A formal toolchain for model-based testing of PLC systems

Gaadha Chariyarupadannayil Sudheerbabu 1900645
Master's degree Programme in Information Technology, Computer Engineering
Supervisors: Dragos Truscan, Tanwir Ahmad
Faculty of Science and Engineering
Åbo Akademi University
2021

Abstract

Programmable logic controllers (PLCs) are programmable controller devices broadly used for industrial automation in industrial control systems. The correct functioning of such systems using PLCs is of utmost importance since their failure can result in financial losses and even human life losses. For the verification and validation of PLC systems, there is an increasing demand for test automation solutions. Early verification of the system's functional and safety features based on requirements can prevent malfunctions and ensure software quality. Besides, this enhances the feedback mechanisms during the software's development and operation phases.

Model-based testing (MBT) is a black-box testing approach that generates abstract tests automatically from a behavioral model describing the expected behavior of the system. Such model can be typically built from the requirements of the system. The quality and performance evaluation of the MBT while testing the robustness of complex systems is an area under active research.

This thesis proposes a model-based testing toolchain approach for modeling, test case generation, and test execution for safety-critical PLC systems. The proposed methodology considers the system under test as a black-box, and the behavioral model of the system is built based on the requirements specification. The model is created using UPPAAL timed automata and its properties are verified via model checking in the UPPAAL tool via Timed Computation Tree Logic (TCTL) queries. The UPPAAL tool is then used for test generation either via model checking or via structural coverage. The generated tests are executed using the Pytest testing framework against the PLC application program.

In this work, a machinery control system implemented using the IEC 61131-3's Function Block Diagrams (FBD) programming language is used as the case study. Pytest testing framework uses OPC UA communication protocol to connect to the PLC application running on a simulated device in the CODESYS development environment. The results of the study show that the proposed toolchain can be an efficient solution for the verification of safety-critical PLC systems in the industry.

Keywords: Model-based Testing; Model checking; Programmable Logic Controllers; PLC; IEC 61131-3; IEC 61508; UPPAAL; OPC UA; Function block diagram; FBD.

Acknowledgements

I would like to wholeheartedly thank my supervisor, Adjunct Professor Dragos Truscan, for the guidance, encouragement, and continuous support. I express my sincere gratitude to Professor Ivan Porres Paltor and Dragos Truscan for the opportunity to do the thesis work as part of the VeriDevOps team.

Thanks to my labmates, Tanwir Ahmad, Junaid Iqbal, and Srijan Chapagain, for their cooperation. I would like to thank Shameena V.M, Sargamates, Shahariya R.P, Induja M.J, Farzana Tajuddin, Imran Ahmad Shahid for your love and friendship. I fondly thank all the friendly and smiling faces at third-floor Agora for making the workplace wonderful.

Big thanks to my dear parents, Sudheerbabu and Indira, for the love and prayers. An ocean of love and gratitude to my partner Fuaad and our little bundle of joy, Alia.

Contents

List of Figures

List of Tables

1. Introduction

Industrial control systems (ICS) are increasingly present in many application domains, such as as manufacturing, electric power generation, chemical manufacturing, oil refineries, and water and wastewater treatment. An industrial control system consists of different types of control systems and associated instrumentation such as devices, systems, networks, controls working together to operate or automate industrial processes [1, 2]. An ICS includes control systems such as Programmable Logic Controllers (PLCs), distributed control systems (DCS), and supervisory control and data acquisition systems (SCADA). PLCs are controller devices that collects information from sensors and input commands and control a process based on a program logic execution [3]. PLCs are used in complex industrial control systems such as nuclear power plants, heavy machinery equipment control, system monitoring, logistics, energy research, rail automation, etc. Such systems should not only implement their functions correctly, but also should be safe to use and secure.

The gradual shift of ICS systems from physical to cyber-physical systems introduces potential security threats, emphasizing the importance of security testing. There has been incidents reported about cyber-attacks on nuclear power plants (NPP) such as emergency shutdown of the Brown Ferry NPP in 2006 [4], Hatch NPP in 2008 [5] , and the Stuxnet worm attack on the Natanz nuclear facility in 2010 [6]. Table 1.1 summarizes the details and causal factors of the aforementioned cyber-attacks targeted on PLC systems to exploit their vulnerabilities and trigger a system malfunction.

An investigative study by Lim, Bernard, et al. [7] demonstrated the impacts of a cyber-attack on a Tricon PLC system of a nuclear power plant. Their research revealed possible ways to trigger an attack and exploit the vulnerabilities of the Tricon PLC that uses a Triple-Modular Redundant (TMR) architecture. The findings from the study shows that using the types of attack: (i) Latent Failure Attack, and (ii) Immediate Failure attack, the control logic of Tricon system can be altered resulting in a common-mode failure. To prevent future cyber-attacks on PLC systems, developing strategies for verifying and validating their functional safety is becoming more important.

Safety-critical software systems need to be verified and validated meticulously for

Incident	Details and Causal factors
Browns Ferry NPP	The failure of two water recirculation pumps caused an emergency manual shutdown in the Browns Ferry NPP in 2006. The failure was caused by a spike in network data traffic due to a malfunctioning PLC. The pattern of attack was similar to a denial of service cyber-attack.
Hatch NPP	A software update to a computer on the NPP's network caused Unit 2 of the Hatch NPP to automatically shutdown. While rebooting the computer collecting the diagnostic data from the process control network, the data on the control network was reset by the synchronization program. This data reset was misinterpreted as a sudden drop in the reactor's water reservoirs, causing an automatic shutdown. A failure in understanding the dependencies between network devices triggered a cyber-attack in this case.
Natanz NPP	Stuxnet worm attack targeted a SIEMEN's Step 7 PLC (used to configure the PLC) and hijacked the Step 7 configuration software by replacing one of the dynamic link library used by it.

Table 1.1: Details of cyber-attacks on nuclear power plants

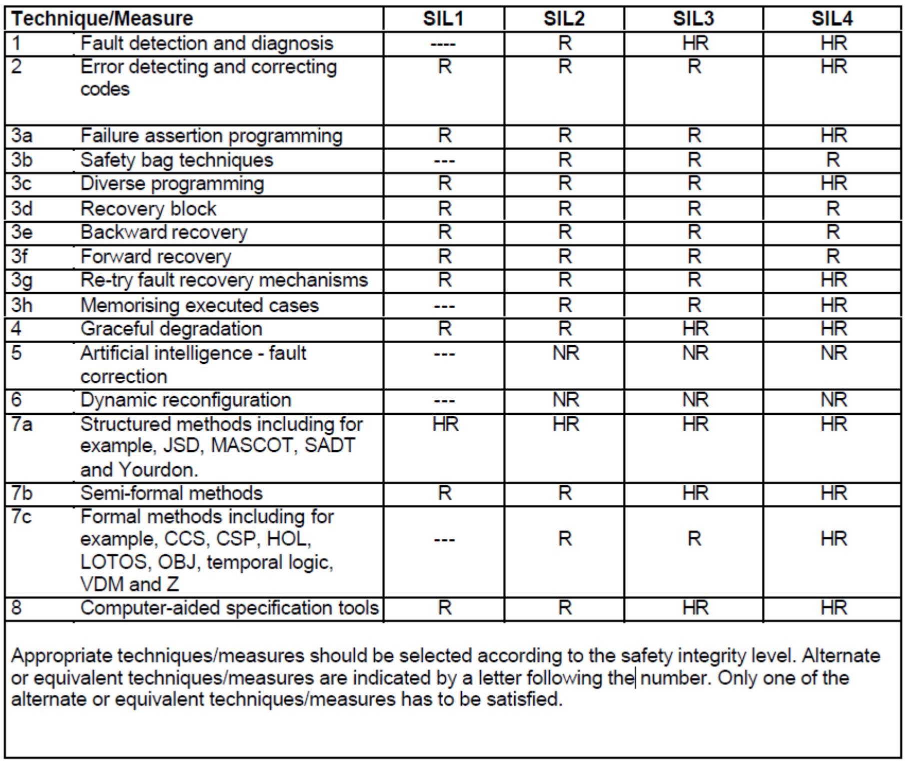
their functional and safety features to prevent malfunctions [8] and there is an increasing demand for test automation tools for the verification of safety-critical PLC applications [9].

Most PLC manufactures in Europe are members of *PLCopen*, an independent organization providing efficiency in industrial automation [10]. *PLCopen* defines IEC 61131-3 as a worldwide standard for PLC hardware and the PLC programming systems [11]. IEC 61508 is an international safety standard for the design of safety related systems implemented using electrical / electronic / programmable electronic devices [12]. The compliance requirements of IEC 61131-3 standardisation keep development and implementation costs of PLC systems under control. The standard consists of seven parts, and part 6 deals with safety-related PLC which aims to adapt the requirements of safety standard IEC 61508 to PLCs.

IEC 61508-3 defines safety integrity levels (SILs) for software in safety-related systems to avoid systemic failures. Figure 1.1 shows the IEC 61508-3 recommendations to avoid the introduction of faults in software design and development phase.

An example of safety-related PLC is a machinery control system which is required to operate its machine unit in a continuous mode maintaining its safe state. In safety-related systems the safety requirements should be translated to SILs in its final stage of development Figure 1.2.

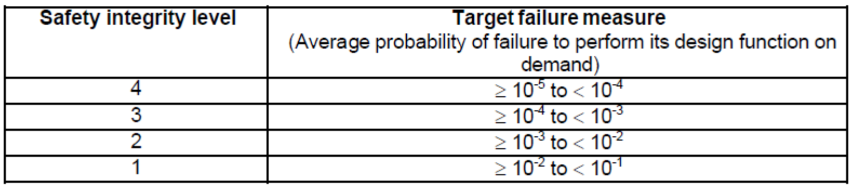
A study on software safety process patterns by Vuori et al. [13] states that IEC 61508-



Technique/Measure		SIL1	SIL2	SIL3	SIL4
1	Fault detection and diagnosis	----	R	HR	HR
2	Error detecting and correcting codes	R	R	R	HR
3a	Failure assertion programming	R	R	R	HR
3b	Safety bag techniques	---	R	R	R
3c	Diverse programming	R	R	R	HR
3d	Recovery block	R	R	R	R
3e	Backward recovery	R	R	R	R
3f	Forward recovery	R	R	R	R
3g	Re-try fault recovery mechanisms	R	R	R	HR
3h	Memorising executed cases	---	R	R	HR
4	Graceful degradation	R	R	HR	HR
5	Artificial intelligence - fault correction	---	NR	NR	NR
6	Dynamic reconfiguration	---	NR	NR	NR
7a	Structured methods including for example, JSD, MASCOT, SADT and Yourdon.	HR	HR	HR	HR
7b	Semi-formal methods	R	R	HR	HR
7c	Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z	---	R	R	HR
8	Computer-aided specification tools	R	R	HR	HR

Appropriate techniques/measures should be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.

Figure 1.1: Recommendations to avoid introduction of faults during software design and development (IEC 61508-3) [12]

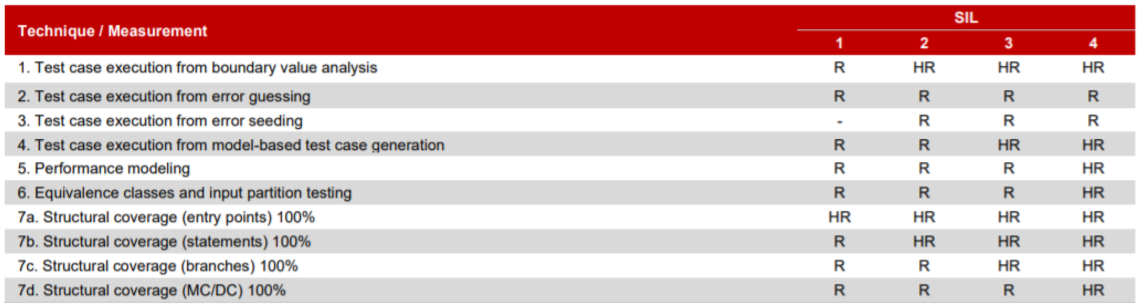


Safety integrity level	**Target failure measure** (Average probability of failure to perform its design function on demand)
4	$\geq 10^{-5}$ to $< 10^{-4}$
3	$\geq 10^{-4}$ to $< 10^{-3}$
2	$\geq 10^{-3}$ to $< 10^{-2}$
1	$\geq 10^{-2}$ to $< 10^{-1}$

Figure 1.2: Safety integrity levels for safety functions operating in the low demand mode of operation [12]

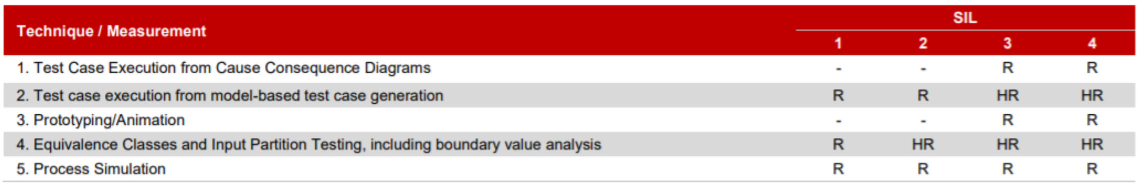
3 (2nd Edition) recommends model-based testing during software design and development to ensure software safety. IEC 61508-3:2010, Part3 [14] recommends as *recommended (R)* or *highly recommended (HR)*, the test execution from model-based test case generation for verification and validation of safety-related software systems based on the SILs (Figure 1.3).

MBT is a promising black-box testing technique that can reveal whether the system's actual implementation conforms to the requirement specification and ensure its operational correctness. It aims to generate tests automatically from abstract behavioral models of the system [15]. It relies on creating a behavioral model of the software system and the behaviour of the environment from its requirements. Recent research in model-based testing strives to develop methods and tools to fine-tune MBT for domain-specific use in



Technique / Measurement	SIL			
	1	2	3	4
1. Test case execution from boundary value analysis	R	HR	HR	HR
2. Test case execution from error guessing	R	R	R	R
3. Test case execution from error seeding	-	R	R	R
4. Test case execution from model-based test case generation	R	R	HR	HR
5. Performance modeling	R	R	R	HR
6. Equivalence classes and input partition testing	R	R	R	HR
7a. Structural coverage (entry points) 100%	HR	HR	HR	HR
7b. Structural coverage (statements) 100%	R	HR	HR	HR
7c. Structural coverage (branches) 100%	R	R	HR	HR
7d. Structural coverage (MC/DC) 100%	R	R	R	HR

(a) Dynamic analysis and testing: IEC 61508-3, Part3, Annex B Table B.2



Technique / Measurement	SIL			
	1	2	3	4
1. Test Case Execution from Cause Consequence Diagrams	-	-	R	R
2. Test case execution from model-based test case generation	R	R	HR	HR
3. Prototyping/Animation	-	-	R	R
4. Equivalence Classes and Input Partition Testing, including boundary value analysis	R	HR	HR	HR
5. Process Simulation	R	R	R	R

(b) Functional and black-box testing: IEC 61508-3, Part3, Annex B Table B.3

Figure 1.3: Verification and validation of software under IEC 61508-3:2010 [14]

industrial applications.

By having a test model created from requirements, MBT allows one to detect early errors and inconsistencies in the specifications. And this is beneficial because, as stated in [16], most common errors in software systems stem from misunderstanding or misinterpretation of software specifications, often resulting in incorrect implementations. Thus there is a need for formal verification of the specs / models via model checking to increase their quality by detecting faults early during design and development phase.

The evolution of ICS systems from physical systems into cyber-physical systems introduces potential security risks and emphasises the need for their functional safety and security testing. ICS systems should undergo thorough testing to confirm their operational correctness. This thesis proposes a model-based testing toolchain approach for modeling, test case generation, and test execution for safety-critical PLC systems. The proposed methodology considers the system under test as a black-box, and the behavioral model of the system is built based on the requirements specification. The model is created using UPPAAL timed automata [17] and its properties are verified via model checking in the UPPAAL tool via Timed Computation Tree Logic (TCTL) queries. The UPPAAL tool is then used for test generation either via model checking or via structural coverage. The generated tests are executed using the Pytest testing framework [18] against the PLC application program.

1.1 Research questions

Model-based testing allows increasing the quality of PLC implementations. The objective of this thesis is to investigate the benefits a formal toolchain for functional and safety testing of PLC systems using MBT techniques. Throughout this work, we attempt to answer the following research questions:

- (RQ1) How to apply verification effectively using model checking of the behavioral specifications for industrial control systems?
- (RQ2) How to generate automated tests and execute offline testing for PLC FBDs using the UPPAAL model checker?
- (RQ3) How does the toolchain perform in terms of quantitative measures such as test suite generation time and memory usage in the MBT process?

1.2 Research contributions

Recent research conducted in model-based testing techniques has revealed the advantages of automated test generation using model checking [9, 19]. In automated test generation input values for a test is generated dynamically from an abstract behavioral model of the system and embedded into executable test scripts. Automated test generation is more effective and efficient in terms of time, cost, and maintenance compared to manual testing. The challenge is to design test automation strategies and tools based on model-based methodologies and coverage measurement techniques.

This thesis proposes a toolchain for model checking and model-based test generation using the UPPAAL tool for PLC systems written in FBDs. The proposed toolchain employs UPPAAL Timed Automata formalism to model the expected behavior of the software system, UPPAAL-Yggdrasil tool of UPPAAL for automated test generation, OPC UA (Open Platform Communications Unified Architecture) [20] as communication protocol for data exchange between PLC system and Python, and Python testing framework Pytest for test execution (Figure 1.4).

The test suite comprising the automatically generated tests from the behavioral model created using the UPPAAL tool will be executed using the Pytest testing framework against the PLC application under test. The behavioral model is designed based on the requirements specification that specifies the expected output for each input combinations. The test code has validation checks for conformance checking of expected output behavior of SUT to the requirements. The test report from Pytest summarizes each test case's pass/fail status, and a coverage measurement add-on named CODESYS Profiler [21] will be used to measure the code coverage achieved by the generated test suite.

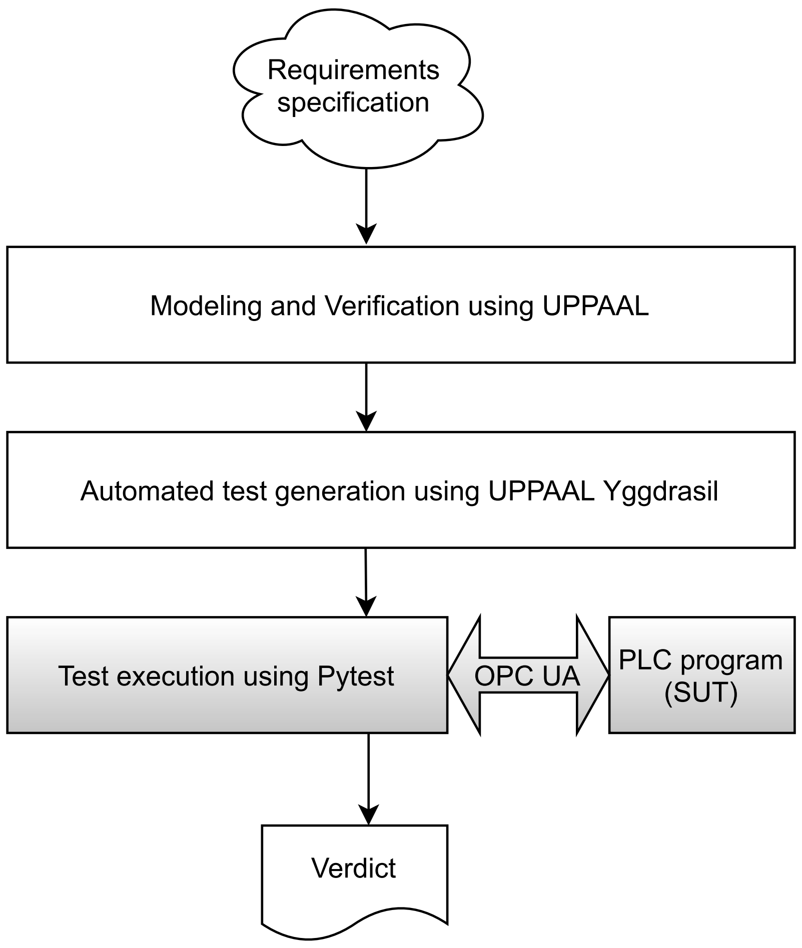

Requirements specification
Modeling and Verification using UPPAAL
Automated test generation using UPPAAL Yggdrasil
Test execution using Pytest
OPC UA
PLC program (SUT)
Verdict

Figure 1.4: Proposed toolchain approach for MBT and verification of PLC systems

1.3 Structure of the thesis

The rest of the thesis is divided into five chapters. Chapter 2 introduces the concept of software testing, model-based testing, PLC, FBDs and the tools UPPAAL, CODESYS, OPC UA, and Pytest. The related works published on model checking of PLC systems and model-based testing applied to industrial case studies are discussed in Chapter 3. Chapter 4 introduces the design and implementation strategy for the proposed toolchain. The case study results and the findings from the verification, test generation, test execution and coverage measurements are presented in Chapter 5. Chapter 6 summarizes the work done, recall the research questions, and discuss the future work.

