tools provided by the new program. The tests have been configured either manually or, in the case of the tram passage dynamic tests, by `PassageGenerator`. The test data and the results have been automatically collected by the `TestRunLogger` and stored in separate JSON report files. For the purpose of visualization, a custom Python script is used to convert the reports to interactive HTML files. It is worth noting that the implementation of the generated scenarios, as well as the contents of the report files and their visualization, shown in this chapter, are not definitive and may be modified to suit the needs of the user.

6.1 Static Tests

The purpose of static tests is to verify that the element modules of the control system, described in Chapter 3, function as intended. Scenarios for such tests do not implement a sequential logic, where one event is supposed to happen after another, but rather contain a series of different unrelated commands. Generally speaking, these tests may validate the reaction of the control module on commands from the control and test structures. Correct reaction on a command from a test structure ensures that the control modules' digital twin counterparts are connected and function properly. Correct reaction on a command from the control module itself implies the functionality of the control system. For example, in the case of the point machine control module, a `setLeft` command from the test structure reflects the situation, when the point machine sensors signalize that it is in the left position. This command may model the situation, where the point machine has been switched manually. On the contrary, a `cmdMoveLeft` command from the control structure means that the control system is trying to switch the point machine to the left, for example, to build a route. However, both these tests evaluate the outcome based on the `status` value of the control structure of the device.

6.1.1 Case 1: Test of the track circuits functionality

A static test of a track circuit checks that the control structure of the device is properly connected to the test module. The testing scenario starts with a sequence of `setTrue` and `setFalse` commands from the test module, which simulates the occupation and the release of the detector, respectively. These tests should result in the detector control module status value switching to `1` in case of occupation and to `0` in case of release. Then, the detector's test module complementary outputs, `outNO` and `outNC`, are toggled directly. The purpose of these outputs is to verify the signals of one another (if `outNO==1`, then `outNC==0`, and vice versa). If both outputs have the same value, either `0` or `1`, the status of the detector's control module should set the `errorInputCompl` bit to `1`.

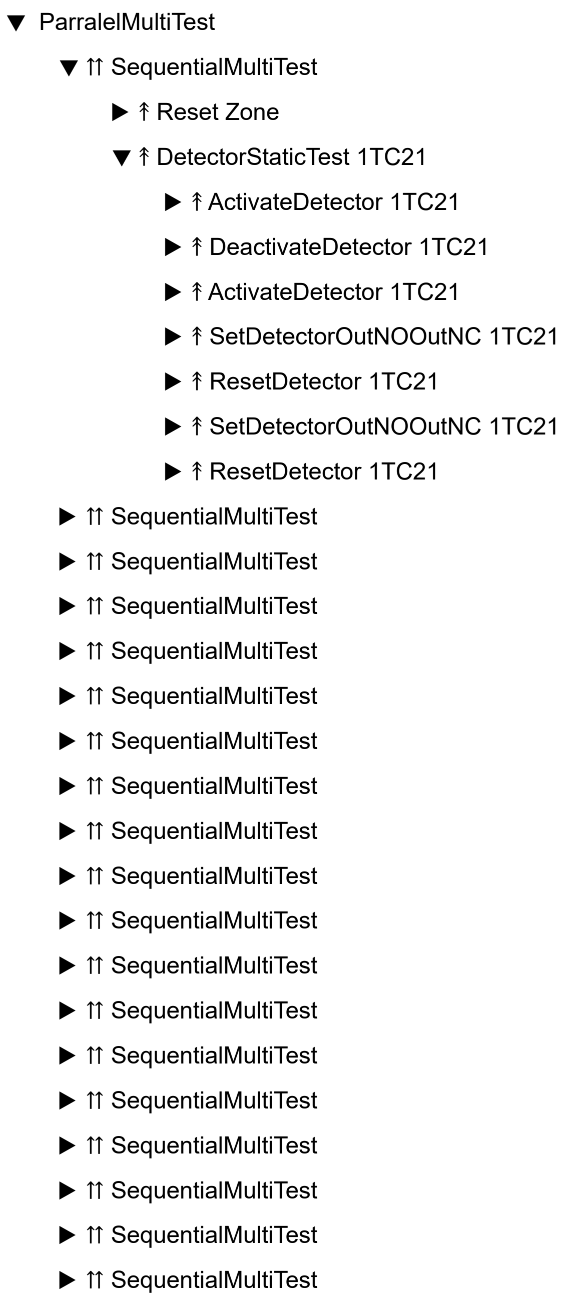
This testing scenario is recreated in a series of `SingleTest` objects, wrapped in a `SequentialMultiTest`. The source code of one of the tests, `SetDetectorOutNOOutNC SingleTest`, is provided below to illustrate the process of the custom test configuration.

```
class SetDetectorOutNOOutNC(testlib.SingleTest):
    """Set compl. outputs to the same value
    and check errorInputCompl status bit"""
    def __init__(self, detector: Detector, set_to: bool, assured_duration:
        float = 0, timeout: float = 10, require_datachange: bool = True):
        self.set_to = set_to
        super().__init__(element=detector,
            expected_value=Detector.ControlStateBits.ERROR_INPUT_COMPL,
            mask=Detector.ControlStatusBitMask.ERROR_INPUT_COMPL,
            assured_duration=assured_duration,
            timeout=timeout,
            require_datachange=require_datachange)

    async def command(self):
        if self.set_to is True:
            cmd_outNO = "outNO.setTrue"
            cmd_outNC = "outNC.setTrue"
        else:
            cmd_outNO = "outNO.setFalse"
            cmd_outNC = "outNC.setFalse"
        async with asyncio.TaskGroup() as tg:
            tg.create_task(self.element.test.write_value(cmd_outNO, True))
            tg.create_task(self.element.test.write_value(cmd_outNC, True))

    # Since SetDetectorOutNOOutNC verifies the value of the status in the
        control structure, it may utilize the inherited get_value method.
```