2. Background

The concepts of PLC, FBD, software testing, model-based testing, formal verification using model checking, and the tools UPPAAL, CODESYS, OPC UA, and Pytest are introduced in this chapter.

2.1 Programmable Logic Controllers

PLCs are programmable controller devices widely used for automation purposes in ICS systems due to their simultaneous input/output processing capacity. A PLC system comprises the components: processing unit, memory, input/output interface, communication interface, and the programming device [22]. The processing unit in PLC (Figure 2.1) interprets the input signals, carries out control actions issued by the program in the memory, and communicates the decisions as action signals to the outputs. The memory unit stores the programs and the input/output data. The communications interface transfers the data on communication networks among the connected PLC devices. A program organization unit (POU) in a PLC application periodically repeats its execution like a cyclic task.

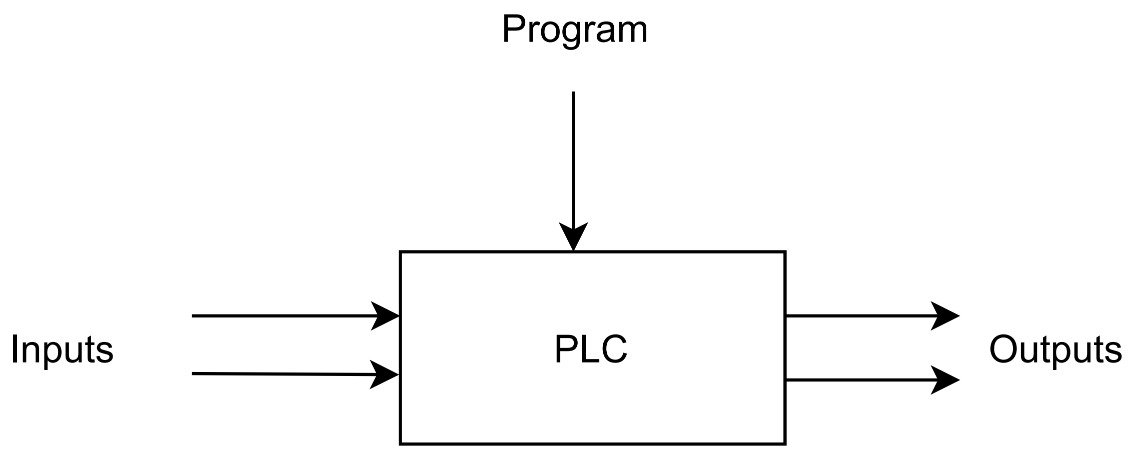


Figure 2.1: A Programmable Logic Controller

The IEC 61131-3 standard followed by most of the PLC manufactures includes the programming languages [23] :

- Instruction List (IL)
- Structured Text (ST)
- Ladder Diagram (LD)
- Function Block Diagram (FBD)
- Sequential Function Chart (SFC)

IL and ST are text-based languages, while LD, FBD, and SFC are graphical programming languages. The standard suggests improvements to the programming languages in aspects such as addressing, execution, data structures, sequential control, use of symbols, and connection between languages. It also promotes the reuse of the program code.

2.1.1 Function block diagrams

Function Block Diagram (FBD) is a graphical modeling PLC programming language widely used in developing safety-critical industrial systems [23]. An FBD program is composed of interconnected blocks and follows a hierarchical software architecture. FBD uses graphical objects such as connections, graphical elements for execution control, graphical elements to call a function or a function block, and connectors. It uses standard logical functions such as AND, OR, NOT, in a graphical form. Figure 2.2 presents an example of the implementation of an XOR with an AND and OR.

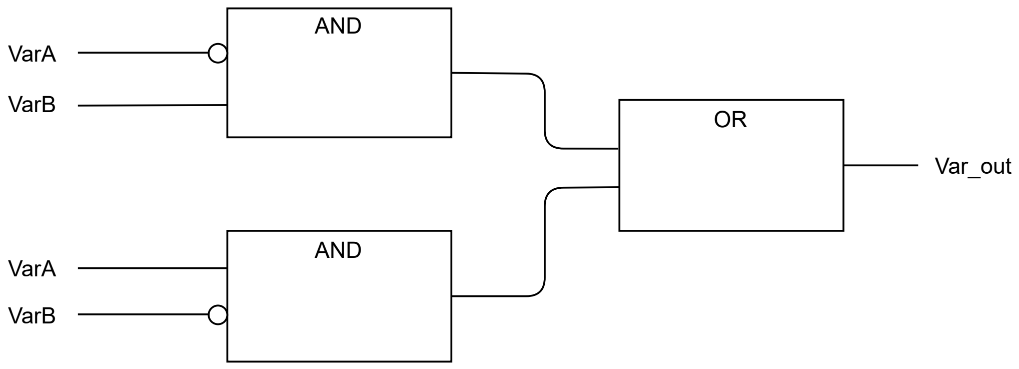


Figure 2.2: Implementation of an XOR with an AND and OR

In an FBD program, the code is divided into networks containing a logical or arithmetic expression, calls of other programs or functions or function blocks, jumps or return conditions. These networks form a program that is executed sequentially by the PLC. When a block in an FBD program is activated, it reads the input variables, executes, and writes the outputs.

IEC 61131-3 defines five groups of standard function blocks: bistable elements, edge detection, counters, timers, and communication function blocks [11]. Table 2.1 gives a list of all the standard function blocks available in bistable elements, edge detection, counters, and timers.

Name of FB with input parameter names	**Names of output parameters**	**Short description**
Bistable elements		
SR (S1,R,	Q1)	Set dominant
RS (S,R1,	Q1)	Reset dominant
Edge detection		
R_TRIG { ->} (CLK,	Q)	Rising edge detection
F_TRIG { -<} (CLK,	Q)	Falling edge detection
Counters		
CTU (CU, R, PV,	Q, CV)	Up counter
CTD (CD, LD, PV,	Q, CV)	Down counter
CTUD (CU, CD, R, LD, PV,	Q, CV)	Up/down counter
Timers		
TP (IN, PT	Q, ET)	Pulse
TON (IN, PT	Q, ET)	On-delay
TOF (IN, PT	Q, ET)	Off-delay

Table 2.1: List of standard function blocks [11]

The table lists the names of the input and output variables. Their name, along with the meaning and the elementary data types are listed in Table 2.2.

Bistable elements(Flip-flops): The bistable function blocks SR and RS (Figure 2.3) implement dominant setting and resetting.

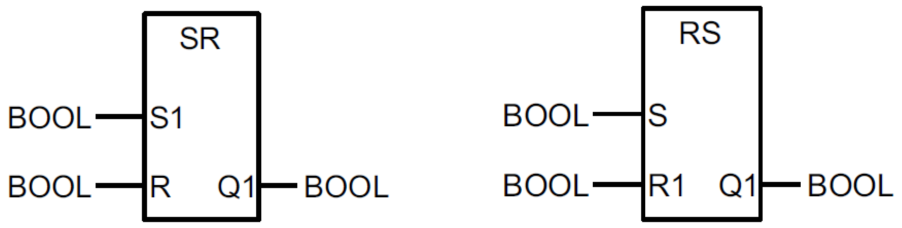


Figure 2.3: Graphical declarations of the bistable function blocks SR and RS [11]

For $SR, Q1 := S1\ OR\ (NOT\ R\ AND\ Q1)$;

For $RS, Q1 := NOT\ R1\ AND\ (S\ OR\ Q1)$.

Edge Detection: The function blocks for edge detection R_TRIG and F_TRIG (Figure 2.4) detect an edge on the first call, if the input of R_TRIG is true or or the input of F_TRIG is false.

Inputs / Outputs	**Meaning**	**Data Type**
R	Reset input	BOOL
S	Set input	BOOL
R1	Reset dominant	BOOL
S1	Set dominant	BOOL
Q	Output (standard)	BOOL
Q1	Output (flip-flops only)	BOOL
CLK	Clock	BOOL
CU	Input for counting up	R_EDGE
CD	Input for counting down	R_EDGE
LD	Load (counter) value	INT
PV	Pre-set (counter) value	INT
QD	Output (down counter)	BOOL
QU	Output (up counter)	BOOL
CV	Current (counter) value	INT
IN	Input (timer)	BOOL
PT	Pre-set time value	TIME
ET	End time output	TIME
PDT	Pre-set date and time value	DT
CDT	Current date and time	DT

Table 2.2: Abbreviations and meaning of the input and output variables in Table 2.1 [11]

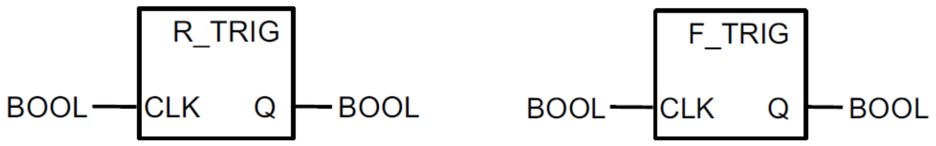


Figure 2.4: Graphical declarations of the function blocks R_TRIG and F_TRIG [11]

Counters: The counter inputs CU and CD are of the data type BOOL and have an additional attribute R_EDGE that represents a rising edge to be recognised in order to count up or down (Figure 2.5).

Timers: The function block TP (Figure 2.6) acts as a pulse generator which supplies a pulse of constant length at output Q when a rising edge is detected at input N. The output ET shows the time that has elapsed. The on-delay timer TON supplies the input value IN at Q with a time delay when a rising edge is detected at IN. if input IN is "1" only for a short pulse, the timer is not started for this edge. The off-delay timer TOF performs the inverse function to TON. It delays a falling edge in the same way as TON delays a rising one.

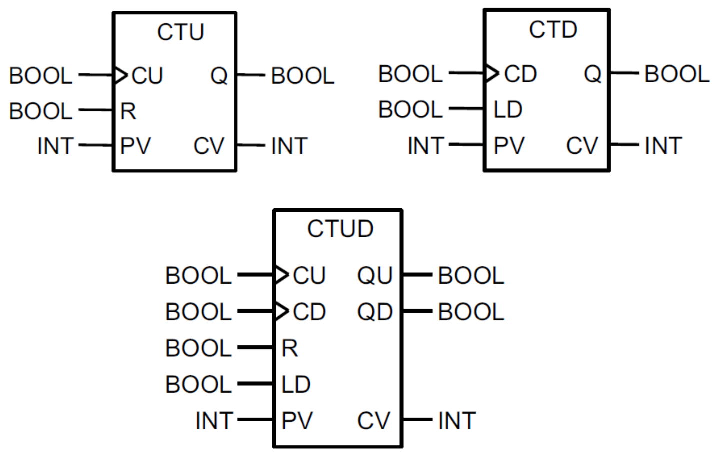


Figure 2.5: Graphical declarations of the function blocks CTU, CTD, and CTUD [11]

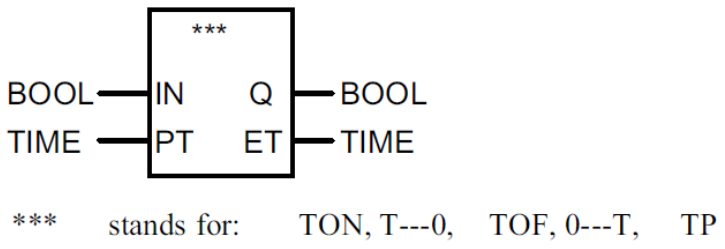


Figure 2.6: Graphical declarations of the function blocks TON, TOF, and TP [11]

2.2 CODESYS

CODESYS is an integrated development environment developed by Smart Software Solutions GmbH for programming PLC systems. CODESYS follows the IEC 61131-1 standard defined for the PLC programming languages IL, ST, LD, FBD, and SFC (Refer section 2.1). In this thesis, CODESYS version 3.5 is used as it has a built-in software PLC (softPLC). A softPLC is a built-in simulator available in CODESYS to run, monitor, and debug a PLC program. CODESYS V3.5 SP16 supports the OPC UA features and allows to set up a communication link between CODESYS softPLC and Pytest for test automation.

A standard project in CODESYS V3.5 has a program organization unit (POU) with the name PLC_PRG, a cyclic task where PLC_PRG is called every 200 ms, and the references to latest available libraries [23]. In this work, the PLC application runs on a simulated device, CODESYS Control Win V3, which allows testing the application's functionality without PLC hardware.

2.2.1 CODESYS Profiler

CODESYS Profiler is an add-on solution that comes with CODESYS Professional Developer Edition [21]. It is used for measurement and evaluation of the processing times and code coverage of program organization units in an IEC 61131-3 (Refer section 2.1) application. The code coverage measurement during testing can be performed on the CODESYS softPLC and the tool also shows number of statements covered and statements which were not covered during test execution.

2.3 Open Platform Communications Unified Architecture

Open Platform Communications Unified Architecture (OPC UA) [20] is a data exchange protocol standard released by the OPC foundation for industrial automation. The basic OPC UA specification consists of the following parts: [UA1] provides an overview of the security model, [UA2] specifies the security model assessment, [UA3] defines the address space model, [UA4] defines the services, [UA5] defines the information model, [UA6] the mapping of the services to a concrete technology [20]. The different profiles for OPC UA clients and servers are specified in [UA7], [UA8] covers specializations for data access, [UA9] covers alarms and conditions, [UA11] covers historical access, and specializations for programs are covered in [UA10] [20]. Figure 2.7 shows OPC UA security architecture.

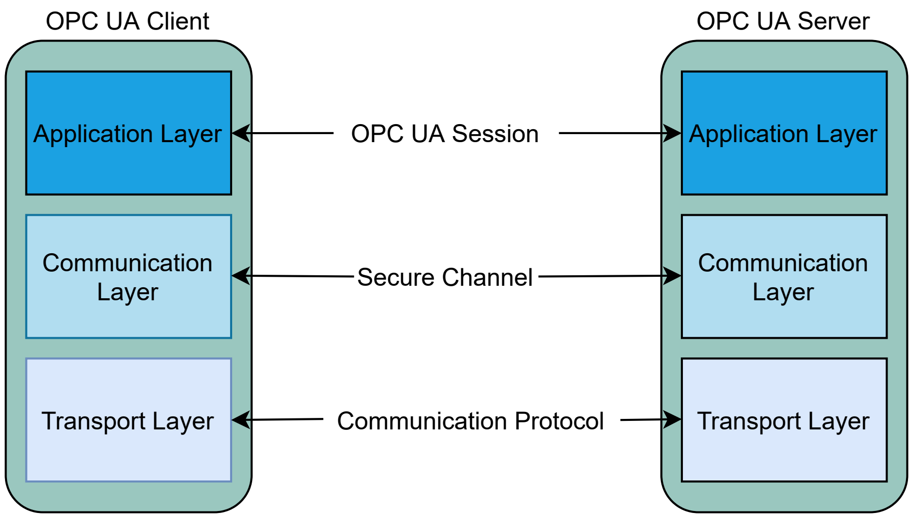


Figure 2.7: OPC UA Security Architecture [20]

OPC follows a client-server software architecture for data exchange. OPC UA standard announced in 2006, uses one address space for all different OPC client-server architectures [24]. This standard addresses the security goals such as authentication, authorization, confidentiality, integrity, auditability, and availability to maintain security of industrial automation systems.

Authentication and User Authorization are assigned to the application layer, Application authentication verified by the communication layer, and confidentiality and integrity maintained by the transport layer. The layered architecture and modular design allow module-level updates to maintain the application's security without redesigning the whole application. OPC UA data exchange between client and server supports robust, secure communication and resists attacks.

2.3.1 CODESYS OPC UA Server

CODESYS OPC UA server [25] is included in the CODESYS control SoftPLC systems and allows communication with any OPC UA client. The CODESYS OPC UA supports the following features [26]:

- Browsing of the data types and variables.
- Standard read or write functions.
- Encrypted communication according to OPC UA standard.
- Imaging of the IEC application according to "OPC UA Information model for IEC 61131-3".
- Sending of events according to the OPC UA standard.

2.3.2 Python OPC UA Client

Python OPC UA [27] is a Python library which provides an interface to send and receive UA defined structures over OPC UA protocol. Python OPC UA client can connect to the CODESYS OPC UA server and browse the address space containing the input and output variables in the PLC program.

2.4 Software Testing

Software testing is an essential validation and verification process used to reveal faults and ensure the software systems' quality. The reliability and quality of software can be compromised due to bugs in the implementation. As per Amman et al. [28] software fault, error, and failure are defined as follows:

Software Fault: A static defect in the software

Software Error: An incorrect internal state that is the manifestation of some fault

Software Failure: External, incorrect behaviour with respect to the requirements or another description of the expected behavior.

Software testing can be performed at different development phases, and tests can be generated from requirements specification, design, or source code. The "V model" of software testing depicts the testing levels and the corresponding software development phase (Figure 2.8). The most cost-effective way of testing is to identify the defects at an early stage of software development.

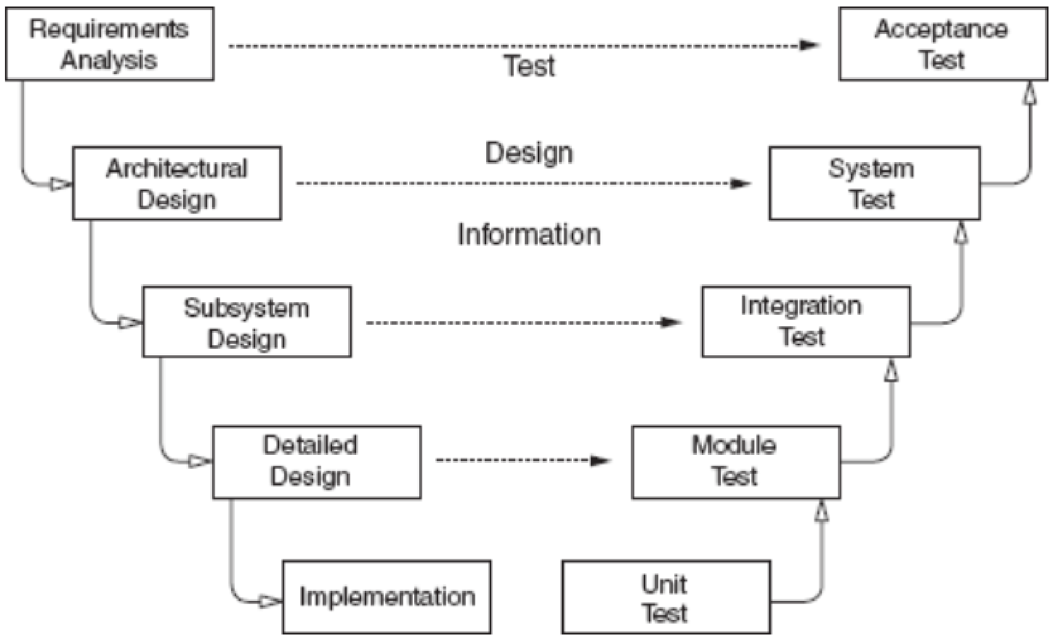


Figure 2.8: The Software Development Life Cycle and testing levels "V model" [28].

- The requirements analysis phase captures the user requirements for implementing the software. Acceptance testing decides whether or not the implementation meets the specified requirements.

- The architectural design determines components and connectors that comprise the system as per the specified requirements. System testing decides if the components work well and the system as a whole meets the specifications.

- The subsystem design defines the behavior of the subsystem in the architecture of the system. Integration testing checks if the interfaces between the modules in a subsystem work correctly.

- The detailed design specifies the structure and behavior of each module in the system. Module testing probes if the individual module, the modules' components, and the data structures work correctly.

- The implementation phase is when the actual code of the system is implemented. Unit testing is the lowest level of testing and checks the correctness of each unit in the implementation.

2.4.1 Model-based testing

Model-based testing (MBT) is a testing approach to systematically generate tests from a model of the system and its environment [29]. Model-based testing has three steps: modeling, test generation, and test execution.

Tests are generated from an abstract behavioral model of the system, and one execution trace of the model is considered a test case. The test quality can be maintained by applying test selection criteria while generating test cases from the model and having good quality models used for test generation. The generated test cases can then be executed against the system under test to verify that the implemented system conforms to the expected behavior specified by the model. The test execution process in MBT can be either offline or online. In this work, offline testing is done where the test cases are generated before the test execution. It provides the advantage that test generation and test execution can be done in different environments.

Utting et al. [15] describe the process of MBT as the following five steps:

1. The model of the system under test referred to as the test model is created from the requirements. The test model is built at a more abstract level than the SUT. The test model is validated for consistency and completeness to ensure the derivation of meaningful test cases from the model.

2. The test selection criteria are chosen for automatic test generation to generate a quality test suite.

3. A test specification, a high-level description of a test case, is then generated from the test selection criteria.

4. A set of test cases is generated satisfying all the test case specifications. The optimization of the test suite depends on the test generator.

5. The test cases are executed on the SUT as a test script, using an adapter that sets up the interface to the SUT. The adapter component performs the concretization of test inputs and abstraction of test outputs. The test script execution generates the verdict of the test.

Based on the above five steps mentioned (Figure 2.9), Utting et al. [15] define the taxonomy of MBT approaches as six dimensions.

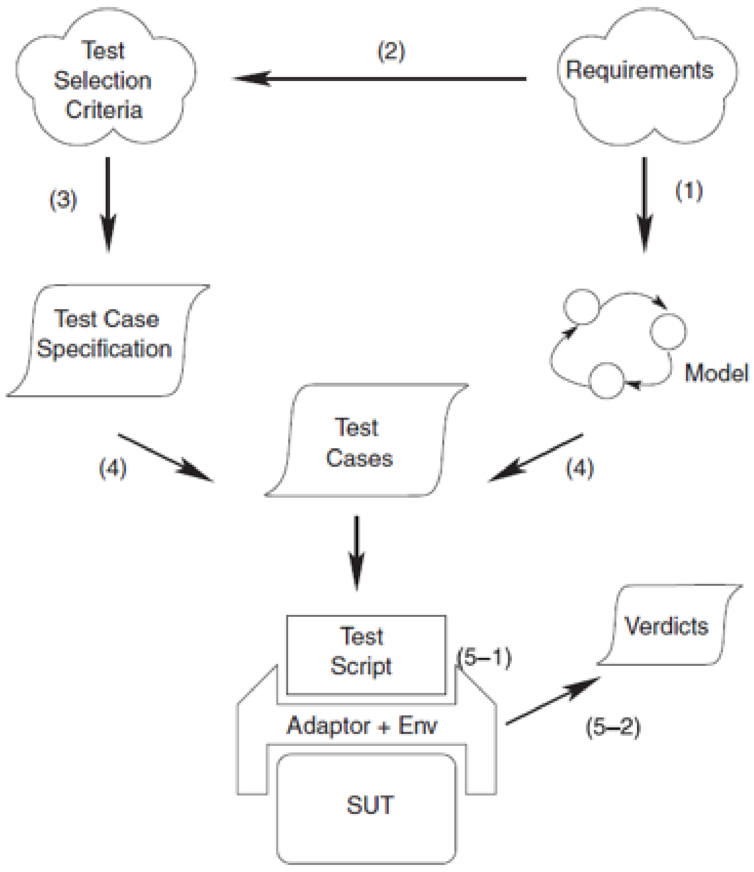


Figure 2.9: The Model-based testing process [15].

Building the model corresponds to the three dimensions scope, characteristics, and modelling paradigm. The test selection criteria and technology used for generating the tests are defined in the test generation phase. The test execution dimension deals with the relative timing of test generation and test execution. Test execution can be done either online/offline. In online testing, the test generation and test execution are done in parallel. In the offline testing approach, the test generation is done before test execution. While testing real-time systems that consume more time for test generation, the offline testing method provides the advantage of generating tests once and reusing them multiple times for testing the system. Figure 2.10 shows the taxonomy of MBT approach.

Testing complex systems can be time-consuming, and it is not possible to cover all the test scenarios. The advantages of using environment models in MBT of complex systems are test oracle creation, automated and optimal test generation, reducing the size of the state space, early validation of requirements, and re-usability [30]. Environment models can be applied for safety, robustness, and regression testing and at all testing levels: unit, system, and integration.

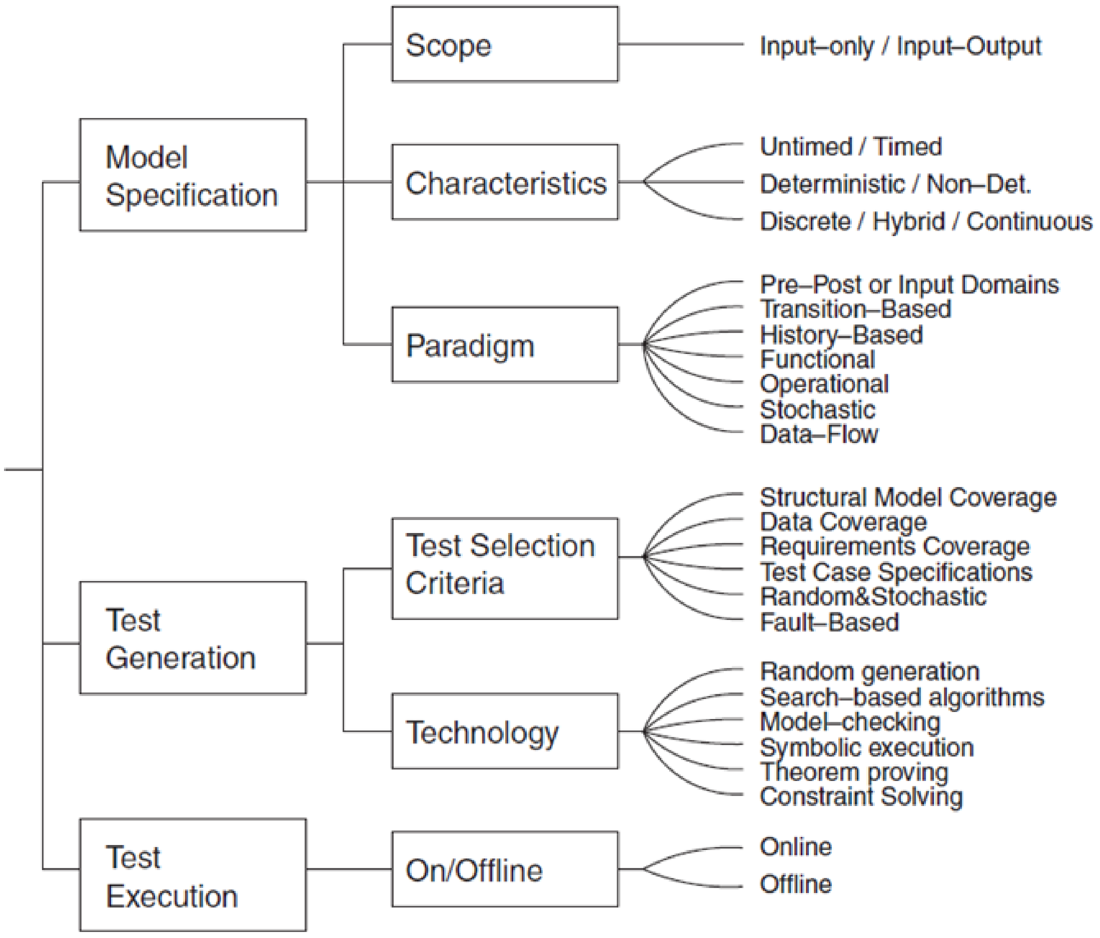


Figure 2.10: The overview of the taxonomy of model-based testing process [15].

2.4.2 Related work

In this section, the related works published on model checking, model-based testing applied on industrial case studies, and the testing of PLC FBD programs are discussed.

2.4.2.1 Verification and validation of PLC systems

The capability of the PLC system to process a high number of I/O operations simultaneously conforming to real-time constraints makes it ideal for application in domains like railway systems [19, 31], avionics [32], manufacturing conveyors [33], and nuclear power plants [34, 35]. The benefits of adopting model-based testing techniques to verify and validate PLC systems have been investigated in several studies [36, 37, 38] . A study conducted on the use of the model-based testing for conformance testing of PLC systems stated that conformance relation based on the minimum duration of a test step in the PLC execution cycle reduced time and cost of testing and performed well in fault detection [36]. Bochot et al. [32] used Lustre specification language and Luster model checker for the model checking of Airbus's ground spoiler controller function. Ferrari et al. [39] conducted a study on General Electric Transportation System's Automatic Train Protection system transforming Simulink programs to NuSMV models.

2.4.2.2 Model checking tools for PLC systems

Several researchers have used UPPAAL-based model checking to verify and test ICSs implemented using PLC software and a systematic literature review can be found in [9]. Enoiu et al. [19, 40] conducted a study on transforming PLC FBD programs into timed automata models, model checking, and automatically generating test suites using the UPPAAL tool. They defined an FBD program as a tuple consisting of Functional Elements (FE) defined as Function Blocks (FB) and Functions (FUNC), V the variables set, P the parameters set, and Con the set of connectors between the functional elements. A set of transformation rules is defined to construct a timed automata model that follows read-write-execute semantics from FBD programs. The timed automata are created with system declaration and templates corresponding to FE instantiations along with an environment model to interact with the processes. The FBD program is executed and the time for completion from reading inputs to writing outputs is noted and used to define clock variables in the timed automata model. The execution order, functional, and timing behavior of functional elements in the FBD program is mapped to the timed automata model. Using a PLC scan cycle, the test suite is generated as test sequences separated by resets. A reset transition is added to reset the variables and parameters to their default value. The test coverage achieved in the generated test suite is measured as function coverage, decision coverage, and condition coverage. The study was conducted on a train control and management system used in the railway industry and the approach proved effective for generating test suites for functional testing of PLC FBD programs. The toolbox, named COMPLETETEST, can transform FBDs to Timed Automata and generates test cases designed for logic coverage of FBD programs.The tool takes the input variables into account and generates the test cases to satisfy coverage criteria. However, the expected outputs should be entered manually to the toolbox.

Adiego et al. [41] propose a methodology for transforming PLC programs written in ST language to formal models using NuSMV [42], UPPAAL model checker, and the BIP framework [43]. The study discusses the challenges in building the correct formal model of PLC programs and verifying the safety and liveness properties. The proposed solution is to transform the ST code into an intermediate model (IM) and transform IM into formal models. Transforming the ST code into IM includes building an Abstract Syntax Tree (AST), representing the syntax of the PLC code, and a Control Flow Graph (CFG), which represents the semantics of the PLC code. To address the state-space explosion problem,

Model checker	Advantages	Disadvantages
NuSMV	Better performance for verification Supports Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) for the specification properties	Lacks good simulation facilities
UPPAAL	Good simulation facilities	Supports only a subset of CTL
BIP Frame-work	Provides a language for modeling component-based systems, code generation and simulation	Verification tool supports only deadlock properties

Table 2.3: Comparison of verification tools.

they suggested reduction techniques such as Cone of Influence (COI) [44] reduction while verifying the properties and rule-based reductions. The study also compared the model checker tools used (Table 2.3).

Sacha [45] presented a modeling method for PLC controllers using UML-based state machine diagrams and verification using UPPAAL. The study also formulated a way for the automatic programming of PLC controllers. Their approach uses the UML state machine diagram to write the specification, validate and verify it against user and safety requirements and follow a formal method for defining the semantics of the specification and rules for automatic code generation.

Mokadem, Houda Bel, et al. [46] proposed a modeling and verification method for timed multi-task PLC program using the tool UPPAAL. Their case study was conducted on MSS (Mecatronic Standard System) platform from the Bosch group, and the PLC program for the system is written in LD. Their approach considered the execution of the PLC program as three phases: input reading, program execution and output writing and the cycle duration of execution called PLC scan. The PLC cycle is modeled using the UPPAAL tool as an automaton structured as a loop and a clock to measure the cycle time and used one broadcast channel to synchronize all the TON blocks. The model is verified for its safety and timing properties, and results proved that timed model checking is a useful technique for verifying PLC multi-task programs.

Soliman et al. [47] presented an approach for verification of safety critical applications based on PLCopen safety function blocks (SFBs). They developed a transformation tool to transform inputs, SFBs, connections, and execution flow in an FBD into UPPAAL TA models. A PLC programming tool that supports PLCopen-SFB-Library is used to represent the safety application in PLCopen XML format and then transformed to UPPAAL XML. The model checking is done on the UPPAAL TA model to verify and validate its properties.

2.4.2.3 Research challenges in PLC model checking

The widespread use of PLC in the development of safety-critical real-time systems emphasizes the importance of the verification process [48]. Formal methods for the verification of PLC systems is also an active area of research [48, 49]. A study conducted by Ovatman, Tolga, et al. [9] identified that the majority of publications in verification of PLC systems are in PLC FBD model checking area. According to their findings, the most common challenges in model checking of PLC systems are state-space explosion, model consistency, correctly specifying the properties to be checked, emulating the PLC execution cycle to the model checking environment, and modeling the timer functions.

2.4.3 Formal methods for software verification

Formal methods for software verification employs two techniques: theorem proving and model checking [50]. The model checking technique depends on building a model of the system and verifying if a desired property as per the specification holds in that model. There are different model checkers available for different formalisms, such as UPPAAL [17] for Timed automata, SPIN [51] for PROMELA , NuSMV [42] for Finite state machine (FSM) , ProB [52] for B, etc. In this work, UPPAAL model checker (Version 4.1.19) is used as it allows to model and model check real-time properties.

2.4.4 UPPAAL Timed Automata

UPPAAL is a model checker tool for modeling, simulation, and verification of real-time systems using an extended version of timed automata called UPPAAL timed automata [17].

A Timed Automaton is a state machine with a set of *locations*, directed edges that connect the locations to each other, actions, and clocks.

Theorem 2.4.1 (Timed Automata) *Let X be a set of non-negative real valued variables called clocks and $\mathcal{G}(X)$ be a set of guards on clocks that are conjunctions in form of $x \bowtie c$, and let $\mathcal{U}(X)$ denote the set of* updates *of clocks corresponding to sequences of statements of the form $x := c$, where $x \in X, c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, >, \geq\}$.*

A timed automation [53] over (A,X) is a tuple (L, l_0, I, E) [54] , where:

- *L is a finite set of* locations *and $l_0 \in L$ is* initial *location;*
- *$E \subseteq L \times A \times \mathcal{G}(X) \times \mathcal{U}(X) \times L$ is a set of edges including an action, a set of guards, and a set of clocks.*
- *$I : L \to \mathcal{G}(X)$ is a function, which assigns location invariants.*

UPPAAL Timed Automata is an extension of timed automata with additional features. The additional features include templates, constants, bounded integer variables, binary synchronization channels, broadcast channels, urgent synchronization channels, urgent locations, committed locations, arrays (for clocks, channels, constants, and integer variables), initialisers, record types, custom types, and user functions [55]. In UPPAAL TA, a state of a system is denoted by a set of locations, variables, and clocks.

■ **Process Declarations**

- ☐ *Templates*: UPPAAL TA has templates defined with a set of parameters of type `int, chan,` etc.
- ☐ *Constants*: A constant is declared as an integer type and cannot modify the defined value. Declaration format: `const name value`.
- ☐ *Clocks*: A clock variable can be declared as a global or a local variable. A clock variable can have a value x, where $\{x \in \mathbb{R} \mid x \geq 0\}$. In the initial state, the clock variables can have a value zero and when an automaton is waiting in a location and time elapses, the value of clock variables increase. Declaration format: `clock` $x_1,..x_n$;
- ☐ *Bounded integer variables*: This feature allows to declare integer variables with a lower and upper bound. Declaration format: `int[min,max] name`. The bounds can be applied to guards, invariants, and assignments, and the verification can perform bound checks. Violation of a bound check can lead to an invalid state that is discarded at run-time.

■ **Synchronizations**

- ☐ *Binary synchronization*: Declared as `chan c` ; a channel labelled `c!` synchronizes with `c?`
- ☐ *Broadcast channels*:Declared as `broadcast chan c`; A sender channel `c!` synchronize with the receiver `c?`
- ☐ *Urgent Synchronization*: A synchronization channel can be declared as urgent by prefixing with the keyword `urgent`. There should be no delay if the synchronization transition is activated on an urgent channel. Clock guards cannot be defined on the edges using urgent channels for synchronization.

■ **Locations**

- ☐ *Normal locations*: A normal location defined in UPPAAL can have an invariant. If an invariant is defined for a normal location, time can pass when system waits in that state/location as long as invariant is satisfied. The system should leave the location when the invariant becomes false.

- ☐ *Urgent locations*: When a location is declared `urgent`, there should be no further delay in proceeding forward from that location.
- ☐ *Committed locations*: When a location is declared `committed`, the system should move from that state without any delay by enabling a transition through an output edge from the committed location.

■ **Data types**

- ☐ *Arrays*: The clock, channels, constants, and integer variables can be declared as arrays.
- ☐ *Initialisers*: Integer variables and array integer variables can be declared using initialisers.
- ☐ *Record types*: Declared using *struct* construct like in C.
- ☐ *Custom types*: Declared using *typedef* construct like in C.

■ **Functions**

- ☐ *User functions*: Functions can be global or local to templates. A function defined locally in a template can access the parameters in the template.

In UPPAAL TA, the system's model and its environment can be defined as separate automata, called processes, working parallel. A process can be executed individually or synchronized with another process. Synchronization of two processes is done using channel synchronizations. The synchronization between processes is done using input actions denoted as "!" for emitting and output actions denoted as "?" for receiving. A timed automaton consists of locations and edges. An edge from a location to another shows the transition from one state to another in the process. Transitions can be enabled or disabled using predicates known as guards. Timing constraints for a location can be provided using clock invariants to decide the time duration for being in that location. UPPAAL supports global and local variables (local to a process) of type integer, boolean, and clock.

2.4.5 Verification using UPPAAL

The UPPAAL verification engine *verifyta* is used to verify the model for properties such as reachability, safety, deadlock-freeness, and liveliness [56]. The query language used in UPPAAL is a subset of TCTL (timed computation tree logic) [53]. It consists of state formulae to verify the states and path formulae to check safety, liveliness, and reachability properties using paths of the model. These formula are defines as follows:

1. State Formula (φ) is an expression that describes the properties of a state in a model.The deadlock freeness of a model can be checked using a special state formula. A model is said to be in a deadlock state if in that state there are no enabled outgoing transitions.