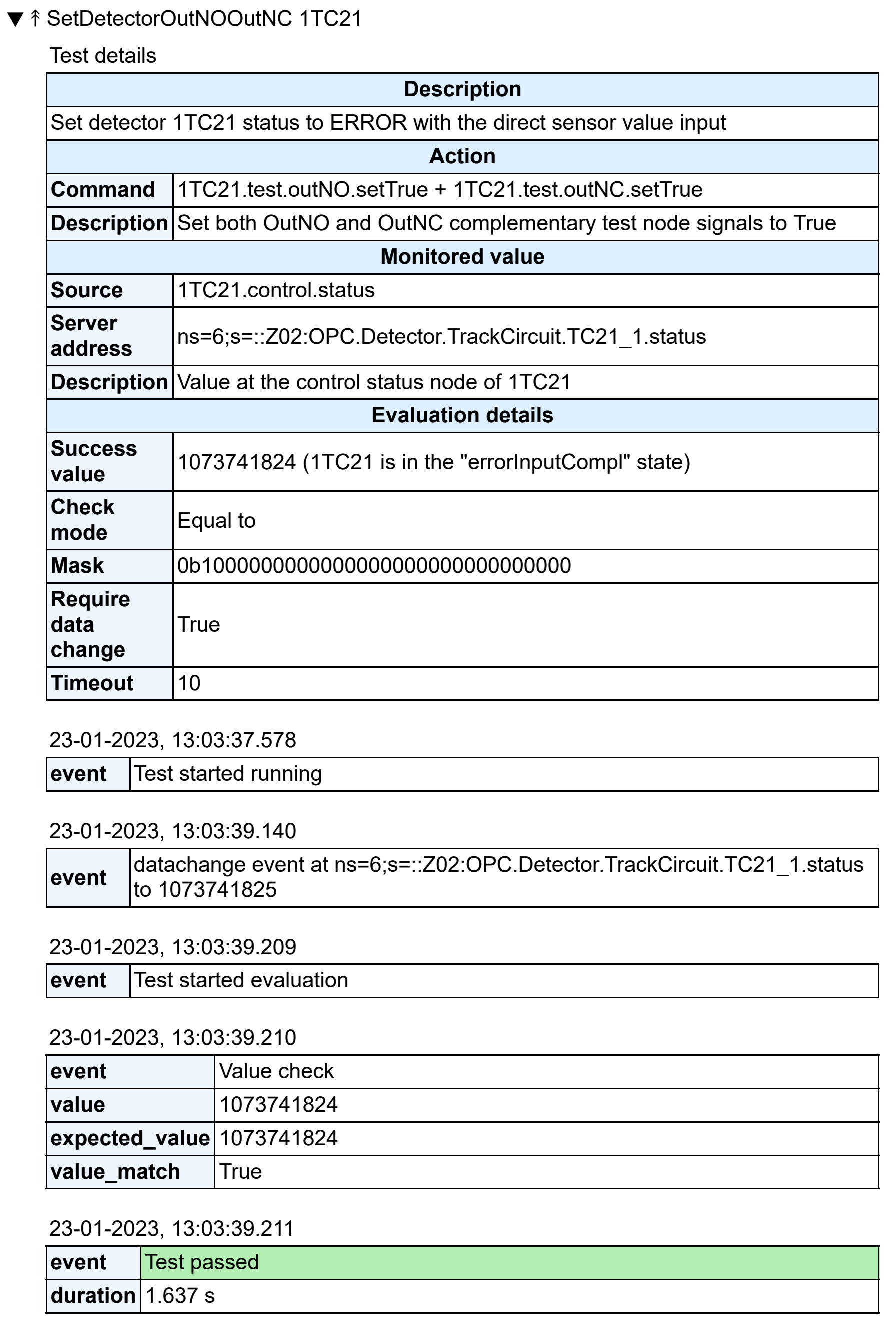
The created `SequentialMultiTest` test scenario is applied to each detector module in the control system. To test all the detector modules simultaneously, these scenarios are enfolded once again in a `ParallelMultiTest` wrapper. The testing is then started with a single `run` command.



```
▼ ParralelMultiTest
    ▼ ⥣ SequentialMultiTest
        ▶ ↟ Reset Zone
        ▼ ↟ DetectorStaticTest 1TC21
            ▶ ↟ ActivateDetector 1TC21
            ▶ ↟ DeactivateDetector 1TC21
            ▶ ↟ ActivateDetector 1TC21
            ▶ ↟ SetDetectorOutNOOutNC 1TC21
            ▶ ↟ ResetDetector 1TC21
            ▶ ↟ SetDetectorOutNOOutNC 1TC21
            ▶ ↟ ResetDetector 1TC21
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
    ▶ ⥣ SequentialMultiTest
```

Figure 19: Wrapped track circuit testing scenarios. The double-tip arrow represents a `SequentialMultiTest`, and two parallel arrows represent a `ParallelMultiTest`.

The results of the test are saved to a JSON report file and visualized to an interactive HTML page with tables with the data about the occurred events printed in chronological order. The report table of the `SetDetectorOutNOOutNC` test, the implementation of which has been discussed above, is illustrated below.



▼ ⇡ SetDetectorOutNOOutNC 1TC21

Test details

Description	
Set detector 1TC21 status to ERROR with the direct sensor value input	
Action	
Command	1TC21.test.outNO.setTrue + 1TC21.test.outNC.setTrue
Description	Set both OutNO and OutNC complementary test node signals to True
Monitored value	
Source	1TC21.control.status
Server address	ns=6;s=::Z02:OPC.Detector.TrackCircuit.TC21_1.status
Description	Value at the control status node of 1TC21
Evaluation details	
Success value	1073741824 (1TC21 is in the "errorInputCompl" state)
Check mode	Equal to
Mask	0b1000000000000000000000000000000
Require data change	True
Timeout	10

23-01-2023, 13:03:37.578

event	Test started running

23-01-2023, 13:03:39.140

event	datachange event at ns=6;s=::Z02:OPC.Detector.TrackCircuit.TC21_1.status to 1073741825

23-01-2023, 13:03:39.209

event	Test started evaluation

23-01-2023, 13:03:39.210

event	Value check
value	1073741824
expected_value	1073741824
value_match	True

23-01-2023, 13:03:39.211

event	Test passed
duration	1.637 s

Figure 20: Report for the `SetDetectorOutNOOutNC` test of detector 1TC21

The `pytester` program performed a similar testing scenario sequentially for every track circuit in the system. In the case of zone Z02, nineteen track circuits 1TC21, 2TC01 - 2TC18 have been tested with an average one module testing time of 8 seconds, and an overall testing time of 152.73 seconds (approx. 2 minutes 33 seconds). The new testing program performed the same testing slightly longer - 11.18 seconds per module on average. However, due to the fact that the program has performed the testing scenarios for all the track circuit modules simultaneously, the overall time decreased to 12 seconds.

6.1.2 Case 2: Test of the point machines functionality

A static test of a point machine functionality checks the response of the point machine software control module to setting it to the right, left, and middle positions, and resetting it. The commands first come from the point machine digital twin function (e.g. `pme.test.setLeft`, or `pme.test.outLeft01.setTrue` for the direct sensor value setting), and then from the control node itself (e.g. `pme.control.cmdMoveRight`). Implementation of this scenario in the new program is similar to the one described in section 6.1.1 above.

The reaction of the control module is determined by the control node status value. The bit on index **0** of the point machine status **dword** is set when its tongue position sensors report position "left", the bit on index **1** is set when position "right" is reported, the bit on index **2** corresponds to position "middle", and bit on index `30` is set in case of an error, for instance, when the tongue position detectors output the logical `True` signal for both positions. The "reset" command unblocks the point machine and sets it to the position "left". To test the point machine module, each of the above-mentioned situations is simulated one after the other.

The `pytester` program again performed the testing scenario sequentially for eight point machines V03 - V10 of zone Z02. The test finished with an average one module testing time of 31.5 seconds and an overall testing time of 219.36 seconds (approx. 3 minutes 40 seconds). The new testing program performed the same scenario with a similar one module average testing time of 27.5 seconds. However, as expected, the overall testing time decreased to 31.7 seconds due to the concurrent test evaluation.

6.2 Dynamic Route Passages

Dynamic testing aims at validating the routing logic aspects, described in section 3.5: route claiming, route build prioritizing, tram passage simulation, etc.

During the testing, both single passages and multiple simultaneous passages are validated. The testing scenario that the `pytester` program applies has been discussed in section 4.3.2 during the `01_generatedDynamic.csv` scenarios file discussion. Its main parts are briefly recapitulated for the sake of comparison with the new testing process.

The testing sequence starts with a route request test. A route request string message is inserted into the memory of the route's start gate. `pytester` evaluates a successful request based on the route status (the route should either switch to READY (`status==3`), if all its elements are free, or remain FREE (`status==0`) if it is colliding with another currently active route). During the single tram passage simulation, the program also checks the correct position of the point machines. The moment the route switches to the READY state, route position detectors are occupied and subsequently freed in the order of their appearance in the route layout, simulating a tram passing through them. The occupied detector should have the `status` value of `3` ("occupied" bit `0b01` set + "claimed" bit `0b10` set). During each step, the route status is checked. Freeing the last detector should switch the route to the FREE state while freeing other detectors should not affect the status value (i.e. the route must remain OCCUPIED).

The new testing program introduces several `SingleTest` objects combined in a `SequentialMultiTest` wrapper as a reproduction of this process. `RequestRoute` test performs the requesting of the route, but instead of the route status, it follows the start gate memory and checks, if the string route name can be found in the last row of the gate memory string. After the request has been received and recorded, the `WaitForRoute MultiTest` starts. It checks if the route is set correctly by sequentially performing the `CheckIfRouteReady` test, which evaluates the status of the route, and the number of tests that verify the statuses of the route elements. Finally, the passage itself is represented by the `ActivateDetector` and `DeactivateDetector` tests. The latter run sequentially with the `CheckIfRouteOccupied` and `CheckIfRouteFree` tests and saves the freed detector object in the shared test memory. To simulate two or more simultaneous route passages, such scenarios can be wrapped in the `ParallelMultiTest` wrapper.

6.2.1 Case 1: Two routes, fast mode, passage truncation

In the case of two routes, the most often dynamic testing scenario modification is to request both routes one after another and follow the passages on the part of the layout they have in common. The first route's passage is followed to the first detector after the last common point machine or crossing. After this detector has been freed, it is verified if the second route has started building in time. If so, its first detector is occupied, and the testing scenario is finished.

To showcase a dynamic test execution, a scenario for the passage of route 2R01 and route 2R03 has been generated by `PassageGenerator` with parameters `truncate = True`, `fastmode = True`, `optimal_routing = True`. The structure of the produced `SequentialMultiTest` can be observed in fig. 21. Also, both routes have been assigned with `Tram` objects with a length of 25 meters and a speed of 10 km/h.

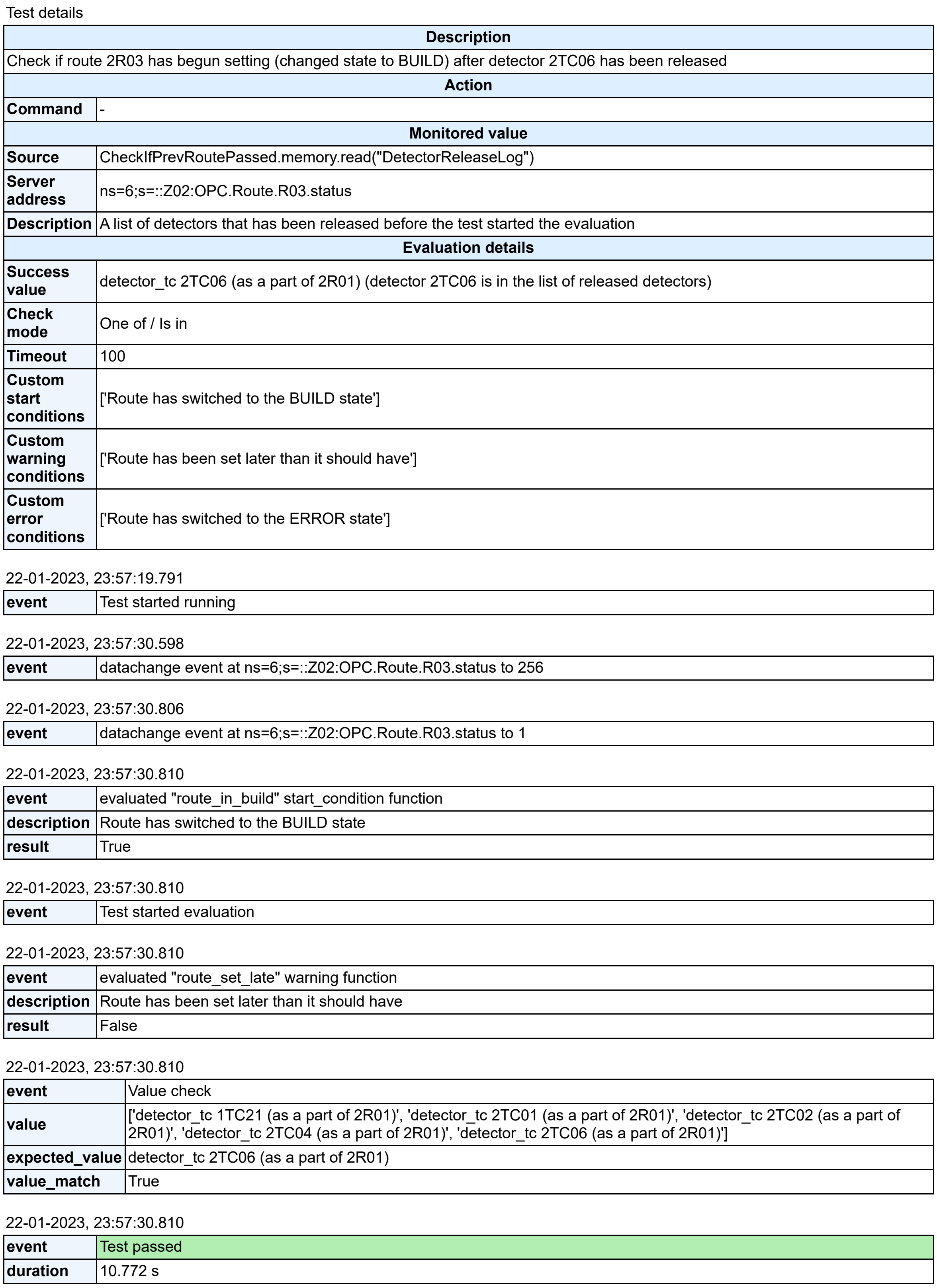
Both routes start at gate 2G01; their layouts intersect on the segment between the position detectors 1TC21 and 2TC06. If a tram requests a passage through route 2R01, the next tram, that needs to pass through route 2R03, has to wait until detector 2TC06 is released. The layouts can be seen in detail in fig. 18.