   - $A\Box$ *notdeadlock*: Verifies that for all paths in the model, there is no deadlock state.

2. Reachability properties are used to check if a certain state in a model can be reached through any path from the initial state.

   - $E\diamond\varphi$: Verifies that there exists a path from the initial state, such that φ is eventually satisfied along that path.

3. Safety properties are used to check that a system will continue to be in a safe state and never violate a property which causes it to be unsafe.

   - $A\Box\varphi$: Verifies that φ holds true in all reachable states.
   - $E\Box\varphi$: Verifies that there should exist a maximal path such that φ is always true.

4. Liveliness properties of a system can be verified using a path formula to prove there exists a path that will eventually activate a certain state in the model. If the model has a communication protocol with a sender and receiver, the receiver will receive the sent message.

   - $A\diamond\varphi$: Verifies that φ is eventually satisfied.
   - $\varphi \rightsquigarrow \phi$: Verifies that there should exist a maximal path such that φ is always true.

UPPAAL is widely used for the conformance testing of real-time systems [54, 57]. The practical applications of UPPAAL TA for verification and validation of real-time systems have been studied and proven to be more effective than traditional testing methods [19, 58, 47, 59].

2.4.6 Model checking using UPPAAL tool

UPPAAL is a tool used for model checking of real-time systems represented as networks of timed automata [60, 55]. The UPPAAL tool has a graphical user interface for creating / editing a model, to simulate the model and to submit verification queries to the model checker. The UPPAAL graphical user interface mainly consists of the following parts: a system editor, a symbolic simulator, a concrete simulator, and a verifier.

2.4.6.1 System Editor

The system editor (Figure 2.11) is used to create and edit the model of the system under test. The test model of a system is defined by a set of process templates, global declarations, and a system definition.

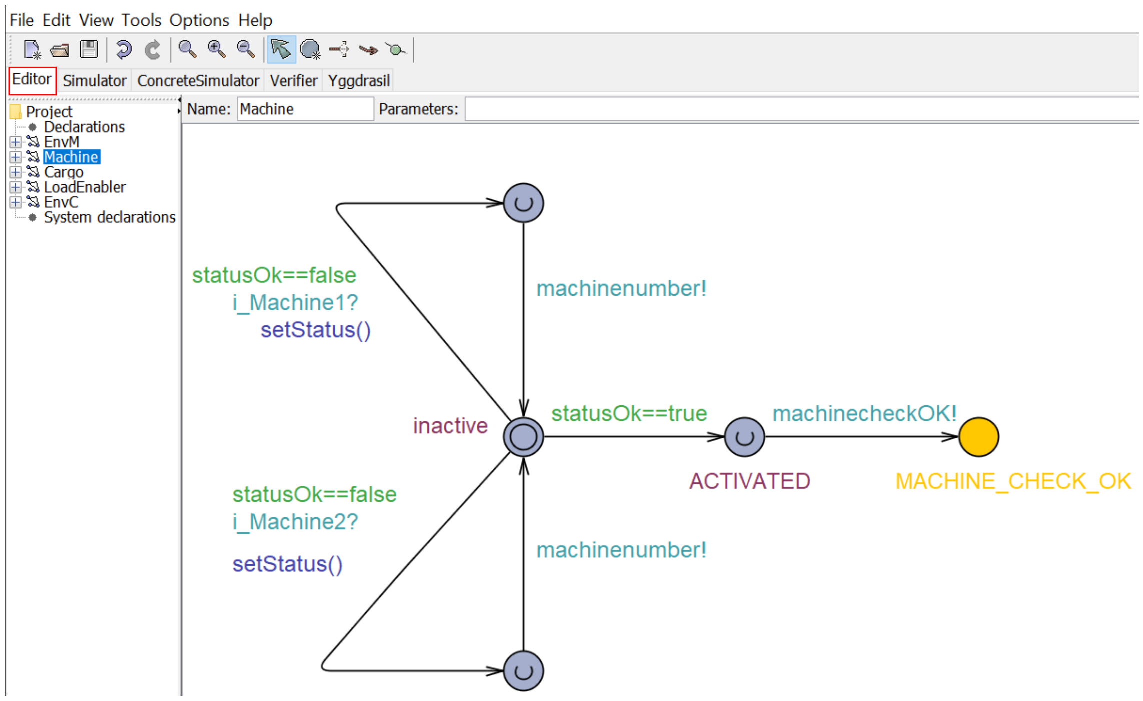


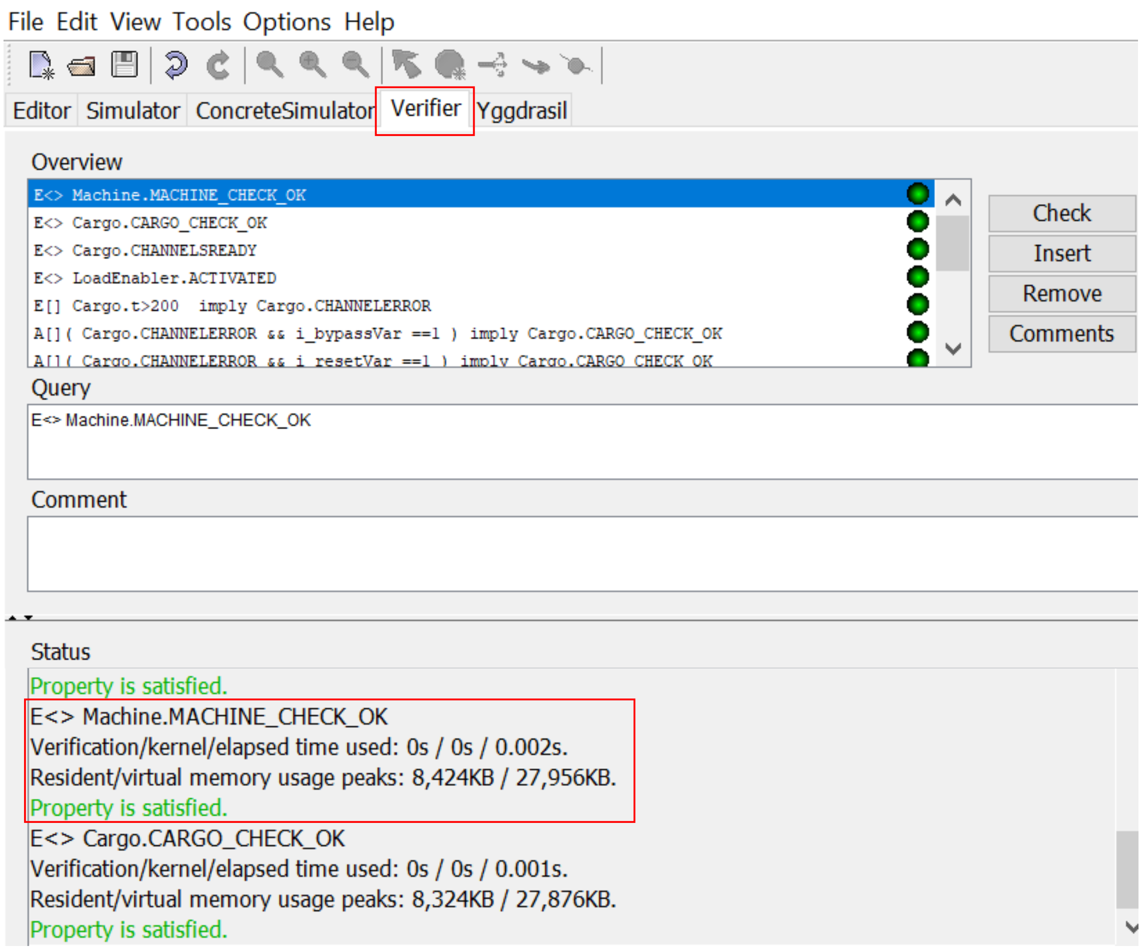
Figure 2.11: UPPAAL Editor tab

The components of a system can be accessed using the navigation tree in the left panel of the system editor. For drawing the automata model, drawing tools named *Select*, *Location*, *Branch*, *Edge*, and *Nail* are present in the tool bar of the system editor.

2.4.6.2 Verifier

The verifier tab is used to verify the system for properties such as reachability, liveness, deadlock-freeness, or safety. The verification queries can be entered and verified if the system satisfies the properties checked using a query. The two editor fields *Query* and

Comment are used to enter/edit the query and add any comments related to the query. Queries can be added using a button named *Insert* and deleted using the button named *Remove*. Queries are verified using the button named *Check* and the output is displayed in the *Status* field at the bottom of the verifier tab. Figure 2.12 shows the verification query to check the reachability property that there exists a path from the initial state,such that the machine reach the activated state.



```
E<> Machine.MACHINE_CHECK_OK
E<> Cargo.CARGO_CHECK_OK
E<> Cargo.CHANNELSREADY
E<> LoadEnabler.ACTIVATED
E[] Cargo.t>200  imply Cargo.CHANNELERROR
A[]( Cargo.CHANNELERROR && i_bypassVar ==1 ) imply Cargo.CARGO_CHECK_OK
A[]( Cargo.CHANNELERROR && i_resetVar ==1 ) imply Cargo.CARGO_CHECK_OK
```

```
E<> Machine.MACHINE_CHECK_OK
```

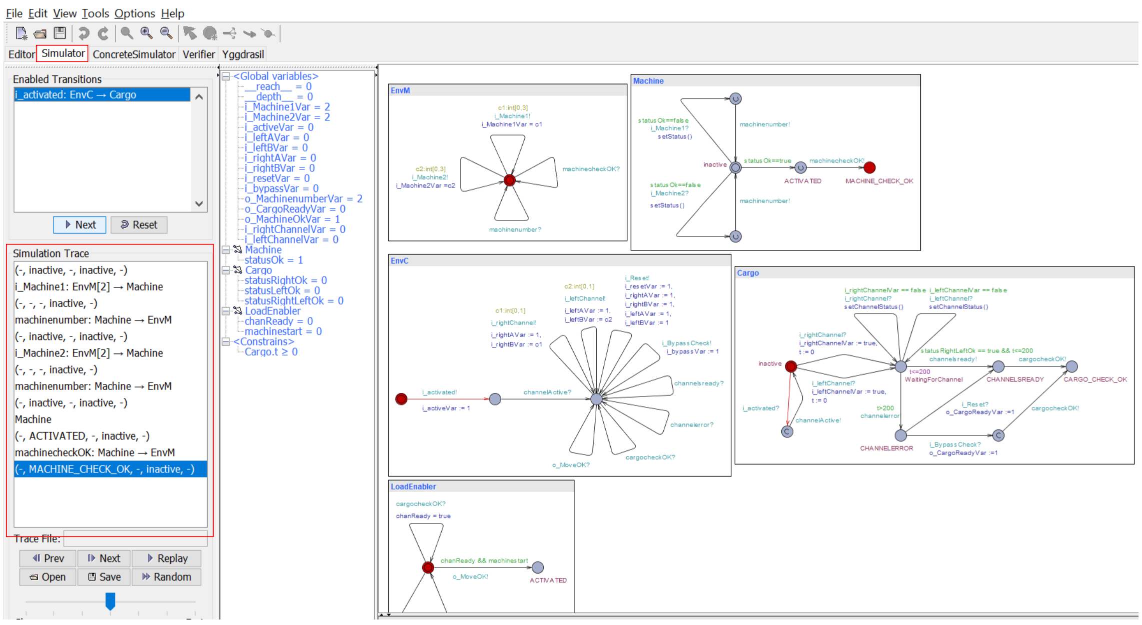
```
Property is satisfied.
E<> Machine.MACHINE_CHECK_OK
Verification/kernel/elapsed time used: 0s / 0s / 0.002s.
Resident/virtual memory usage peaks: 8,424KB / 27,956KB.
Property is satisfied.
E<> Cargo.CARGO_CHECK_OK
Verification/kernel/elapsed time used: 0s / 0s / 0.001s.
Resident/virtual memory usage peaks: 8,324KB / 27,876KB.
Property is satisfied.
```

Figure 2.12: UPPAAL Verifier tab

2.4.6.3 Symbolic Simulator

The simulator tab in the graphical interface is a validation tool to examine possible dynamic execution of the system during the modeling stage and to visualize the diagnostic trace of executions generated by the verifier. The simulator has a simulation control, variables panel, process panel, message sequence chart panel, and symbolic traces panel. The simulation control panel is used to control the simulation, and to select the state or transition to be visualized in the other two parts of the control panel. The upper part is used for step-by-step simulation and the lower part is for displaying the generated trace.

Figure 2.13 shows the generated trace for the machine activation check query showed in Figure 2.12.



Enabled Transitions

i_activated: EnvC → Cargo

Simulation Trace

```
(-, inactive, -, inactive, -)
i_Machine1: EnvM[2] → Machine
(-, -, -, inactive, -)
machinenumber: Machine → EnvM
(-, inactive, -, inactive, -)
i_Machine2: EnvM[2] → Machine
(-, -, -, inactive, -)
machinenumber: Machine → EnvM
(-, inactive, -, inactive, -)
Machine
(-, ACTIVATED, -, inactive, -)
machinecheckOK: Machine → EnvM
(-, MACHINE_CHECK_OK, -, inactive, -)
```

Trace File:

```
<Global variables>
  __reach__ = 0
  __depth__ = 0
  i_Machine1Var = 2
  i_Machine2Var = 2
  i_activeVar = 0
  i_leftAVar = 0
  i_leftBVar = 0
  i_rightAVar = 0
  i_rightBVar = 0
  i_resetVar = 0
  i_bypassVar = 0
  o_MachinenumberVar = 2
  o_CargoReadyVar = 0
  o_MachineOkVar = 1
  i_rightChannelVar = 0
  i_leftChannelVar = 0
Machine
  statusOk = 1
Cargo
  statusRightOk = 0
  statusLeftOk = 0
  statusRightLeftOk = 0
LoadEnabler
  chanReady = 0
  machinestart = 0
<Constrains>
  Cargo.t ≥ 0
```

Figure 2.13: UPPAAL Simulator tab

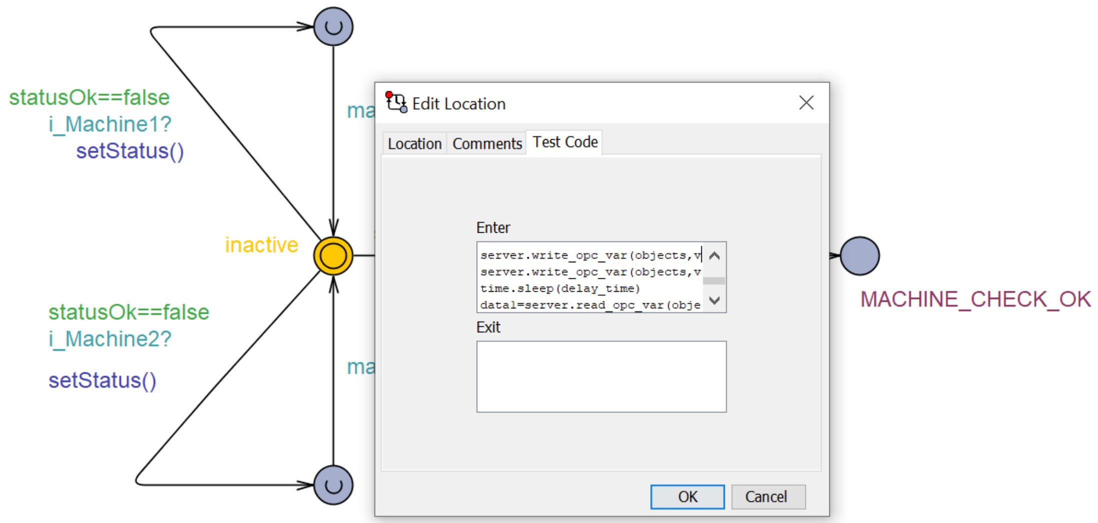
2.4.6.4 Model-based test generation using UPPAAL Yggdrasil

Yggdrasil is an offline test generation tool available in UPPAAL to generate traces from the test model and translate them into test cases as per the test code entered in the model. The test model should be deadlock-free for Yggdrasil to generate the test cases. Query-based test case generation can also be done using this feature. UPPAAL Yggdrasil is integrated into the UPPAAL main component [61]. The test code can be entered on the locations and transitions in the model to achieve the coverage criteria (Figure 2.14). Whenever model checking traverses that transition or location, it generates a trace and translates them into test cases using the test code entered. The test cases are written into files named `testcaseN.code`.

In a transition system, each state/location has two areas for entering the test code. For a location in the model, the test code can be added in two areas, `Enter` and `Exit`. When the trace reaches or leaves this location, the test code is added to the test case file. As the test scripts are executed using `Pytest` testing framework, the test code dumped on the location or transition is in Python programming language (Figure 2.15).

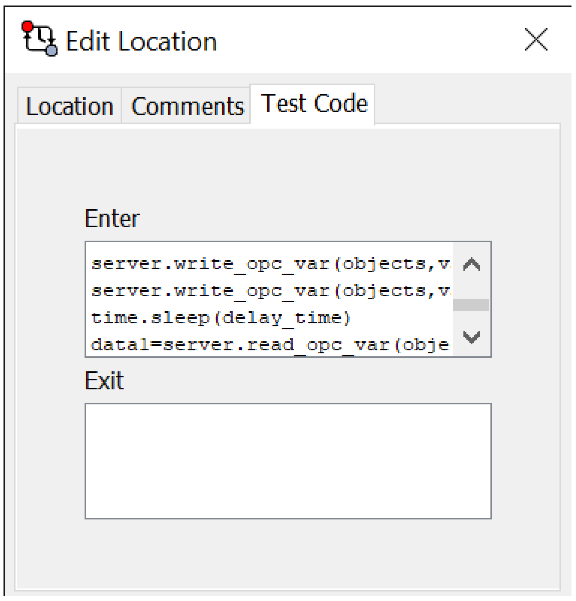
To dynamically populate the input and output value in the test code , the variables can be used in the test code as `$(var)` for global variable or `$(Process.var)` for local variables.

Traces are generated in three phases:



```
server.write_opc_var(objects,v
server.write_opc_var(objects,v
time.sleep(delay_time)
data1=server.read_opc_var(obje
```

Figure 2.14: Entering test code in UPPAAL Editor



```
server.write_opc_var(objects,v
server.write_opc_var(objects,v
time.sleep(delay_time)
data1=server.read_opc_var(obje
```

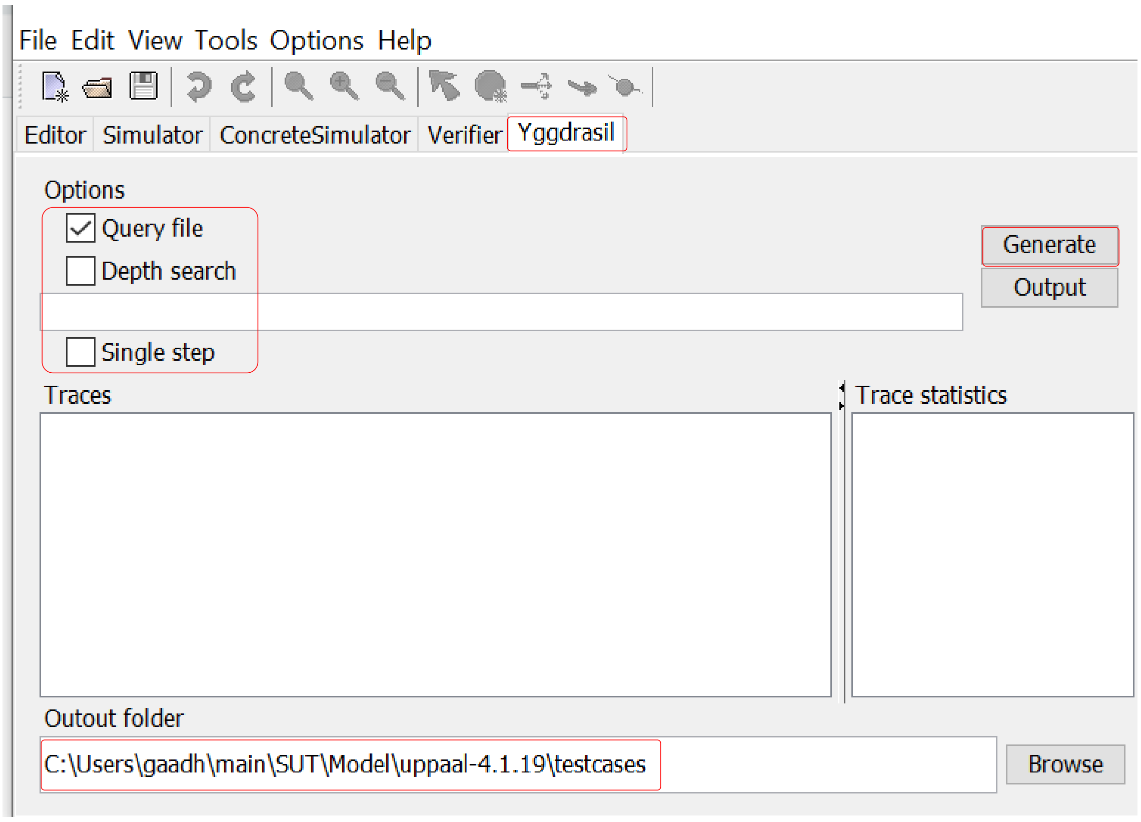
Figure 2.15: Test code entered in the behavioral model

Query file: The first phase checks the query file loaded in the verifier for verifying the properties.