During the passage of route 2R01, detectors 1TC21 - 2TC06 have been occupied and subsequently freed and added to the shared test memory. The `CheckIfPrevRoutePassed` test, which has been waiting for the route 2R03 to switch to the BUILD state, then has accessed the memory to evaluate if the detector 2TC06 object is the last object in the released detectors list. Fig. 22 illustrates the successful evaluation of this test. If 2TC06 were not in the list, the test would evaluate a fail (this situation is modeled in the next case study), and if there were detectors in the that had been released after 2TC06, the test would raise a warning `"route_set_late"`, but still evaluate as passed.

The testing scenario took 15.1 seconds to complete, which is a slight improvement over the testing time of 16.7 seconds when performed by `pytester`. However, the main advantage of the new program is the great number of testing details that it provides, and the ability to chronologically visualize them in interactive data tables.

- ▼ Routes 2R01, 2R03 passsage
  - ▼ ⇡ Reset Zone
    - ▶ ⇡ ResetSystem
    - ▶ ⇡ SetZoneToAB
    - ▶ ⇡ SetZoneToNORM
  - ▼ ⇡ Request routes 2R01, 2R03
    - ▶ ⇡ RequestRoute 2R01
    - ▶ ⇡ RequestRoute 2R03
  - ▼ ⇡ Routes 2R01, 2R03 dynamic passsages
    - ▼ ⇈ Route 2R01 dynamic passsage
      - ▶ ⇡ WaitForRoute 2R01
      - ▶ ⇡ ActivateDetector 1TC21
      - ▶ ⇡ ActivateDetector 2TC01
      - ▶ ⇡ ActivateDetector 2TC02
      - ▼ ⇡ DeactivateDetector 1TC21 + CheckIfRouteOccupied 2R01
        - ▶ ⇡ DeactivateDetector 1TC21
        - ▶ ⇡ CheckIfRouteOccupied 2R01
      - ▶ ⇡ ActivateDetector 2TC04
      - ▶ ⇡ DeactivateDetector 2TC01 + CheckIfRouteOccupied 2R01
      - ▶ ⇡ ActivateDetector 2TC06
      - ▶ ⇡ DeactivateDetector 2TC02 + CheckIfRouteOccupied 2R01
      - ▶ ⇡ DeactivateDetector 2TC04 + CheckIfRouteOccupied 2R01
      - ▼ ⇡ DeactivateDetector 2TC06 + CheckIfRouteOccupied 2R01
        - ▶ ⇡ DeactivateDetector 2TC06
        - ▶ ⇡ CheckIfRouteOccupied 2R01
    - ▼ ⇈ Route 2R03 dynamic passsage
      - ▶ ⇡ WaitForRoute 2R03
      - ▶ ⇡ ActivateDetector 1TC21

Figure 21: Testing scenario for routes 2R01 and 2R03 passage produced by `PassageGenerator` with the parameters `truncate = True`, `fastmode = True`, `optimal_routing = True`.



Test details

Description	
Check if route 2R03 has begun setting (changed state to BUILD) after detector 2TC06 has been released	
Action	
Command	-
Monitored value	
Source	CheckIfPrevRoutePassed.memory.read("DetectorReleaseLog")
Server address	ns=6;s=::Z02:OPC.Route.R03.status
Description	A list of detectors that has been released before the test started the evaluation
Evaluation details	
Success value	detector_tc 2TC06 (as a part of 2R01) (detector 2TC06 is in the list of released detectors)
Check mode	One of / Is in
Timeout	100
Custom start conditions	['Route has switched to the BUILD state']
Custom warning conditions	['Route has been set later than it should have']
Custom error conditions	['Route has switched to the ERROR state']

22-01-2023, 23:57:19.791

event	Test started running

22-01-2023, 23:57:30.598

event	datachange event at ns=6;s=::Z02:OPC.Route.R03.status to 256

22-01-2023, 23:57:30.806

event	datachange event at ns=6;s=::Z02:OPC.Route.R03.status to 1

22-01-2023, 23:57:30.810

event	evaluated "route_in_build" start_condition function
description	Route has switched to the BUILD state
result	True

22-01-2023, 23:57:30.810

event	Test started evaluation

22-01-2023, 23:57:30.810

event	evaluated "route_set_late" warning function
description	Route has been set later than it should have
result	False

22-01-2023, 23:57:30.810

event	Value check
value	['detector_tc 1TC21 (as a part of 2R01)', 'detector_tc 2TC01 (as a part of 2R01)', 'detector_tc 2TC02 (as a part of 2R01)', 'detector_tc 2TC04 (as a part of 2R01)', 'detector_tc 2TC06 (as a part of 2R01)']
expected_value	detector_tc 2TC06 (as a part of 2R01)
value_match	True

22-01-2023, 23:57:30.810

event	Test passed
duration	10.772 s

Figure 22: The test evaluates if route 2R03 has been set after detector 2TC06 has been released. The "Value check" event at 23:57:30.810 remarks the compared values.

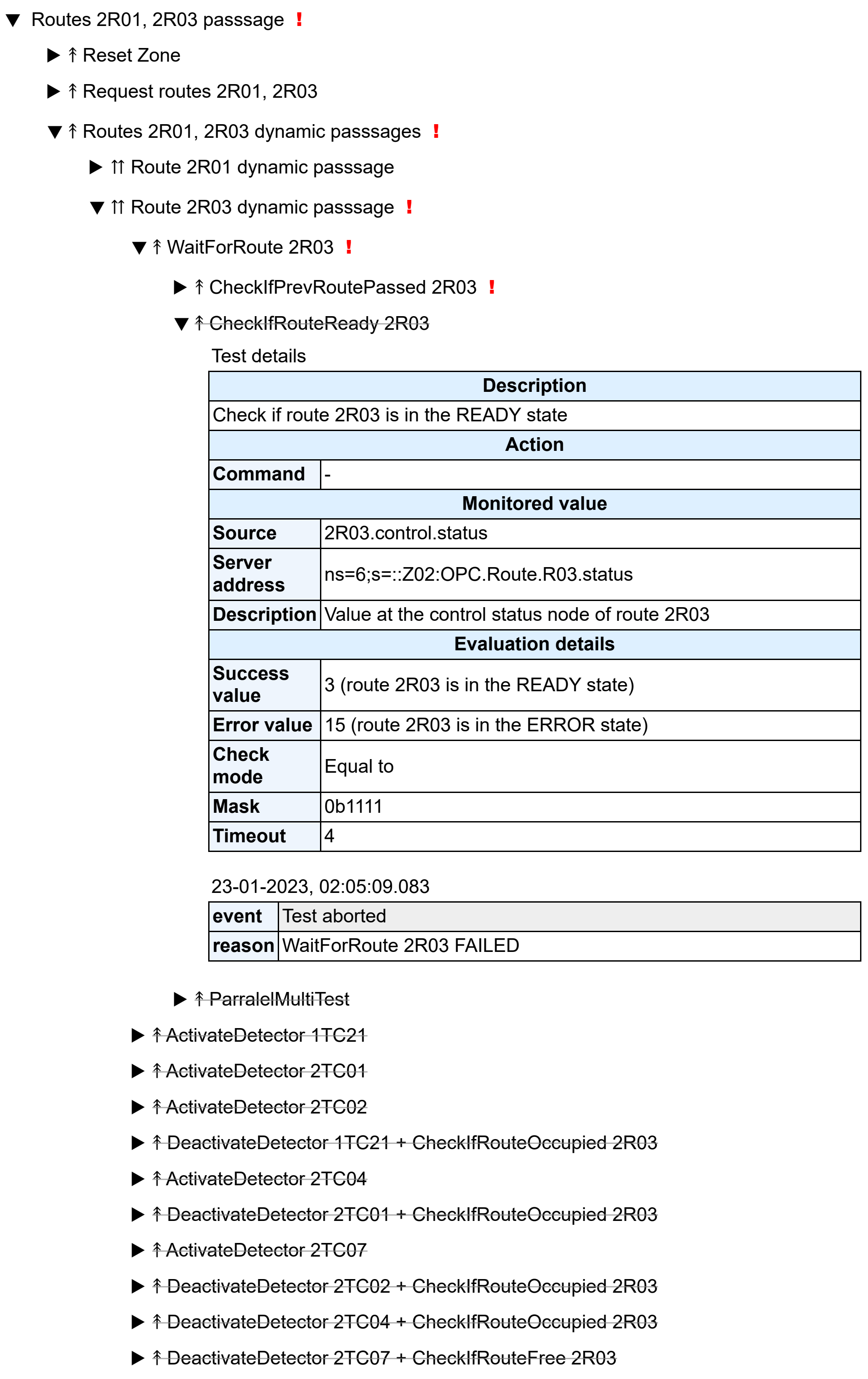
6.2.2 Case 2: Two routes, no truncation, enforced test fail

With the new testing program, it is also possible to simulate a passage of the full route (or a combination of routes). Such an elongated testing scenario allows for complete and thorough testing of a passage from the start of the route till its end. Although such a scenario is already implemented in the `pytester` program, it is limited to only single route passage simulation. On the contrary, `PassageGenerator` allows for the creation of such testing scenarios for any combination of routes. To generate a sequence of tests that represents a passage of the full route, the `truncate` parameter must be set to `False`.

With that modification, the testing scenario for simultaneous passage of routes 2R01 and 2R03 now includes all the layout elements of both routes and ends with the release of the last detector in the route layouts - 2TC08 and 2TC07, respectively. Also, the `CheckIfRouteFree` tests run in parallel with the `DeactivateDetector` tests (in the previous test case, the testing ended with the `ActivateDetectror` test of 1TC21, which is the first detector in the layout).

To showcase the program behavior during a test fail, the testing scenario has been further modified by setting the `optimal_routing` parameter to `False`. This results in the fact that the test now expects route 2R03 to be set after route 2R01 has been completely passed, and all of its elements, including the ones that do not comprise the layout of route 2R03, have been freed. Since the control system still sets the routes in an optimal way, route 2R03 is set before the detector 2TC08 is released, causing the test to evaluate an error.

After the error occurs, the program logs the result and aborts the following tests that have not started running. The testing scenario took 13.3 seconds. The structure of the test, as well as its results, can be seen in fig. 23. It can be seen that the passage test of route 2R03 has been evaluated as successful. However, when the `CheckIfPrevRoutePassed` test of route 2R03 has read the test memory and has accessed the list of released detectors, it has not found the expected detector `2TC08` object among them, subsequently evaluating itself as failed.



- ▼ Routes 2R01, 2R03 passsage ❗
  - ▶ ↟ Reset Zone
  - ▶ ↟ Request routes 2R01, 2R03
  - ▼ ↟ Routes 2R01, 2R03 dynamic passsages ❗
    - ▶ ⇈ Route 2R01 dynamic passsage
    - ▼ ⇈ Route 2R03 dynamic passsage ❗
      - ▼ ↟ WaitForRoute 2R03 ❗
        - ▶ ↟ CheckIfPrevRoutePassed 2R03 ❗
        - ▼ ~~↟ CheckIfRouteReady 2R03~~

          Test details

Description	
Check if route 2R03 is in the READY state	
Action	
Command	-
Monitored value	
Source	2R03.control.status
Server address	ns=6;s=::Z02:OPC.Route.R03.status
Description	Value at the control status node of route 2R03
Evaluation details	
Success value	3 (route 2R03 is in the READY state)
Error value	15 (route 2R03 is in the ERROR state)
Check mode	Equal to
Mask	0b1111
Timeout	4