Depth Search: The second phase performs a random depth-first search of the specified number of steps, and the resulting trace is used as a test case. The process should be repeated until the newly generated trace does not add to the coverage over the previous traces. In this method, a global integer variable named `__reach__` must be initialised to zero and must not be used throughout the model.

Single step: The third phase covers any transition not covered by the previous phases. It is done by formulating a reachability query to reach the uncovered transition. In this method, a global integer variable named `__single__` must be initialised to zero and must not be used throughout the model.

Yggdrasil tab (Figure 2.16) have the above options listed and pressing `Generate` button will generate the traces and translate them into test cases. Yggdrasil generates exe-



Query file [✓]

C:\Users\gaadh\main\SUT\Model\uppaal-4.1.19\testcases

Figure 2.16: Yggdrasil tab in UPPAAL GUI

cutable test cases that contain the oracle information, such as the expected output values of the SUT.

2.5 Pytest

`Pytest` is an open-source Python-based testing framework [18] that supports both unit testing and complex functional testing. Its ease of integration into a range of IDEs including CODESYS makes it an ideal choice for test automation of PLC systems. It provides flexibility in writing tests and finds the tests automatically in the test directory based on naming conventions. As mentioned in section 2.3.2, Python OPC UA library is used to connect Pytest framework to the SUT.

3. Design and implementation

The design and implementation chapter discuss the toolchain used for the model-based testing process for modeling, test case generation, and test execution to verify and validate PLC ICSs.

3.1 Methodology

The proposed methodology and the workflow of the toolchain for the MBT process are depicted in Figure 3.1.

Modeling : The scope, characteristics, and paradigm are defined to create the model of SUT using the UPPAAL tool.

Scope: The scope of the model is designed to specify the expected input-output behavior of the SUT. An input-output model can predict the expected output of the SUT for each input based on behavior as per the requirements specification. By defining the scope based on requirements, it can be verified if the output from the SUT is in sync with the model's output.

Model characteristics: The model characteristics are chosen based on the aspects of the system being tested. In this work, we deal with an event-discrete, real-time system and choose those characteristics for the model.

Model paradigm: Model is described as UPPAAL TA consisting of locations, edges and transitions. Transitions are specified using data variables or action symbols using synchronization channels.

Test Generation: The test selection criteria and test generation technology are defined to generate the tests using the UPPAAL Yggdrasil tool.

Test selection criteria: For a transition-based model, structural model coverage is used as a test selection criteria. Coverage criteria are defined for the locations and all transitions in the model.

Test generation technology: The UPPAAL model checker is used for verifying if the model satisfies the properties such as reachability, safety, and liveness. Yggdrasil offline test case generator is used to generate test cases based on traces produced by the verified

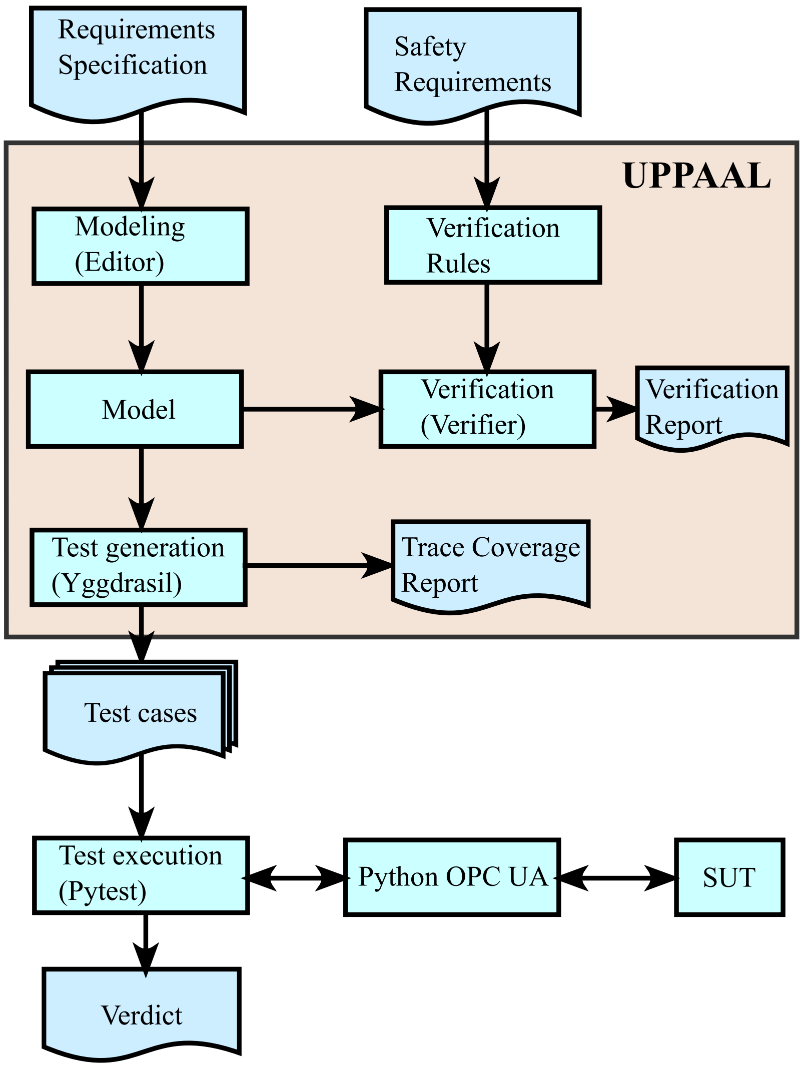


Figure 3.1: Proposed toolchain for MBT of PLC system

properties and random depth search method.

Test Execution: Offline testing using the Python testing framework `Pytest` interacts with the SUT to perform automated test execution of the created test cases. The OPC UA data exchange protocol enables communication between the CODESYS OPC UA server and the Python OPC UA client. The test summary report generated by `Pytest` is analyzed to determine the verdict of the test.

3.2 Case study

The case study in this thesis is a Machinery control system (MCS) for loading and transporting cargo using a machine unit based on different sensor readings and input commands. MCS is implemented in the PLC FBD programming language and has the following three units:

- **Machine control unit**: Activate the machine unit based on the identification number of the machine
- **Cargo control unit**: Activate the cargo unit when the rope channels associated with the loading arm of the machine is ready, and the cargo is securely hooked to the loading arm.
- **Load enabler unit**: Start the loading and transportation process when the machine and the cargo units are activated.

Figure 3.2 shows an abstract schematic representation of the MCS.

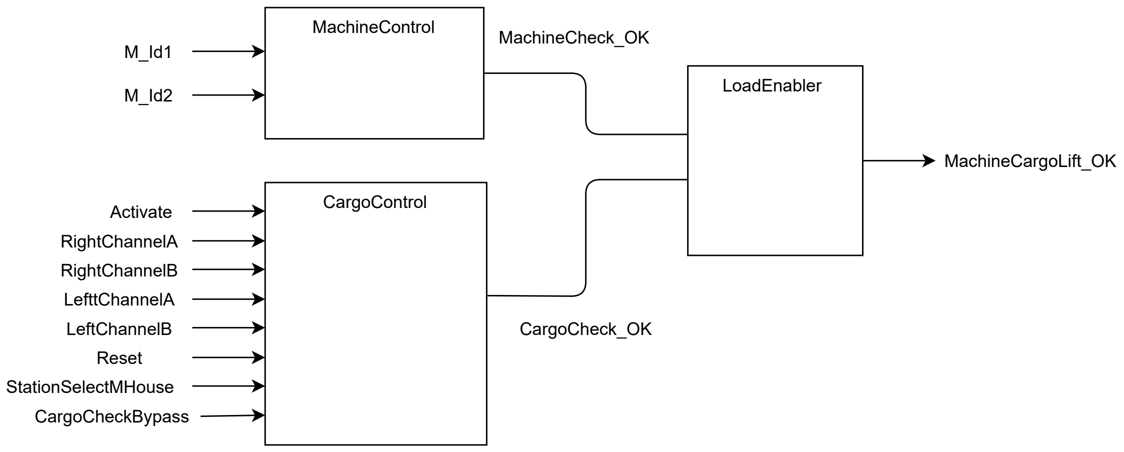


Figure 3.2: Abstract schematic representation of the MCS

Machine control unit: The machine control unit checks the identification number of the machine received as input `M_Id1` against that from the safety module `M_Id2`. If the identification numbers received match and have a non-zero value, the selected machine is ready to be activated for loading the cargo. The output from the machine control module is a boolean variable `MachineCheck_OK` that is set to true if the identification numbers match and vice-versa.

Cargo control unit: The cargo control unit checks if the cargo is ready for loading based on the following criteria:

Criteria 1: All the channel inputs enabled to true indicates that the loading arm of the cargo unit is ready. The timing constraint requires all the channel inputs (`RightChannelA, RightChannelB, LeftChannelA, LeftChannelB`) be active within 200ms of activation of the first one.

Criteria 2: When any channel input remains disabled on the activated channel and the wait time elapses, it results in a channel error. The channel error needs to be fixed by an operator, and `Reset` should be enabled to true. The reset input set to true indicates the channel error condition is corrected, and all channel inputs are enabled.

Criteria 3: When the channels are not in an error state, and the operator has verified the channels, the bypass commands can set the cargo to enabled state. `StationSelectMHouse AND CargoCheckBypass` set to true signifies that the manual check is performed and cargo is ready for loading.

The above three criteria set the cargo as ready for loading, and when both machine check and cargo check conditions are set to true, the loading process starts. All other combinations of the input signals set the output to false (`CargoCheck_OK`), and MCS remains in a stop state.

Load enabler unit: The load enabler unit can start loading the cargo when the output signal `MachineCargoLift_OK` gets enabled. It gets enabled when the `MachineCheck_OK` AND `CargoCheck_OK` are set to true indicating that machine and cargo is ready to initiate the movement of the MCS for the loading and transportation of the cargo. Figure 3.3 shows the detailed implementation of MCS using FBD.

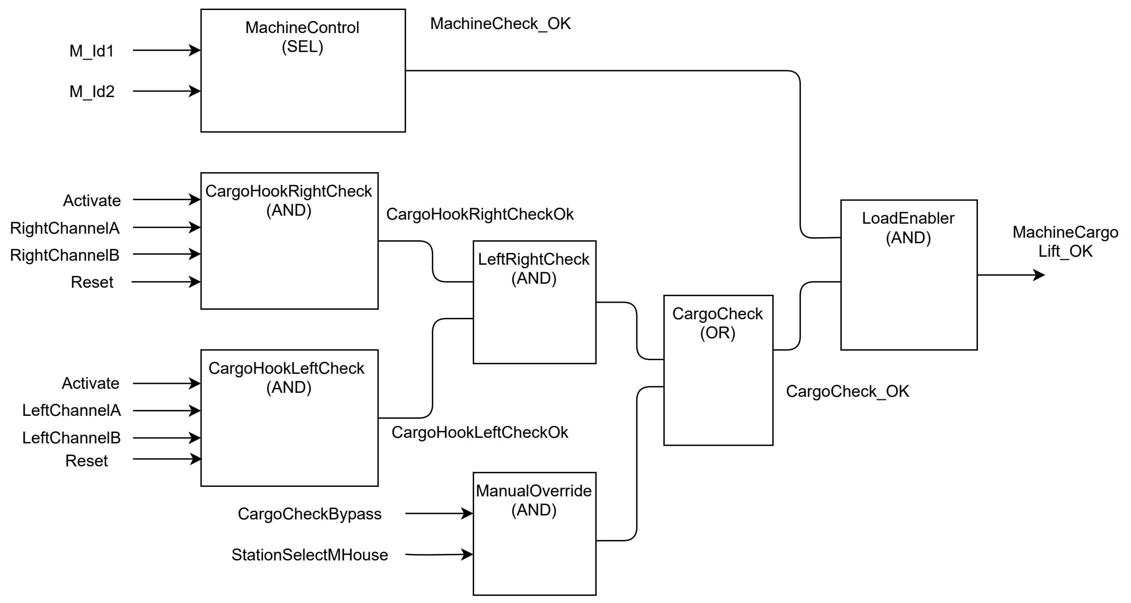


Figure 3.3: Detailed implementation of the MCS using FBD

Safety rules in MCS

The safety rules in MCS are defines as follows:

1. Machine unit: The machine control unit should ensure that the machine remains in deactivated state until the the machine identification numbers match and have a non-zero value.

2. Cargo unit: The cargo control unit follows the below safety rules:

   I The cargo unit should move to active state only when right and left rope channels are ready within 200ms of activation of the first one to pull the loading arm.

II When any of the rope channels are in an overlap state, it indicates that the channels are not ready and the cargo unit should remain in channel error state until the overlap issue is fixed by a manual intervention.

III When any overlap issue in the rope channels are fixed, and a manual reset is done by the operator, the cargo unit should move into ready state and start operating safely.

IV A manual verification of the cargo unit and the issuance of a bypass and station select request from the machine house should activate the cargo unit for its safe operation.

3. Load enabler unit : The load enabler should get activated only when both the machine and cargo ready are active. If any of them remain inactive, the load enabler should remain deactivated.

Above mentioned safety rules enforces the safety of MCS and ensures that the units in MCS operates in a safe state and never moves to an unsafe state in its operational phase.

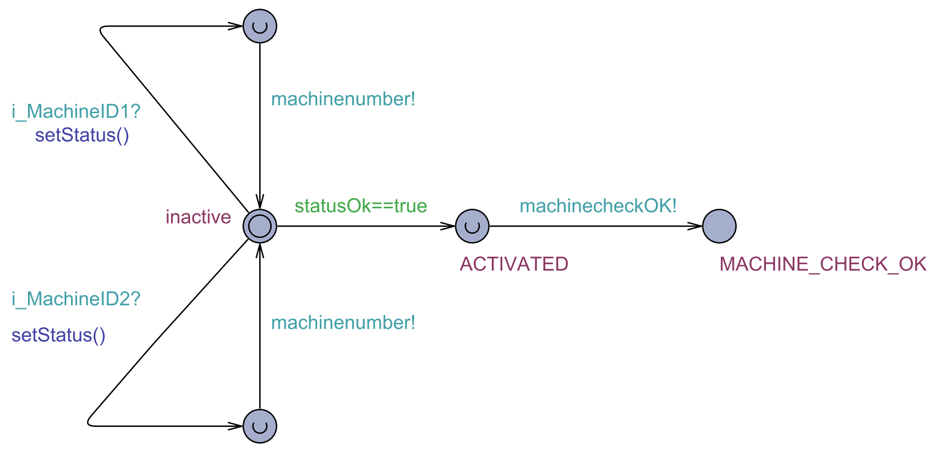
3.3 Modeling and verification

In this section, we focus on building the model of the MCS and verifying the model for properties such as reachability, safety, and liveness.

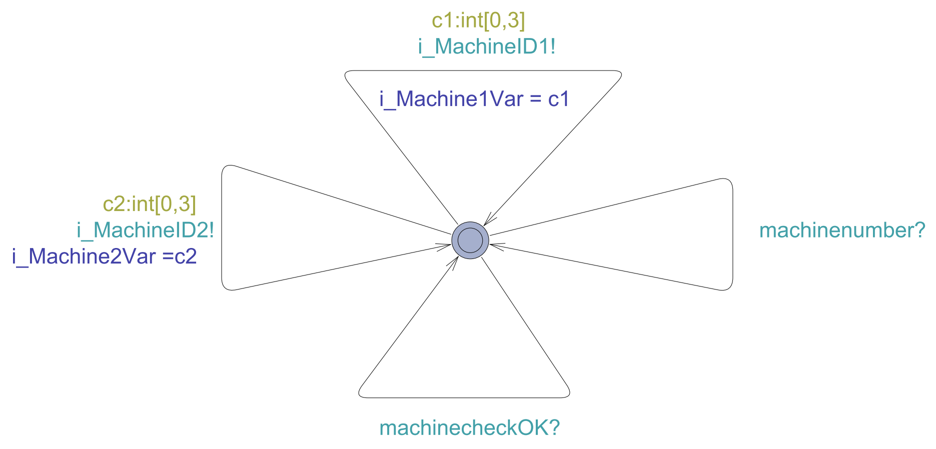
3.3.1 Building the model

An abstract behavioral model of the MCS is created using UPPAAL editor as a set of process templates, global declarations, and system declarations. The behavioral model comprises of the processes `Machine`, `Cargo`, and `LoadEnabler` and its environment `EnvM` and `EnvC`. These processes contain locations corresponding to the states of the system and transitions to move from one state to another. Transitions between states are enabled using binary synchronization channels. The channel names with the suffix '?' denotes input actions and the ones with the suffix '!' denotes the corresponding output actions.

Machine The UPPAAL TA model for machine control unit and its environment EnvM interacts with each other to identify and select the machine to be activated. It accepts the input signals,executes a function `setStatus`, and writes the output. The function `setStatus` checks if the machine identification numbers received as input match and have a non-zero value. If the condition is satisfied, the machine check activates the machine, otherwise the machine remains in inactive/stop state. Figure 3.4 shows the TA model for machine unit.