          23-01-2023, 02:05:09.083

event	Test aborted
reason	WaitForRoute 2R03 FAILED

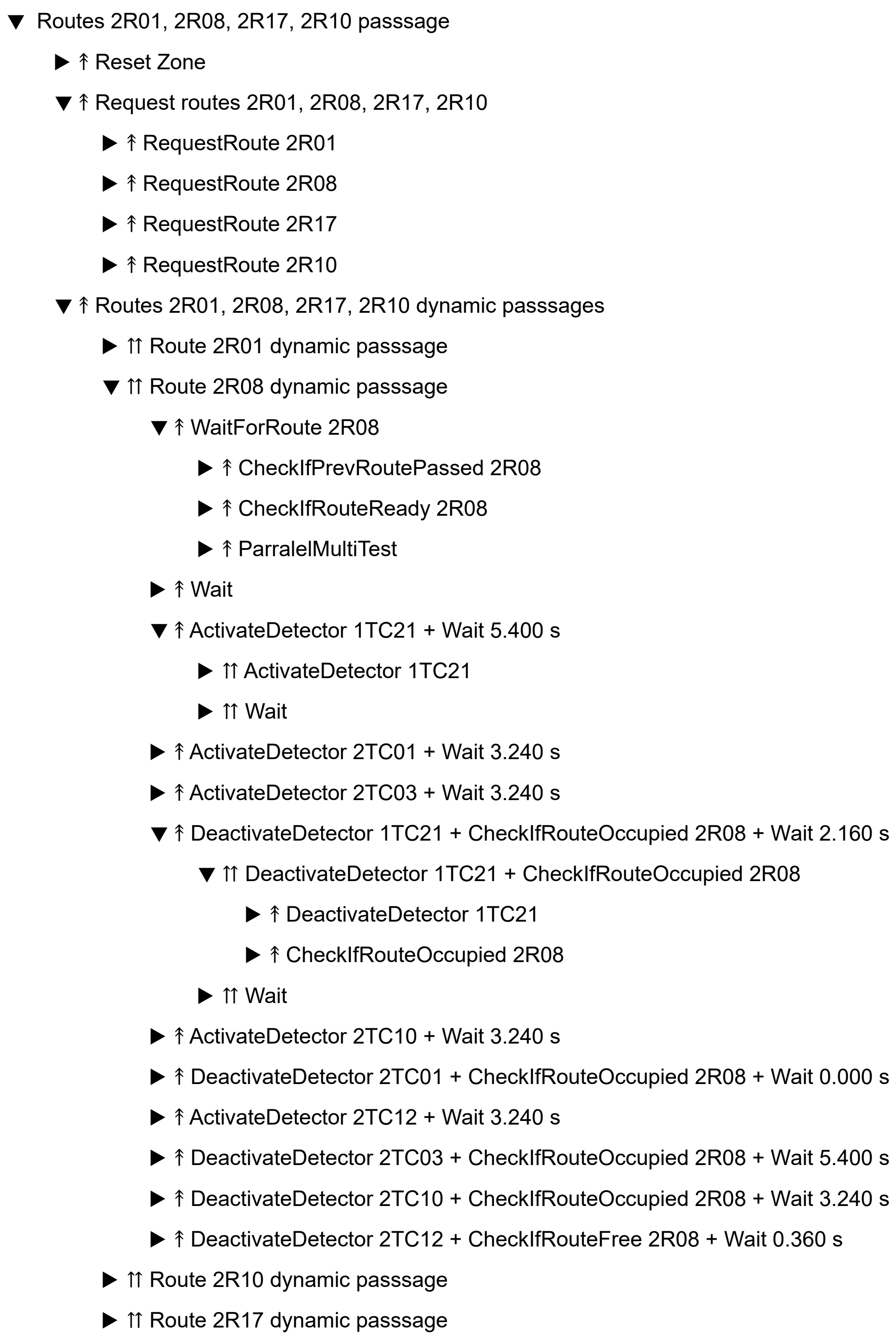
        - ▶ ~~↟ ParralelMultiTest~~
      - ▶ ~~↟ ActivateDetector 1TC21~~
      - ▶ ~~↟ ActivateDetector 2TC01~~
      - ▶ ~~↟ ActivateDetector 2TC02~~
      - ▶ ~~↟ DeactivateDetector 1TC21 + CheckIfRouteOccupied 2R03~~
      - ▶ ~~↟ ActivateDetector 2TC04~~
      - ▶ ~~↟ DeactivateDetector 2TC01 + CheckIfRouteOccupied 2R03~~
      - ▶ ~~↟ ActivateDetector 2TC07~~
      - ▶ ~~↟ DeactivateDetector 2TC02 + CheckIfRouteOccupied 2R03~~
      - ▶ ~~↟ DeactivateDetector 2TC04 + CheckIfRouteOccupied 2R03~~
      - ▶ ~~↟ DeactivateDetector 2TC07 + CheckIfRouteFree 2R03~~

Figure 23: Testing scenario for routes 2R01 and 2R03 with `optimal_routing = False`. The red exclamation mark signalizes that have of the tests from the test scenario have failed, and the gray line goes through the names of the aborted tests.

6.2.3 Case 3: Four routes, real-life duration, full passage length

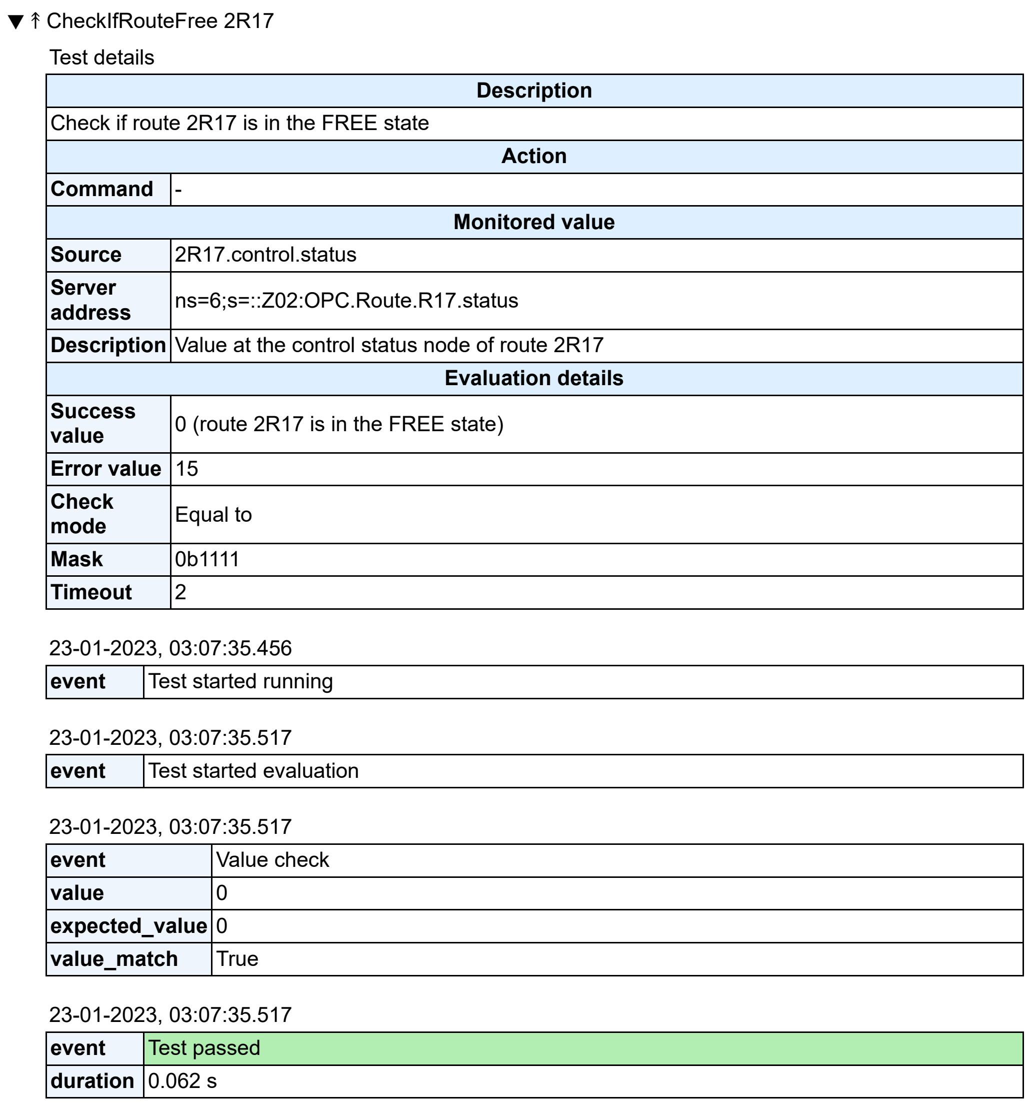
Full route passages, described in the previous test case, can be combined with the passage duration elongation by insertion of the test waiting periods. These periods are represented by the `Wait` objects, which implement the `sleep` function of the `asyncio` library (this function suspends the current task, but allows other tasks to run). The resulting scenario reflects the movement of a real tram between the route detectors. For example, corresponding to the entry gate 2G01, the layout offset of the start of detector 1TC21 (i.e. the distance between 2G01 and the start of 1TC21) is estimated at 1 meter, and the layout offset of the start of detector 2TC01 is estimated at 16 meters. Thus, the distance between the start points of the detectors is 15 meters. If a tram moves at the speed of 10 km/h, or approximately 2.78 m/s, it should take approximately 5.4 seconds to reach the start of detector 2TC01 area from the start of detector 2TC01 area. This value matches the waiting period included after detector 1TC21 is occupied (see fig. 24 - `ActivateDetector 2TC01` test). The calculation of detector release times is similar. The only difference is that the length of the tram should be taken into consideration. In the new program, the calculation of the waiting periods' duration is managed by `PassageGenerator`. The delays can be added to the testing scenario by setting the `fastmode` flag to `False`.

To fully showcase this testing scenario, a simultaneous passage through four routes 2R01, 2R08, 2R17, and 2R10 has been modeled. For such a combination, it is hard to manually estimate the critical common points in route layouts and predict the order of the passages. However, unlike the `pytester` program, `PassageGenerator` is able to handle such scenario generation through the utilization of an implemented algorithm for the estimation of route passage order (for details, see sec. 5.3.3). The structure of the generated test may be seen in fig. 24. The testing scenario took 118 seconds to complete and has been evaluated as successful after the `CheckIfRouteFree` test of route 2R17 has passed, as may be seen in fig. 25.



```
▼ Routes 2R01, 2R08, 2R17, 2R10 passsage
    ▶ ↟ Reset Zone
    ▼ ↟ Request routes 2R01, 2R08, 2R17, 2R10
        ▶ ↟ RequestRoute 2R01
        ▶ ↟ RequestRoute 2R08
        ▶ ↟ RequestRoute 2R17
        ▶ ↟ RequestRoute 2R10
    ▼ ↟ Routes 2R01, 2R08, 2R17, 2R10 dynamic passsages
        ▶ ⇈ Route 2R01 dynamic passsage
        ▼ ⇈ Route 2R08 dynamic passsage
            ▼ ↟ WaitForRoute 2R08
                ▶ ↟ CheckIfPrevRoutePassed 2R08
                ▶ ↟ CheckIfRouteReady 2R08
                ▶ ↟ ParralelMultiTest
            ▶ ↟ Wait
            ▼ ↟ ActivateDetector 1TC21 + Wait 5.400 s
                ▶ ⇈ ActivateDetector 1TC21
                ▶ ⇈ Wait
            ▶ ↟ ActivateDetector 2TC01 + Wait 3.240 s
            ▶ ↟ ActivateDetector 2TC03 + Wait 3.240 s
            ▼ ↟ DeactivateDetector 1TC21 + CheckIfRouteOccupied 2R08 + Wait 2.160 s
                ▼ ⇈ DeactivateDetector 1TC21 + CheckIfRouteOccupied 2R08
                    ▶ ↟ DeactivateDetector 1TC21
                    ▶ ↟ CheckIfRouteOccupied 2R08
                ▶ ⇈ Wait
            ▶ ↟ ActivateDetector 2TC10 + Wait 3.240 s
            ▶ ↟ DeactivateDetector 2TC01 + CheckIfRouteOccupied 2R08 + Wait 0.000 s
            ▶ ↟ ActivateDetector 2TC12 + Wait 3.240 s
            ▶ ↟ DeactivateDetector 2TC03 + CheckIfRouteOccupied 2R08 + Wait 5.400 s
            ▶ ↟ DeactivateDetector 2TC10 + CheckIfRouteOccupied 2R08 + Wait 3.240 s
            ▶ ↟ DeactivateDetector 2TC12 + CheckIfRouteFree 2R08 + Wait 0.360 s
        ▶ ⇈ Route 2R10 dynamic passsage
        ▶ ⇈ Route 2R17 dynamic passsage
```

Figure 24: Testing scenario for routes 2R01, 2R08, 2R17, and 2R10 with `fastmode = False`. The `Wait` tests reflect the movement of the tram between the elements.