(a) Behavioral model for Machine unit

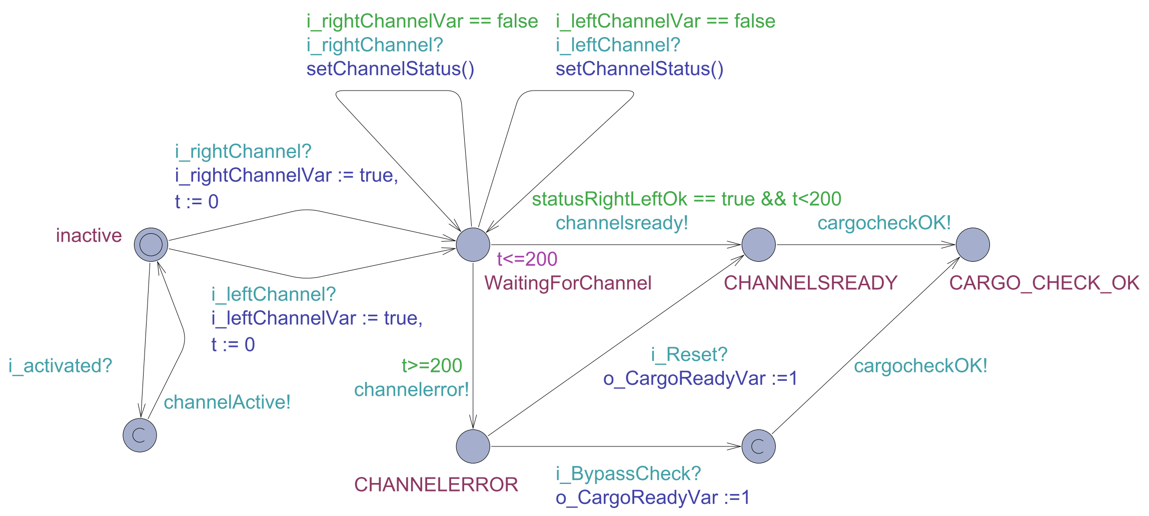


(b) Environment model for Machine unit

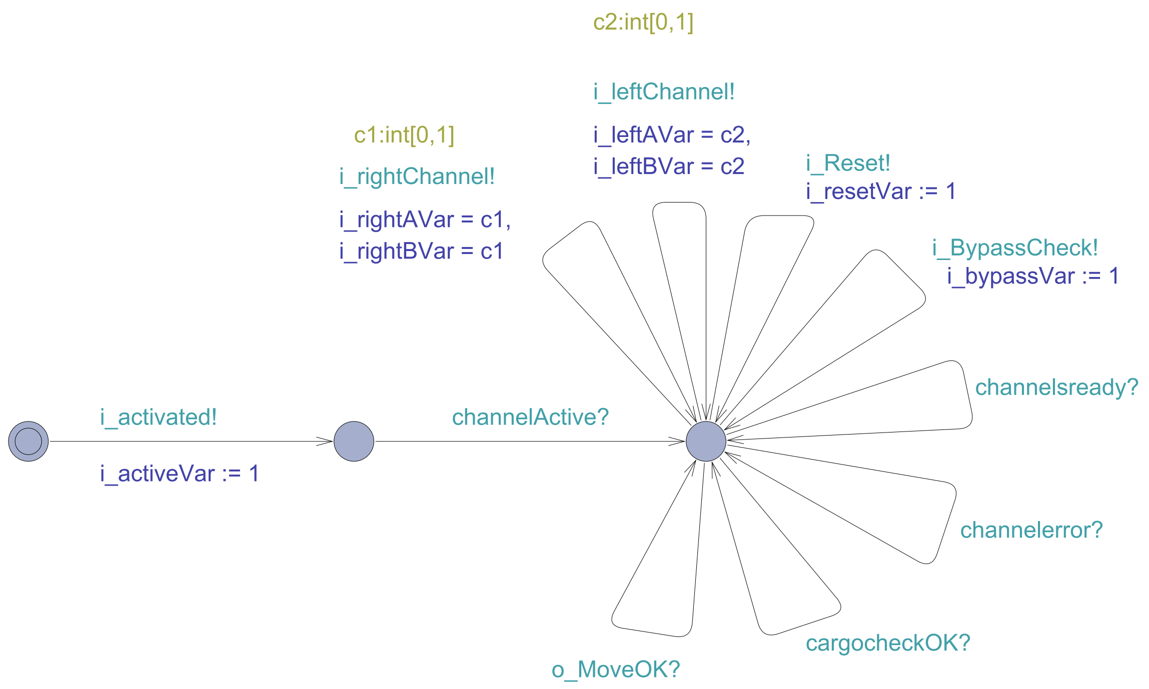
Figure 3.4: Timed Automata model for Machine unit

Cargo The UPPAAL TA model for cargo control unit and its environment EnvC interacts with each other to to check the readiness of rope channels, and any incoming request for reset,bypass and station select from machine house. Time invariant $t \leq 200$ in the location `WaitingForChannel` enforces the system to wait 200 milliseconds to check the readiness of rope channels, before it takes an outgoing transition. The transition from `WaitingForChannel` to `CHANNELERROR` location reflects that the all input signals from the rope channels on left and right are not received and there is a rope channel overlap condition indicating an error in rope channels, when the time condition is $t \geq 200$. The cargo model accepts the input signals,executes a function `setChannelStatus`, and writes the output. The function `setChannelStatus` checks if the channel inputs on left and right channels are enabled and decides if the cargo unit can be enabled or require a manual intervention to issue a reset or bypass and station select commands to overcome the channel error state.

Figure 3.5 shows the TA model for machine unit.



(a) Behavioral model for Cargo unit



(b) Environment model for Cargo unit

Figure 3.5: Timed Automata model for Cargo unit

LoadEnabler The UPPAAL TA model for the cargo control unit takes the output signals from machine control and cargo control units and checks if the machine unit and cargo unit is activated and ready to load the cargo. The system requirement states that the load enabler should get activated only when both the machine and cargo ready are active. If any of them remain inactive, the load enabler should remain deactivated. Figure 3.6 shows the TA model for LoadEnabler unit.

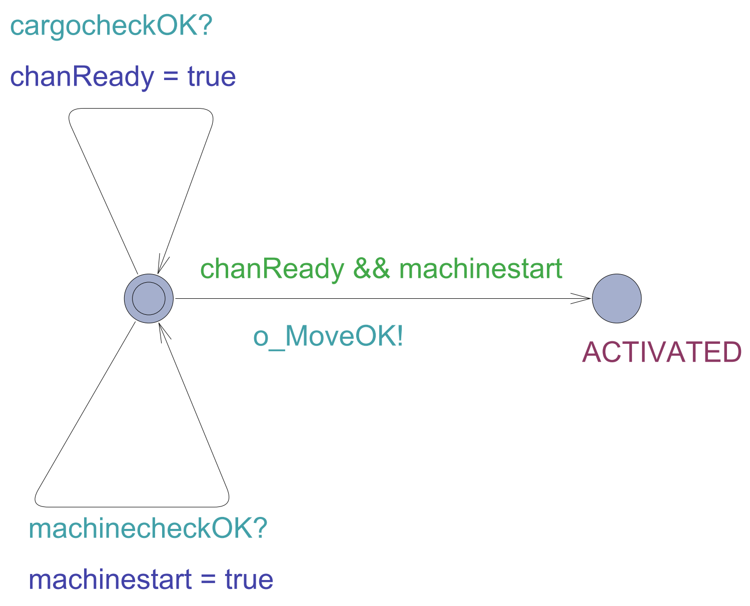


Figure 3.6: Timed Automata behavioral model for LoadEnabler unit

The input and output signals in the behavioral model of MCS are listed in the Table 3.1.

Model	**Inputs**	**Output**
Machine	i_MachineID1	machinecheckOK
	i_MachineID2	
Cargo	i_activated	cargocheckOK
	i_rightChannel	
	i_leftChannel	
	i_Reset	
	i_BypassCheck	
LoadEnabler	machinecheckOK	o_MoveOK
	cargocheckOK	

Table 3.1: Input output structure of UPPAAL TA behavioral model of MCS.

The FBD blocks are translated into predefined UPPAAL operators as below:

- The Logical Operator blocks are translated using the logical UPPAAL operators and, not, or.
- The Arithmetic Operator blocks are translated using the arithmetic UPPAAL operators +, =, - , /, *.
- The Comparison blocks are translated using the relational operators UPPAAL <, >, <=, >=, =.
- The Selection blocks are translated using if-then-else statements.

3.3.2 Verification

Reachability properties: The reachability properties checks if a location/state in the model is reachable, e.g.:

1. *E<> Machine.MACHINE_CHECK_OK*: Checks if the machine check is done and the machine is activated and ready to pick the cargo.

2. *E<> Cargo.CARGO_CHECK_OK*: Checks if the cargo check is done and the cargo is ready for loading.

3. *E<> LoadEnabler.ACTIVATED*: Checks if the loading process starts when the machine and cargo are ready.

Safety properties: The safety properties verified in the MCS model are:

1. *A[](i_Machine1Var==0 && i_Machine2Var==0)* || *(i_Machine1Var != i_Machine2Var) imply !Machine.statusOk*: Ensures that the machine remains in a deactivated state when machine identification numbers do not match or when they are set to zero.

2. *E[](Cargo.statusRightLeftOk && Cargo.t < 200) imply Cargo.CARGO_CHECK_OK*: Ensures that the cargo unit should move to active state only when right and left rope channels are ready within 200ms of activation of the first one to pull the loading arm.

3. *E[] Cargo.t>=200 imply Cargo.CHANNELERROR*: Ensures that the cargo system moves to channel error state when any of the rope channels are in an overlap state and the time-invariant $t >= 200$ while waiting for all the channel inputs to arrive.

4. *A[](Cargo.CHANNELERROR && i_resetVar==1) imply Cargo.CARGO_CHECK_OK*: Ensures that when overlap condition in the rope channels are fixed, and a manual reset is done by the operator, the cargo unit should move into ready state and start operating safely.

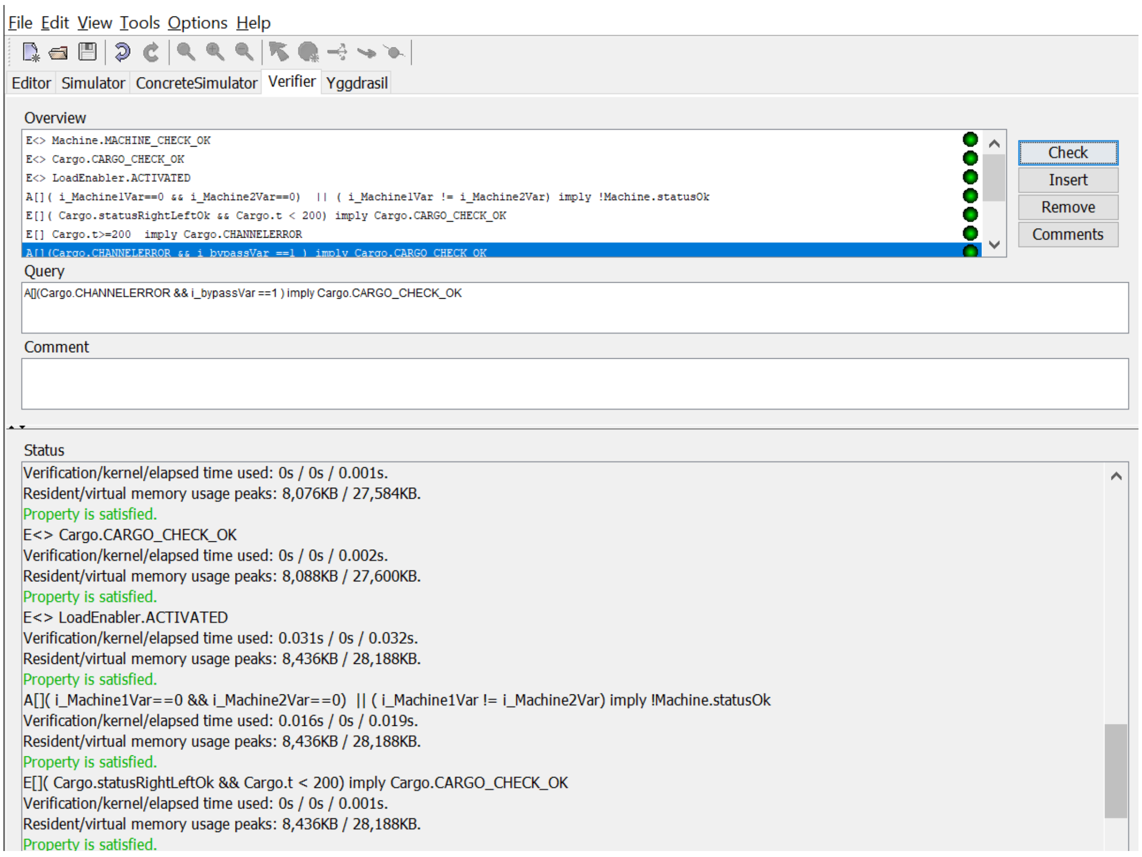
5. *A[](Cargo.CHANNELERROR && i_bypassVar==1) imply Cargo.CARGO_CHECK_OK*: Ensures that a manual verification of the cargo unit and the issuance of a bypass and station select request from the machine house should activate the cargo unit for its safe operation.

Liveness properties: The liveness properties verified in the MCS model are:

1. *A<> Machine.MACHINE_CHECK_OK && Cargo.CARGO_CHECK_OK imply LoadEnabler.ACTIVATED*: Ensures that the load enabler unit starts the loading process when the machine and cargo units are activated.

2. *A<> (Cargo.statusRightLeftOk && Cargo.t<200) imply Cargo.CHANNELSREADY*: Ensures that the cargo unit moves to channels ready state when the left and right channel inputs are received within the time-invariant condition *t<200*

The verification queries are executed using UPPAAL's verification engine `verifyta`. Figure 3.7 shows the verification results in the UPPAAL verifier tab.


File Edit View Tools Options Help
Editor Simulator ConcreteSimulator Verifier Yggdrasil
Overview
Check
Insert
Remove
Comments
Query
Comment
Status

```
E<> Machine.MACHINE_CHECK_OK
E<> Cargo.CARGO_CHECK_OK
E<> LoadEnabler.ACTIVATED
A[]( i_Machine1Var==0 && i_Machine2Var==0)  || ( i_Machine1Var != i_Machine2Var) imply !Machine.statusOk
E[]( Cargo.statusRightLeftOk && Cargo.t < 200) imply Cargo.CARGO_CHECK_OK
E[] Cargo.t>=200  imply Cargo.CHANNELERROR
A[](Cargo.CHANNELERROR && i_bypassVar ==1 ) imply Cargo.CARGO_CHECK_OK
```

```
A[](Cargo.CHANNELERROR && i_bypassVar ==1 ) imply Cargo.CARGO_CHECK_OK
```

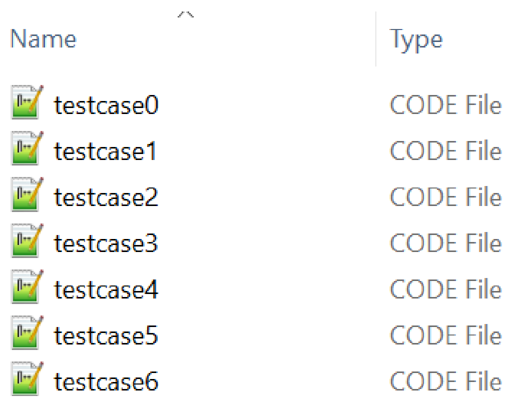
```
Verification/kernel/elapsed time used: 0s / 0s / 0.001s.
Resident/virtual memory usage peaks: 8,076KB / 27,584KB.
Property is satisfied.
E<> Cargo.CARGO_CHECK_OK
Verification/kernel/elapsed time used: 0s / 0s / 0.002s.
Resident/virtual memory usage peaks: 8,088KB / 27,600KB.
Property is satisfied.
E<> LoadEnabler.ACTIVATED
Verification/kernel/elapsed time used: 0.031s / 0s / 0.032s.
Resident/virtual memory usage peaks: 8,436KB / 28,188KB.
Property is satisfied.
A[]( i_Machine1Var==0 && i_Machine2Var==0)  || ( i_Machine1Var != i_Machine2Var) imply !Machine.statusOk
Verification/kernel/elapsed time used: 0.016s / 0s / 0.019s.
Resident/virtual memory usage peaks: 8,436KB / 28,188KB.
Property is satisfied.
E[]( Cargo.statusRightLeftOk && Cargo.t < 200) imply Cargo.CARGO_CHECK_OK
Verification/kernel/elapsed time used: 0s / 0s / 0.001s.
Resident/virtual memory usage peaks: 8,436KB / 28,188KB.
Property is satisfied.
```

Figure 3.7: Verification of queries using UPPAAL Verifier

3.4 Test generation

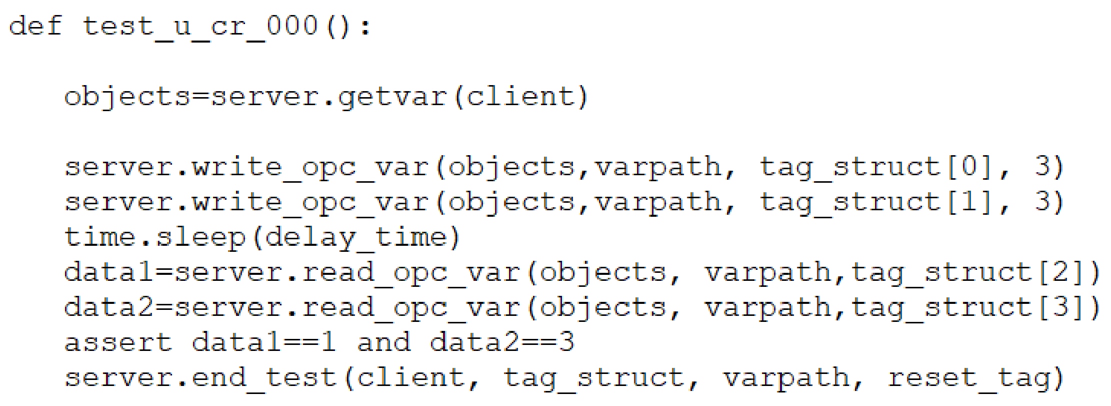
A model-based offline test generator UPPAAL Yggdrasil [59] is used for generating test suites. It uses a three-step strategy to achieve the coverage criteria: Requirements based test purposes formulated as reachability, safety, and liveness properties, random execution, and structural coverage of the transitions in the model.

The test cases are generated using Yggdrasil tool from the requirements based verification queries entered in the model and random depth-first search specifying the number of steps as 10. Figures 3.8 and 3.9 show the seven test case files containing 30 test cases created in the output test directory and a test case to test the machine unit identity module.



Name	Type
testcase0	CODE File
testcase1	CODE File
testcase2	CODE File
testcase3	CODE File
testcase4	CODE File
testcase5	CODE File
testcase6	CODE File

Figure 3.8: Test case files generated using Yggdrasil



```
def test_u_cr_000():

    objects=server.getvar(client)

    server.write_opc_var(objects,varpath, tag_struct[0], 3)
    server.write_opc_var(objects,varpath, tag_struct[1], 3)
    time.sleep(delay_time)
    data1=server.read_opc_var(objects, varpath,tag_struct[2])
    data2=server.read_opc_var(objects, varpath,tag_struct[3])
    assert data1==1 and data2==3
    server.end_test(client, tag_struct, varpath, reset_tag)
```

Figure 3.9: Tests in the test case file

Figure 3.9 shows a test case generated for the machine identity check function in the POU for MCS. The test case sends the machine identification number generated by the behavioral model of machine unit to the SUT, and collects the output from SUT. The model dynamically generated an input value 3 for *MachineID1* and *MachineID2*. It is passed using `write` function to the variable structure defined in the program `Tags.py`, `tag_struct[0]` and `tag_struct[1]` corresponding to *MachineID1* and *MachineID2*. A delay time of 2s is added between the write inputs and read outputs using Python `time` function to represent the PLC execution cycle time. The output values returned from the application program running on the simulated device is collected using the `read` function. An `assert` statement in the Python program checks if the output from the application program matches the output defined in the test model.

Each test case file contains executable tests with the input values to be send to SUT and checks the output value from SUT against the output generated by the behavioral model. A Python script named `gen_tests.py` combines the multiple test case files generated by UPPAAL Yggdrasil to a single Python test file.

3.5 Test execution

The offline testing method is used for automated test execution of the generated test cases by interacting with the SUT. A Python script converts the test case files generated by UPPAAL Yggdrasil to a Python test file. Automatic execution is performed by setting up an adapter to connect to the SUT.

OPC UA: OPC UA is a platform-independent standard based on TCP that can be used as a data exchange protocol for industrial automation. OPC UA is used as the adapter to connect to the CODESYS development environment where the FBD PLC implementation code resides (Figure 3.10).