▼ ⇡ CheckIfRouteFree 2R17

Test details

Description	
Check if route 2R17 is in the FREE state	
Action	
Command	-
Monitored value	
Source	2R17.control.status
Server address	ns=6;s=::Z02:OPC.Route.R17.status
Description	Value at the control status node of route 2R17
Evaluation details	
Success value	0 (route 2R17 is in the FREE state)
Error value	15
Check mode	Equal to
Mask	0b1111
Timeout	2

23-01-2023, 03:07:35.456

event	Test started running

23-01-2023, 03:07:35.517

event	Test started evaluation

23-01-2023, 03:07:35.517

event	Value check
value	0
expected_value	0
value_match	True

23-01-2023, 03:07:35.517

event	Test passed
duration	0.062 s

Figure 25: Testing scenario for routes 2R01, 2R08, 2R17, and 2R10 is evaluated as successful after the `CheckIfRouteFree 2R17` test has passed.

Assuming that the digital twins of the control system elements imitate their behavior with sufficient preciseness and that the route layout coordinates correspond with the physical location of the position detectors, point machines, and crossings, simulating such passages can be beneficial in several ways. By simulating multiple trams passing through a controlled area at the same time, the test scenario becomes more complex and closer to real-world situations, providing a more comprehensive evaluation of the control system. It may be also possible to identify the limitations of both software and hardware components and evaluate the robustness of the system.

7 Conclusion

The thesis has presented the development of a testing program for a PLC signaling control system for tram lines.

The new testing program development started with a comprehensive examination of the hardware and software components of the control system. It provided valuable information about the characteristics of the system and its main actors. It also helped to understand the specifics of different control system modules and to shape the new software to cover a wide range of possible testing scenarios.

Then, the current testing program has been analyzed. It provided valuable insights into the specifics of the testing process and helped to understand the communication process between the PLC and the testing program as well as uncovered the main downsides of the software - lack of flexibility in creating new testing scenarios, poor maintainability and scalability, and suboptimal performance in terms of the testing speed.

After all the steps of the system analysis had been performed, the gained knowledge was used to create a new testing program. The main goals of the program development were to address the above-mentioned issues and introduce further improvements, such as detailed interactive test reports. The program was created with an object-oriented approach with the implementation of interface classes and modular design. A number of utility classes were introduced to enable the creation of complex dynamic tram passage testing scenarios. Also, a big improvement was the successful implementation of the asynchronous code execution model.

The new program was successfully validated through the conduct of several tests of the control system of a tram depot control system in Pilsen, Czechia. The results of these tests demonstrate that the updated program is a significant improvement over the previous one. The static tests of the control system modules (track circuit position detectors and point machines) showed a notable overall testing time improvement. Generally speaking, due to asynchronous execution, contrary to the old testing program, the number of tested modules does not affect the overall testing time. In the case of detector testing, the average testing time for one module was 11.18 seconds, and the overall testing time was 12 seconds. In the case of point machines testing, the program finished the testing in 27.5 seconds per module on average and 31.7 seconds in total. Dynamic tram passage tests showcased the capabilities of the PassageGenerator module. It was able to generate complex passage scenarios through multiple routes simultaneously. Passages of 2 routes, both truncated and full-length, and the passage of 4 routes were successfully implemented. The results of these scenarios show that the new program is able to accurately and efficiently test the system, providing a great number of testing details and the ability to chronologically visualize them in interactive data tables.

Several improvements will be introduced in the future. A potential issue may be the increased size of the test report files. In the case of complex system testing with a big number of modules, the overall size of the JSON report files may be around 100 Mb, so the optimization of the report file structure should be done. Additionally, a more comprehensive visualization tool may be implemented to improve the design of the HTML report and make the JSON-to-HTML conversion tool more versatile and robust. Next, with the implementation of the direct support of the SHV communication protocol, it will be necessary to assess the testing program's correct function once again. Finally, a Telegram chatbot may be created to notify the user about the events during the testing (test fails, timeouts, etc.).

References

[1] Elektroline a.s. *Point machines for tramways & light rail.* Brochure. 2022.

[2] Elektroline a.s. *Tramway signaling systems.* Brochure. 2022.

[3] J. Anderson. “Speed Up Your Python Program With Concurrency”. In: (2019). [Online; accessed Jan-2023]. URL: `https://realpython.com/async-features-python/`.

[4] B&R Industrial Automation. *B&R Coated X20 Systems Catalogue.* [Online; accessed Jan-2023]. 2022. URL: `https://www.br-automation.com/en/products/io-systems/coated-x20-systems/`.

[5] International Electrotechnical Commission. *IEC 61131-3 Standard.* Geneva, Switzerland, 2013.

[6] OPC Foundation. *OPC Unified Architecture Specification.* [Online; accessed Jan-2023]. 2022. URL: `https://opcfoundation.org/developer-tools/specifications-unified-architecture`.

[7] Python Software Foundation. *asyncio — Asynchronous I/O, event loop, and concurrency tools.* [Online; accessed Jan-2023]. URL: `https://docs.python.org/3/library/asyncio.html`.

[8] Python Software Foundation. *json — JSON encoder and decoder.* [Online; accessed Jan-2023]. URL: `https://docs.python.org/3/library/json.html`.

[9] G. v. Rossum I. Katriel Y. Selivanov. *PEP 654 - Exception Groups and except*.* [Online; accessed Jan-2023]. 2022. URL: `https://peps.python.org/pep-0654/`.

[10] M. Jarolímek. “Plz009 Specifikace systému”. Confidential company document. 2022.

[11] J. Kohout. “Testy zapojení - popis periferií”. Confidential company document. 2022.

[12] J. Kuptík. “Automatické testování PLC SW”. Confidential company document. 2022.

[13] J. Kuptík. “G3 PLC control system specifications”. Confidential company document. 2021.

[14] S.F. Lott and D. Phillips. *Python Object-Oriented Programming: Build Robust and Maintainable Object-Oriented Python Applications and Libraries, 4th Edition.* Packt Publishing, Limited, 2021. ISBN: 9781801077262.

[15] H. Percival and B. Gregory. *Architecture Patterns with Python: Enabling Test-driven Development, Domain-driven Design, and Event-driven Microservices.* O’Reilly, 2020. ISBN: 9781492052203. URL: `https://books.google.cz/books?id=rsizyAEACAAJ`.

[16] opcua-asyncio library development team. *Python opcua-asyncio Documentation.* [Online; accessed Jan-2023]. URL: `https://opcua-asyncio.readthedocs.io/en/latest/index.html`.