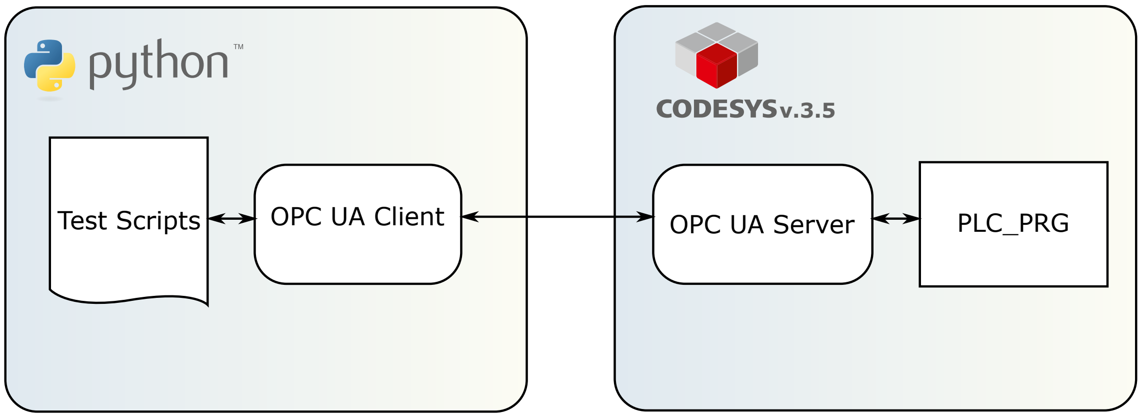


Figure 3.10: OPC UA adaptor layer for test execution

CODESYS OPC UA Server: CODESYS OPC UA server communicates with the OPC UA client that is set up using the Python OPC UA library. CODESYS V3.5 SP16 Patch 4 is used as the development environment for hosting the SUT as it supports the OPC UA features.

Python OPC UA Client: Python OPC UA client connects to the CODESYS OPC UA server and browse the address space in the PLC program. When the connection is established, the Python tests can send values to the input variables while the application program is running in CODESYS development environment.

Pytest: An open-source Python-based testing framework `Pytest` [18] is used to execute the Python test file. The test directory consists of the following three Python programs:

- `OPC.py`: Contains the function to connect to OPC UA client, read, write, and reset the value of the variables in the address space. The code below shows the function to connect to OPC UA client and read read the data objects from the root node in the address to access the variables in the PLC program.

```python
def OPC_Connect(client) :
    client.connect()

def getvar(client):
    root = client.get_root_node()
    objects = client.get_root_node()

    return objects
```

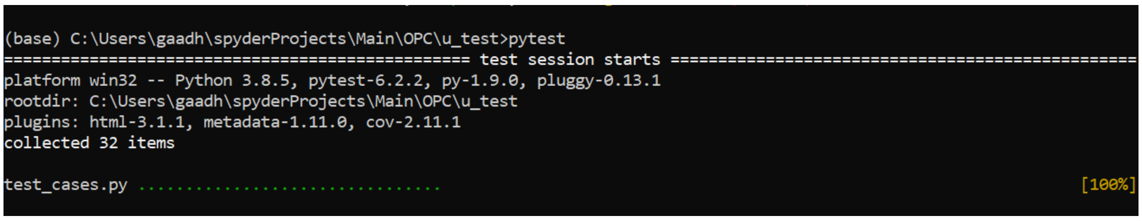
- `Tags.py`: Contains the variable structure in the PLC program, and the variable path defined in it. The code below shows the mapping of variables in the machine unit PLC program (MachineTestTags) and the variable path to reach them (MachineTestpath).

```python
   MachineTestTags = ["MacineID1",        #0
                      "MacineID2",        #1
                      "MachineStatus",    #2
                      "MachineNumber",    #3
                      ]

    CalibTags = {
                 }

    MachineTestpath = ["0:Objects", "2:DeviceSet","4:CODESYS Control Win V3",
    "3:Resources" , "4:Application" , "3:Programs","4:PLC_PRG","var"]
```

- `test_cases.py`: Contains the test cases generated by the UPPAAL Yggdrasil tool.

The test cases are then executed by invoking `Pytest` utility with the command *pytest*. This command will execute all tests in the files named as `test_*.py` in the test directory. The test cases are executed on the SUT. The application program runs on a standard installation of CODESYS V3.5 SP16 Patch 4 that includes an OPC UA server. Python OPC UA client communicates with the CODESYS OPC UA server, and the test scripts send the values to input variables in the application program. There is also a delay time added in the Python code after sending the input values. This delay time represents the program execution time. Following this, the output values are sent back to the Python program. An `assert` statement in the Python program checks if the output from the application program matches the output generated from the test model. The test results are analyzed to determine the verdict of the test.

The test execution is initiated by invoking `Pytest` utility with the command *pytest* (Figure 3.11). The test suite contains the functions to connect and disconnect to OPC client and the 30 test cases generated by Yggdrasil.



```
(base) C:\Users\gaadh\spyderProjects\Main\OPC\u_test>pytest
============================================== test session starts ==============================================
platform win32 -- Python 3.8.5, pytest-6.2.2, py-1.9.0, pluggy-0.13.1
rootdir: C:\Users\gaadh\spyderProjects\Main\OPC\u_test
plugins: html-3.1.1, metadata-1.11.0, cov-2.11.1
collected 32 items

test_cases.py ................................                                                            [100%]
```

Figure 3.11: Test execution using `pytest`

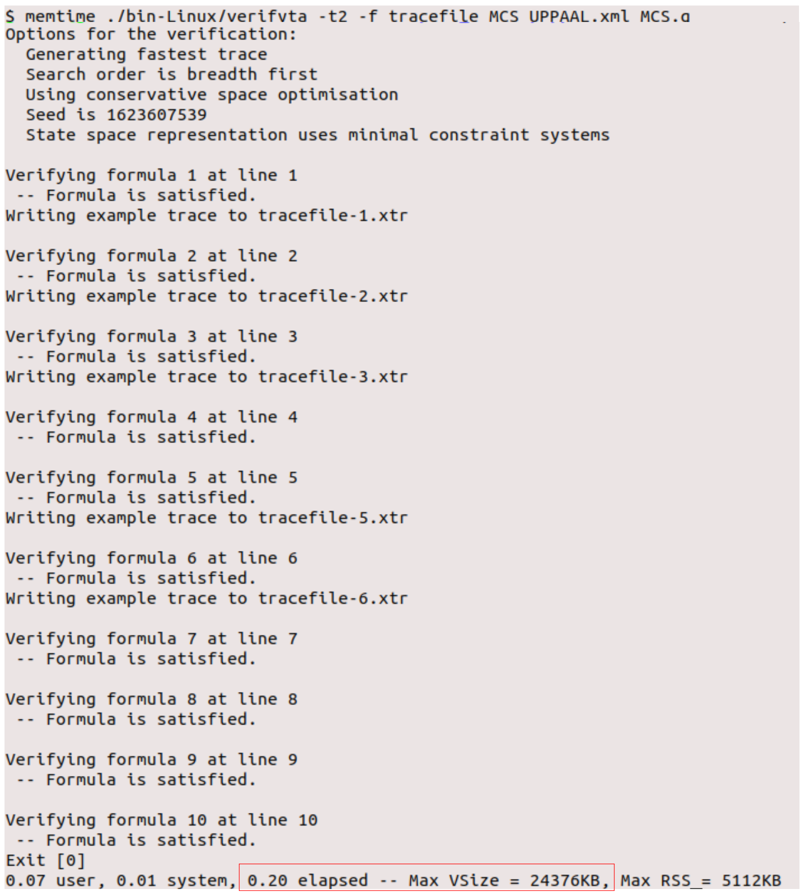
Each test case has a test condition stated in the requirement specification. The input values for the test condition generated by the behavioral model is sent to the application program, and the output is collected. The output values produced during the application program's execution are collected and compared to the output defined in the behavioral model. A passed test indicates that the SUT and behavioral model conforms to the expected input-output behaviour specified in the requirements.

4. Results and Evaluation

In this chapter, the case study results and the findings from the verification, test generation, and coverage measurements collected from the test execution are discussed.

4.1 Verification effort

The behavioral model of MCS is verified for the reachability, safety, and liveness properties using UPPAAL's verification engine `verifyta`. Figure 4.1 shows the result of executing *Memtime* to measure the memory usage and time for verification in UPPAAL.



```
$ memtime ./bin-Linux/verifyta -t2 -f tracefile MCS UPPAAL.xml MCS.q
Options for the verification:
  Generating fastest trace
  Search order is breadth first
  Using conservative space optimisation
  Seed is 1623607539
  State space representation uses minimal constraint systems

Verifying formula 1 at line 1
 -- Formula is satisfied.
Writing example trace to tracefile-1.xtr

Verifying formula 2 at line 2
 -- Formula is satisfied.
Writing example trace to tracefile-2.xtr

Verifying formula 3 at line 3
 -- Formula is satisfied.
Writing example trace to tracefile-3.xtr

Verifying formula 4 at line 4
 -- Formula is satisfied.

Verifying formula 5 at line 5
 -- Formula is satisfied.
Writing example trace to tracefile-5.xtr

Verifying formula 6 at line 6
 -- Formula is satisfied.
Writing example trace to tracefile-6.xtr

Verifying formula 7 at line 7
 -- Formula is satisfied.

Verifying formula 8 at line 8
 -- Formula is satisfied.

Verifying formula 9 at line 9
 -- Formula is satisfied.

Verifying formula 10 at line 10
 -- Formula is satisfied.
Exit [0]
0.07 user, 0.01 system, 0.20 elapsed -- Max VSize = 24376KB, Max RSS_= 5112KB
```

Figure 4.1: Memory usage and running time measurement

Memtime is a utility program developed by the UPPAAL group to record the peak memory usage and running time [62]. It can be used with the Linux version of `verifyta` to measure the peak memory usage and verification time using the following command:

```
$ memtime verifyta -t2 -f tracefile MCS_UPPAAL.xml MCS.q
```

where the parameters are defined as follows:

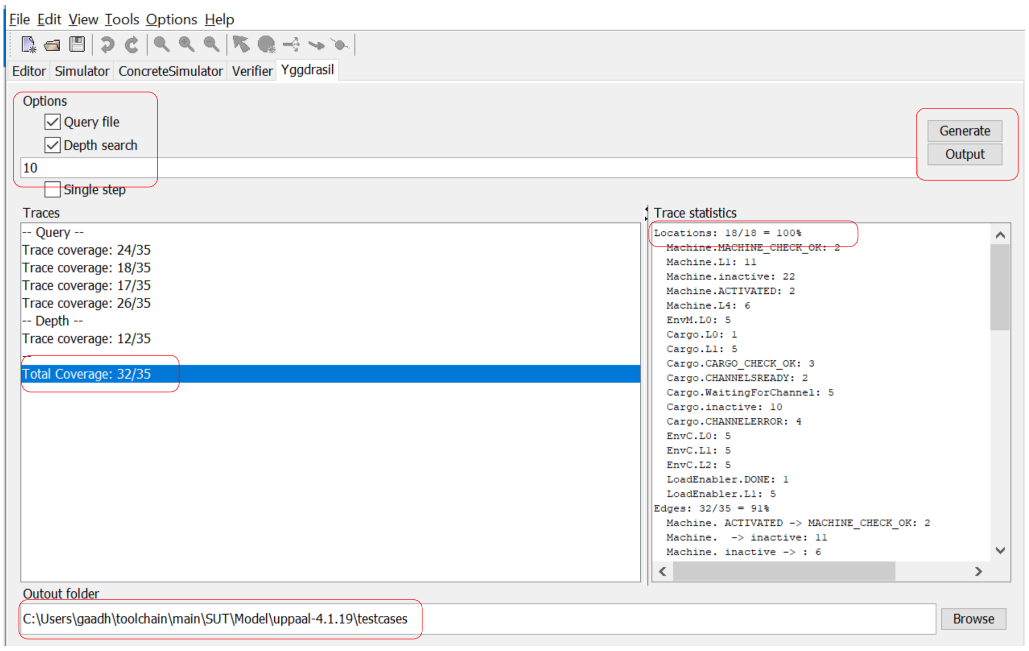
-t2: Instructs UPPAAL to find the shortest simulation trace to the target location defined in the query file `MCS.q`

-f: Dumps out the trace into a file specified by *tracefile* upon successful completion of the model checking and verification of the behavioral model `MCS_UPPAAL.xml`

The verification process takes `0.20s` to complete, and the peak memory usage is `24376 KB`.

4.2 Test efficiency

The reachability, safety, and liveness properties defined for MCS comprising ten verification queries (Section 3.3.2) took `0.20s` to complete. The test generation using query file provided 88% location coverage, whereas random depth-first provided 77% while generating the test cases using each approach separately. However, selecting both query file and random depth search options generated 30 test cases in `0.10s` which provided 100% location coverage (Figure 4.2).


File Edit View Tools Options Help
Editor Simulator ConcreteSimulator Verifier Yggdrasil
Generate
Output
Browse

Options
[✓] Query file
[✓] Depth search
10
[ ] Single step

Traces
```
-- Query --
Trace coverage: 24/35
Trace coverage: 18/35
Trace coverage: 17/35
Trace coverage: 26/35
-- Depth --
Trace coverage: 12/35
--
Total Coverage: 32/35
```

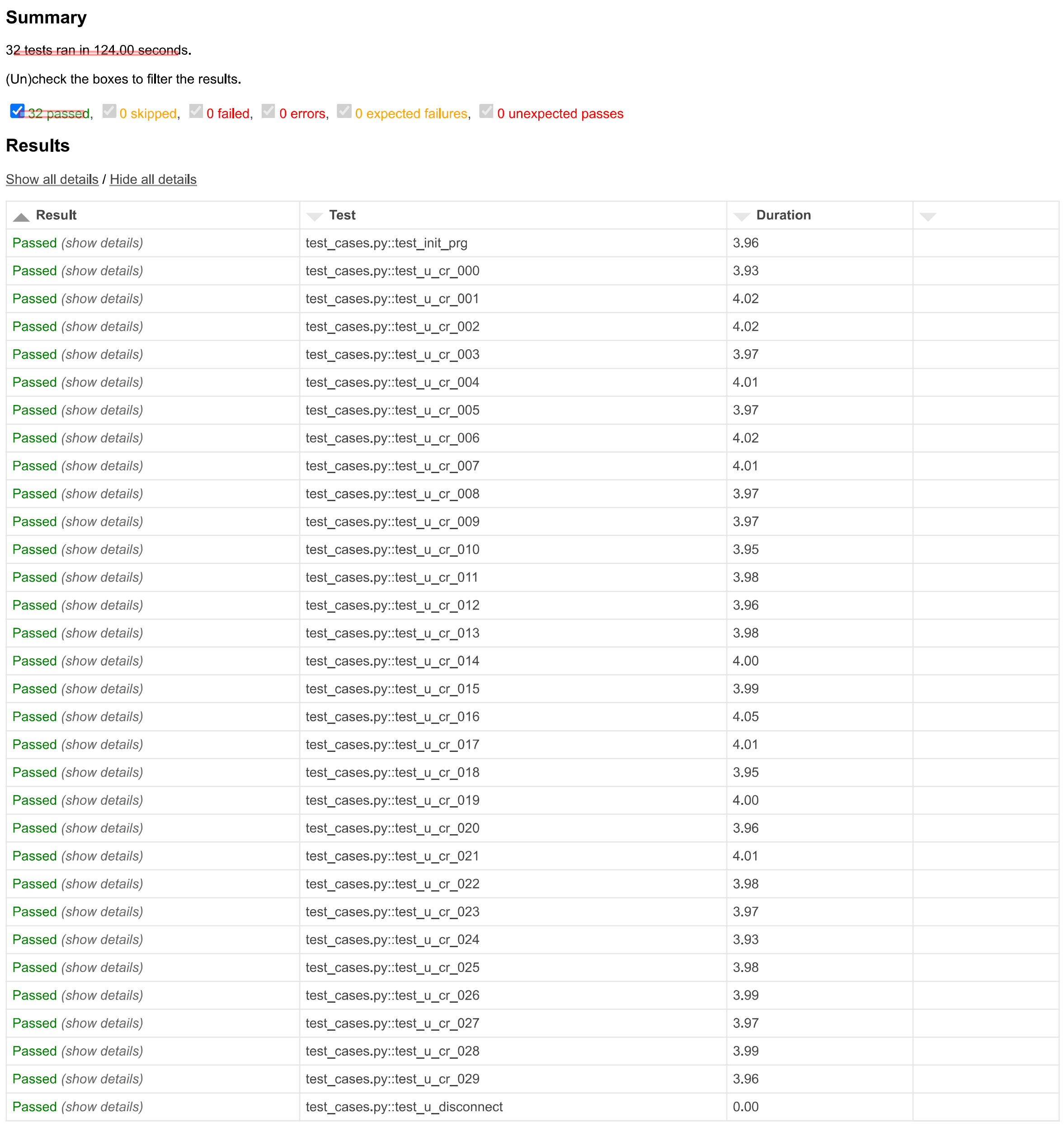
Trace statistics
```
Locations: 18/18 = 100%
  Machine.MACHINE_CHECK_OK: 2
  Machine.L1: 11
  Machine.inactive: 22
  Machine.ACTIVATED: 2
  Machine.L4: 6
  EnvM.L0: 5
  Cargo.L0: 1
  Cargo.L1: 5
  Cargo.CARGO_CHECK_OK: 3
  Cargo.CHANNELSREADY: 2
  Cargo.WaitingForChannel: 5
  Cargo.inactive: 10
  Cargo.CHANNELERROR: 4
  EnvC.L0: 5
  EnvC.L1: 5
  EnvC.L2: 5
  LoadEnabler.DONE: 1
  LoadEnabler.L1: 5
Edges: 32/35 = 91%
  Machine. ACTIVATED -> MACHINE_CHECK_OK: 2
  Machine.  -> inactive: 11
  Machine. inactive -> : 6
```

Outout folder
C:\Users\gaadh\toolchain\main\SUT\Model\uppaal-4.1.19\testcases

Figure 4.2: Test case generation using Yggdrasil

These 30 test cases are generated from the trace created based on the reachability, safety, and liveness queries defined for the MCS in section 3.3.2 and the random depth search based on specified number of steps. Yggdrasil trace statistics shows a total coverage of 91% as edge coverage and 100% location coverage.

Test summary report generated using `pytest` testing framework (Figure 4.3) shows the verdict of each test case and the execution time.



Summary

32 tests ran in 124.00 seconds.

(Un)check the boxes to filter the results.

☑ 32 passed, ☑ 0 skipped, ☑ 0 failed, ☑ 0 errors, ☑ 0 expected failures, ☑ 0 unexpected passes

Results

Show all details / Hide all details

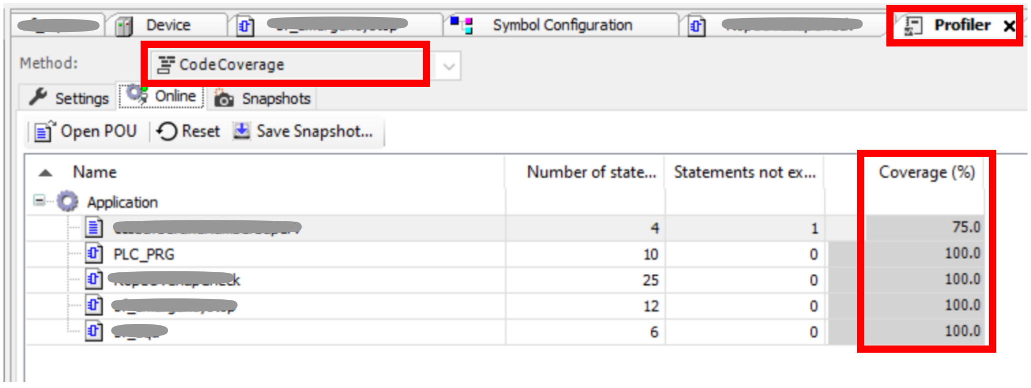
Result	Test	Duration	
Passed *(show details)*	test_cases.py::test_init_prg	3.96	
Passed *(show details)*	test_cases.py::test_u_cr_000	3.93	
Passed *(show details)*	test_cases.py::test_u_cr_001	4.02	
Passed *(show details)*	test_cases.py::test_u_cr_002	4.02	
Passed *(show details)*	test_cases.py::test_u_cr_003	3.97	
Passed *(show details)*	test_cases.py::test_u_cr_004	4.01	
Passed *(show details)*	test_cases.py::test_u_cr_005	3.97	
Passed *(show details)*	test_cases.py::test_u_cr_006	4.02	
Passed *(show details)*	test_cases.py::test_u_cr_007	4.01	
Passed *(show details)*	test_cases.py::test_u_cr_008	3.97	
Passed *(show details)*	test_cases.py::test_u_cr_009	3.97	
Passed *(show details)*	test_cases.py::test_u_cr_010	3.95	
Passed *(show details)*	test_cases.py::test_u_cr_011	3.98	
Passed *(show details)*	test_cases.py::test_u_cr_012	3.96	
Passed *(show details)*	test_cases.py::test_u_cr_013	3.98	
Passed *(show details)*	test_cases.py::test_u_cr_014	4.00	
Passed *(show details)*	test_cases.py::test_u_cr_015	3.99	
Passed *(show details)*	test_cases.py::test_u_cr_016	4.05	
Passed *(show details)*	test_cases.py::test_u_cr_017	4.01	
Passed *(show details)*	test_cases.py::test_u_cr_018	3.95	
Passed *(show details)*	test_cases.py::test_u_cr_019	4.00	
Passed *(show details)*	test_cases.py::test_u_cr_020	3.96	
Passed *(show details)*	test_cases.py::test_u_cr_021	4.01	
Passed *(show details)*	test_cases.py::test_u_cr_022	3.98	
Passed *(show details)*	test_cases.py::test_u_cr_023	3.97	
Passed *(show details)*	test_cases.py::test_u_cr_024	3.93	
Passed *(show details)*	test_cases.py::test_u_cr_025	3.98	
Passed *(show details)*	test_cases.py::test_u_cr_026	3.99	
Passed *(show details)*	test_cases.py::test_u_cr_027	3.97	
Passed *(show details)*	test_cases.py::test_u_cr_028	3.99	
Passed *(show details)*	test_cases.py::test_u_cr_029	3.96	
Passed *(show details)*	test_cases.py::test_u_disconnect	0.00	

Figure 4.3: Test summary from `pytest`

The test execution took `124s` to execute, and all of them passed, according to the test summary report generated by `pytest` . The verification, test generation, and test execution process took `124.30s` to complete. The test summary shows that the SUT passed all the tests generated based on the functional and safety features defined in the specification and achieved 100% code coverage at the PLC program level. The testing approach followed is black-box MBT, and code coverage is used as a metric for measuring the test effectiveness.

4.3 Evaluation of test effectiveness

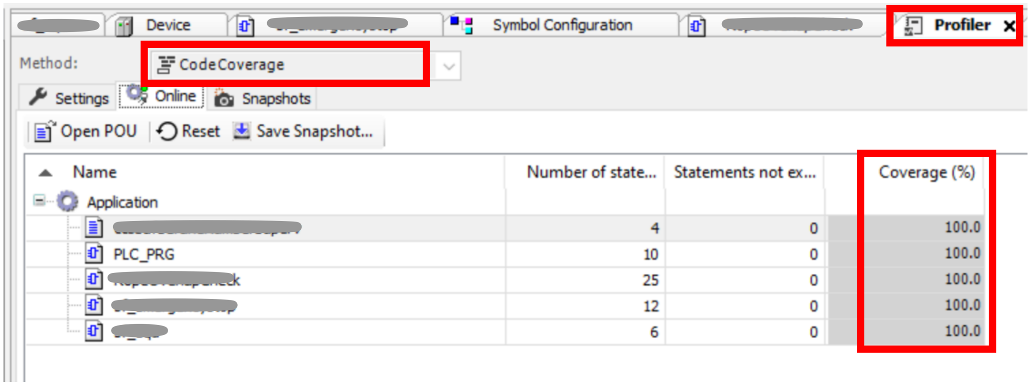
The tests are executed on the PLC FBD program running on a standard installation of CODESYS V3.5 SP16 Patch 4. To measure the code coverage at module level, `CODESYS Profiler 1.3.1.0` is used. Figure 4.4 shows the code coverage measurements from CODESYS Profiler during the test execution on the SUT.



Method: CodeCoverage

Name	Number of state...	Statements not ex...	Coverage (%)
Application			
[redacted]	4	1	75.0
PLC_PRG	10	0	100.0
[redacted]	25	0	100.0
[redacted]	12	0	100.0
[redacted]	6	0	100.0

(a) Code Coverage= 75%



Method: CodeCoverage

Name	Number of state...	Statements not ex...	Coverage (%)
Application			
[redacted]	4	0	100.0
PLC_PRG	10	0	100.0
[redacted]	25	0	100.0
[redacted]	12	0	100.0
[redacted]	6	0	100.0

(b) Code Coverage= 100%

Figure 4.4: Code coverage measurement using CODESYS Profiler

When the test inputs are executed on the SUT running on CODESYS via the OPC UA data exchange protocol, the CODESYS Profiler measures the percentage of code executed in the PLC program. CODESYS Profiler shows the number of statements covered and the statements that are not executed along with the percentage of code coverage achieved during test execution.

In this experiment, the initial code coverage achieved for one module was 75% while the location coverage in Yggdrasil was 94%. The code coverage measurement is used to check the adequacy of generated tests. Upon regenerating the test cases from Yggdrasil with a location coverage 100%, the code coverage achieved for the PLC program under test is 100%.

4.4 Advantages and disadvantages of using UPPAAL Yggdrasil for test generation

For our approach we have used Yggdrasil tool for generating tests from UPPAAL models. As any other tool, Yggdrasil comes with some advantages and disadvantages, which we discuss in the following.

Advantages: Yggdrasil provides the option to generate test cases from traces created by verification queries. This feature, along with the possibility of entering test code on either a transition or entry or exit of the location, gives the advantages of generating tests for specific test purposes. It also serves to generate targeted test cases for locations that remain uncovered during test execution. As the tester only need to dump test code once and the tool generates tests automatically for repeating the testing process, the testing effort s reduced considerably. The tool's design enables the tester to generate tests that provide good structural coverage of the behavioral model.

Disadvantages: There is limited documentation available describing the features of the tool. The tool needs the tester to design the tests and dump the skeleton test code for automatic test generation. When the test cases are generated by selecting multiple options such as query file, depth search, and single step, it is difficult to map the generated tests to the corresponding requirements. While testing the model for its timing properties, the specific properties defined using timing constraints are not reflected in the test cases. Further more, based on our trials, Yggdrassil can only generate test from reachability queries and not from the other ones. It is a limitation while testing the TON function block in a PLC FBD program.

5. Conclusions and Future work

This chapter summarizes the work done, recall the research questions, and discuss the future work.

5.1 Conclusions

In this thesis, a toolchain is presented for automatic test generation based on model checking to test PLC industrial control software systems. The approach considered the SUT as a black box, and the behavioral model of the system is built based on the requirements specification. Model-based testing techniques using the tool UPPAAL and `Pytest` testing framework is used to test PLC program written in IEC 61131-3 FBD programming language. The UPPAAL model checking tool is used for modeling, simulation and verification of the functional and safety properties of the system. The safety properties of the model are verified before generating the tests. The verification of safety properties and timing constraints allows to check the correctness of the behavior model and allows to modify the model if it fails the verification. The Python tests are designed to represent the read-execute-write cyclic task behaviour of the POU in a PLC application.

The study results show that model checking of the behavioral model allows to verify that the model satisfies the safety requirements in the specification. The proposed approach for automated test generation and execution approach is also significantly faster than the manual testing process. The test execution is performed offline on the SUT using the test suite generated automatically and achieved a code coverage of 100% in $124s$.

The experiment results also suggest that the proposed toolchain for automatic test generation using model checking can be adopted in the industry for automation testing of ICS systems. The toolchain is suited for industrial adoption as it uses the open-source testing framework `pytest` and OPC UA as a communication protocol, which is already supported in CODESYS IDE for IEC 61131-3 PLC programming languages.

5.2 Future work

In this thesis, a toolchain is presented for automatic test generation using the model-based testing technique to verify PLC ICS systems. The system is mainly verified for its reachability, liveness, and safety properties. There are several tools for automatic test generation such as COMPLETETEST [40], CPTest+ [63], and FPCCTestGen [64]. There is a plan to explore these tools in the future and compare them to our methodology.

Even though automatic test generation using model checking effectively reduces human effort and enables faster test suite generation, its effectiveness in terms of fault detection capabilities needs to be evaluated. Research on automatic test generation for PLC ICS systems based on model-based mutation testing is also of interest. There are mutant generation tools such as ECDAR [65], `MuUTA` [66] for the generation of mutants from a model using selected mutation operators. It is also of interest to investigate the best-suited mutation operators for PLC FBD programs that can improve the fault detection capability of the model-based mutation testing in comparison to manual testing.

The test execution is performed offline on the application program running on the SoftPLC system in the current study. Online testing of the application program running on PLC hardware will be part of future work.

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957212.

References

[1] K. Stouffer, J. Falco, and K. Scarfone, "Guide to industrial control systems (ics) security," *NIST special publication*, vol. 800, no. 82, pp. 16–16, 2011.

[2] *Industrial Control System*, 2021. [Online]. Available: `https://www.trendmicro.com/vinfo/us/security/definition/industrial-control-system/`.

[3] E. R. Alphonsus and M. O. Abdullah, "A review on the applications of programmable logic controllers (plcs)," *Renewable and Sustainable Energy Reviews*, vol. 60, pp. 1185–1205, 2016.

[4] B. Krebs, "Cyber incident blamed for nuclear power plant shutdown," *Washington Post, June*, vol. 5, no. 2008, p. 5, 2008.

[5] B. Kesler, "The vulnerability of nuclear facilities to cyber attack," *Strategic Insights*, vol. 10, no. 1, pp. 15–25, 2011.

[6] H. Park, "Detailed analysis report for stuxnet," *IBM Korea*, 2010.

[7] B. Lim, D. Chen, Y. An, Z. Kalbarczyk, and R. Iyer, "Attack induced common-mode failures on plc-based safety system in a nuclear power plant: Practical experience report," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, IEEE, 2017, pp. 205–210.

[8] H.-s. Eom, G.-y. Park, S.-c. Jang, H. S. Son, and H. G. Kang, "V&v-based remaining fault estimation model for safety–critical software of a nuclear power plant," *Annals of Nuclear Energy*, vol. 51, pp. 38–49, 2013.

[9] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of plc software," *Software & Systems Modeling*, vol. 15, no. 4, pp. 937–960, 2016.

[10] *PLCopen org*, 2021. [Online]. Available: `https://plcopen.org/what-plcopen`.

[11] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010.

[12] S. Brown, “Overview of iec 61508. design of electrical/electronic/programmable electronic safety-related systems,” *Computing & Control Engineering Journal*, vol. 11, no. 1, pp. 6–12, 2000.

[13] M. Vuori, H. Virtanen, J. Koskinen, and M. Katara, “Safety process patterns in the context of iec 61508-3,” 2011.

[14] *V&V IEC-61508-3:2010*, 2021. [Online]. Available: `https://assets.vector.com/cms/content/products/VectorCAST/Docs/Whitepapers/English/Understanding_Verification_Validation_of_Software_Under_IEC-61508_v2.0.pdf`.

[15] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[16] V. R. Basili and B. T. Perricone, “Software errors and complexity: An empirical investigation0,” *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[17] D Alexandre and G. Kim, *A tutorial on uppaal, formal methods for the design of real-time systems: 4th intern school on formal methods for the design of computer, comm and software systems. sfm-rt 2004, no 3185 in lncs*, 2004.

[18] J. Hunt, “Pytest testing framework,” in *Advanced Guide to Python 3 Programming*, Springer, 2019, pp. 175–186.

[19] E. P. Enoiu, D. Sundmark, and P. Pettersson, “Model-based test suite generation for function block diagrams using the uppaal model checker,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, IEEE, 2013, pp. 158–167.

[20] S.-H. Leitner and W. Mahnke, “OPC UA–service-oriented architecture for industrial applications,” *ABB Corporate Research Center*, vol. 48, pp. 61–66, 2006.

[21] *CODESYS Profiler*, 2021. [Online]. Available: `https://store.codesys.com/codesys-profiler.html`.

[22] W Bolton, *Programmable logic controllers, newnes*, 2009.

[23] D. H. Hanssen, *Programmable logic controllers: a practical approach to IEC 61131-3 using CODESYS*. Wiley Online Library, 2015.

[24] M. H. Schwarz and J. Börcsök, “A survey on OPC and OPC-UA: About the standard, developments and investigations,” in *2013 XXIV International Conference on Information, Communication and Automation Technologies (ICAT)*, IEEE, 2013, pp. 1–6.

[25] *CODESYS OPC UA*, 2020. [Online]. Available: `https://www.codesys.com/products/codesys-runtime/opc-ua.html` (visited on 05/18/2021).

[26] *CODESYS Online help*, 2021. [Online]. Available: `https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_runtime_opc_ua_server/` (visited on 05/18/2021).

[27] O. Roulet-Dubonnet, *Python OPC-UA Documentation*, 2021. [Online]. Available: `https://python-opcua.readthedocs.io/en/latest/index.html`.

[28] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[29] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, "Model-based security testing: A taxonomy and systematic classification," *Software Testing, Verification and Reliability*, vol. 26, no. 2, pp. 119–148, 2016.

[30] F. Siavashi and D. Truscan, "Environment modeling in model-based testing: Concepts, prospects and research challenges: A systematic literature review," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, 2015, pp. 1–6.

[31] O. Pavlovic and H.-D. Ehrich, "Model checking plc software written in function block diagram," in *2010 Third International Conference on Software Testing, Verification and Validation*, IEEE, 2010, pp. 439–448.

[32] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, "Model checking flight control systems: The airbus experience," in *2009 31st International Conference on Software Engineering-Companion Volume*, IEEE, 2009, pp. 18–27.

[33] S Lampérière-Couffin and J.-J. Lesage, "Formal verification of the sequential part of plc programs," in *Discrete Event Systems*, Springer, 2000, pp. 247–254.

[34] E. Jee, S. Jeon, S. Cha, K. Koh, J. Yoo, G. Park, and P. Seong, "Fbdverifier: Interactive and visual analysis of counter-example in formal verification of function block diagram," *Journal of Research and Practice in Information Technology*, vol. 42, no. 3, pp. 171–188, 2010.

[35] J. Yoo, S. Cha, and E. Jee, "A verification framework for fbd based software in nuclear power plants," in *2008 15th Asia-Pacific Software Engineering Conference*, IEEE, 2008, pp. 385–392.

[36] A. Guignard, J.-M. Faure, and G. Faraut, "Model-based testing of plc programs with appropriate conformance relations," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 350–359, 2017.

[37] B. F. Adiego, E. B. Viñuela, J.-C. Tournier, V. M. G. Suárez, and S. Bliudze, "Model-based automated testing of critical plc programs," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, IEEE, 2013, pp. 722–727.

[38] S. Rösch, D. Tikhonov, D. Schütz, and B. Vogel-Heuser, "Model-based testing of plc software: Test of plants' reliability by using fault injection on component level," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 3509–3515, 2014.

[39] A. Ferrari, A. Fantechi, G. Magnani, D. Grasso, and M. Tempestini, "The metrô rio case study," *Science of Computer Programming*, vol. 78, no. 7, pp. 828–842, 2013.

[40] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson, "Automated test generation using model checking: An industrial evaluation," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 335–353, 2016.

[41] B. F. Adiego, D. Darvas, J.-C. Tournier, E. B. Viñuela, J. O. Blech, and V. G. Suárez, "Automated generation of formal models from st control programs for verification purposes," *CERN, Internal Note CERN-ACC-NOTE-2014-0037*, 2014.

[42] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *International Conference on Computer Aided Verification*, Springer, 2002, pp. 359–364.

[43] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE software*, vol. 28, no. 3, pp. 41–48, 2011.

[44] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[45] K. Sacha, "Verification and implementation of dependable controllers," in *2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*, IEEE, 2008, pp. 143–151.

[46] H. B. Mokadem, B. Berard, V. Gourcuff, O. De Smet, and J.-M. Roussel, "Verification of a timed multitask system with uppaal," *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 4, pp. 921–932, 2010.

[47] D. Soliman and G. Frey, "Verification and validation of safety applications based on plcopen safety function blocks," *Control engineering practice*, vol. 19, no. 9, pp. 929–946, 2011.

[48] G. Frey and L. Litz, “Formal methods in plc programming,” in *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.’cybernetics evolving to systems, humans, organizations, and their complex interactions’(cat. no. 0*, IEEE, vol. 4, 2000, pp. 2431–2436.

[49] S. Klein, G. Frey, and M. Minas, “Plc programming with signal interpreted petri nets,” in *International Conference on Application and Theory of Petri Nets*, Springer, 2003, pp. 440–449.

[50] E. M. Clarke and J. M. Wing, “Formal methods: State of the art and future directions,” *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[51] G. J. Holzmann, “The model checker spin,” *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[52] M. Leuschel and M. Butler, “Prob: A model checker for b,” in *International symposium of formal methods europe*, Springer, 2003, pp. 855–874.

[53] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[54] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, “Testing real-time systems using uppaal,” in *Formal methods and testing*, Springer, 2008, pp. 77–117.

[55] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on uppaal 4.0,” *Department of computer science, Aalborg university*, 2006.

[56] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, “Symbolic model checking for real-time systems,” *Information and computation*, vol. 111, no. 2, pp. 193–244, 1994.

[57] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, “Uppaal—a tool suite for automatic verification of real-time systems,” in *International hybrid systems workshop*, Springer, 1995, pp. 232–243.

[58] D. Truscan, T. Ahmad, F. Siavashi, and P. Tuuttila, “A practical application of uppaal and dtron for runtime verification,” in *2015 IEEE/ACM 2nd International Workshop on Software Engineering Research and Industrial Practice*, IEEE, 2015, pp. 39–45.

[59] J. H. Kim, K. G. Larsen, B. Nielsen, M. Mikučionis, and P. Olsen, “Formal analysis and testing of real-time automotive systems using uppaal tools,” in *International Workshop on Formal Methods for Industrial Critical Systems*, Springer, 2015, pp. 47–61.

[60] F. Siavashi, D. Truscan, and J. Vain, "On mutating uppaal timed automata to assess robustness of web services.," in *ICSOFT-EA*, 2016, pp. 15–26.

[61] K. G. Larsen, F. Lorber, and B. Nielsen, "20 years of uppaal enabled industrial model-based validation and beyond," in *International Symposium on Leveraging Applications of Formal Methods*, Springer, 2018, pp. 212–229.

[62] Z. Gu, M. Yuan, and X. He, "Tutorial for the uppaal model generater,"

[63] M. Jamro, "Pou-oriented unit testing of iec 61131-3 control software," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 5, pp. 1119–1129, 2015.

[64] Y.-C. Wu and C.-F. Fan, "Automatic test case generation for structural testing of function block diagrams," *Information and Software Technology*, vol. 56, no. 10, pp. 1360–1376, 2014.

[65] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman, "Mutation-based test-case generation with ecdar," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2017, pp. 319–328.

[66] F. Siavashi, J. Iqbal, D. Truscan, and J. Vain, "Testing web services with model-based mutation," in *International Conference on Software Technologies*, Springer, 2016, pp. 45–67.