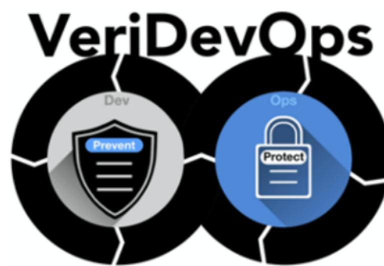


VeriDevOps

Automated Protection and Prevention to Meet Security

Requirements in DevOps Environments

D4.4 Tools for prevention at design level - final version

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957212. This document reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

Contract number:	957212
Project acronym:	VeriDevOps
Project title:	Automated Protection and Prevention to Meet Security Requirements in DevOps Environments
Delivery Date:	30.3.2023
Coordinator:	ABO
Partners contributed:	MDU, MI, SOFT
Release Date:	30.3.2023
Version:	03
Abstract:	This deliverable is an updated version of D4.1 and discusses the final set of methods and tools for creating secure-by-design specifications in the VeriDevOps project. These specifications will be created using formal and semi-formal languages and different security properties stemming from security requirements will be verified. The tools discussed in this deliverable use as input the formal security requirements specification described in deliverable D2.1 and will provide input to the technologies for test generation discussed in Deliverable D4.5.
Status:	• PU (Public)

Revision History

VERSION	DATE	DESCRIPTION	AUTHOR
04	29/03/2023	Review comments addressed	ABO
03	20/03/2023	Version Released for internal review	ABO
02	01/03/2023	Partner contributions updated	VeriDevOps consortium
01	10/11/2022	Initial version created from D4.1	ABO

Executive Abstract

This deliverable overviews the final list of methods and tools for creating secure-by-design specifications in the VeriDevOps project. These specifications will be created using formal and semi-formal languages and different security properties stemming from security requirements will be verified. The tools discussed in this deliverable use as input the formal security requirements specification described in deliverable D2.1 and will provide input to the technologies for test generation discussed in Deliverables D4.2 and D4.5.

The initial version of this document, namely Deliverable D4.1, was released on 30.06.2021. The current version is an update of the previous one, by adding new methods and tools, updating the description of the previous ones. For each presented tool we also discuss the perceived benefits and drawbacks of applying the tool in the project. The hypotheses on the perceived benefits of applying the tools and the drawbacks of not applying any of the chosen tools in the project will be validated in the later stages of the project by evaluating the tools on the two use cases of VeriDevOps.

Table of Contents

Revision History 2
Executive Abstract 2
Table of Contents 3
1. Introduction 5
2. UPPAAL model-checking tool suite 7
2.1. General description 7
2.1.1. UPPAAL timed automata 7
2.1.2. Tool features 8
2.1.3. Relation to VeriDevOps and use cases 15
2.1.4. How to get it, install it, licensing 15
2.2. New compared to D4.1 16
3. CompleteTest - Model Generation and Vulnerability Detection using Model Checking 16
3.1. General description 16
3.2. Relation to VeriDevOps and CaseStudies 19
3.3. Detailed overview for relevant usage scenarios 19
3.4. How to get it, install it, licensing 20
3.5. New compared to D4.1 20
4. PyLC [new] 21
4.1. General description 21
4.2. Relation to VeriDevOps and CaseStudies 25
4.3. Detailed overview for relevant usage scenarios 27
5. GW2UPPAAL [new] 28
5.1. General description (purpose, features, interfaces) 28
5.2. Relation to VeriDevOps and CaseStudies 30
5.3. Detailed overview for relevant usage scenarios 30
5.4. How to get it, install it, licensing 31
6. Modelio 32
6.1. General description 32
6.2. Relation to VeriDevOps and CaseStudies 34
6.3. Detailed overview for relevant usage scenarios 35
6.4. How to get it, install it, licensing 36
6.4.1. Modelio Open Source 36
6.4.2. Modelio Commercial 37

6.5. New compared to D4.1 37
7. Conclusions **37**
References **38**

1. Introduction

In the context of VeriDevOps, security specifications play a central role in generating different artifacts needed later on for prevention at development in WP4. The first deliverable of this WP, namely D4.1 [1], listed the initial methods and tools for creating system specifications that satisfy the security properties imposed by security requirements. This deliverable, D4.4, complements deliverable D4.1 with additional technologies that have been developed and applied in the context of the industrial use cases.

Figure 1 shows the list of tools used for prevention at design level and how they rely on the security requirements produced in WP2. The tools are color-coded based on their availability: in green tools already introduced in D4.1 and are still in use, in red tools that have been introduced in D4.1 and discontinued, and in blue the tools that are new in this deliverable. These tools use formal or semi-formal semantics to specify the expected behavior of the system under test and to verify that the specification satisfies the security requirements. The input for these tools is provided in the form of *formalized security requirements* developed in WP2. These formalized security requirements are created from textual-based requirements using the methods and tools discussed in detail in deliverable *D2.5 Specification Verification Tool Set - initial version* (submitted May 2022). In this project, we focus on three formalisms to represent formalized security requirements: Timed Computational Tree Logic (TCTL), Seamless Object Oriented Requirements (SOOR) and, respectively, Organization-based access control (OrBac) rules. These requirements are used to enforce the specifications of the system which are later on used in this work package 4 for security test generation and execution.

We present an extension to the model-based analysis and testing approach using UPPAAL. Specifically, a new tool called GW2UPPAAL has been developed to automate the modeling step which significantly reduces the time required for verifying the created models and their security requirements. The results indicate that the use of GW2UPPAAL offers significant benefits in terms of time efficiency and model verification for security requirements. Secondly, we have extended our CompleteTest approach for security test generation and program analysis. Specifically, we focus on addressing potential vulnerabilities in Function Block Diagram (FBD) programs. We propose a method for addressing potential security requirements at the code level, such as input validation, prevention of out-of-bound data, and prevention of false negatives and false positives. The goal is to ensure that FBD programs do not contain vulnerabilities that could be exploited by attackers or intruders. Thirdly, The Modelio tool has been updated to use the latest version of Java.

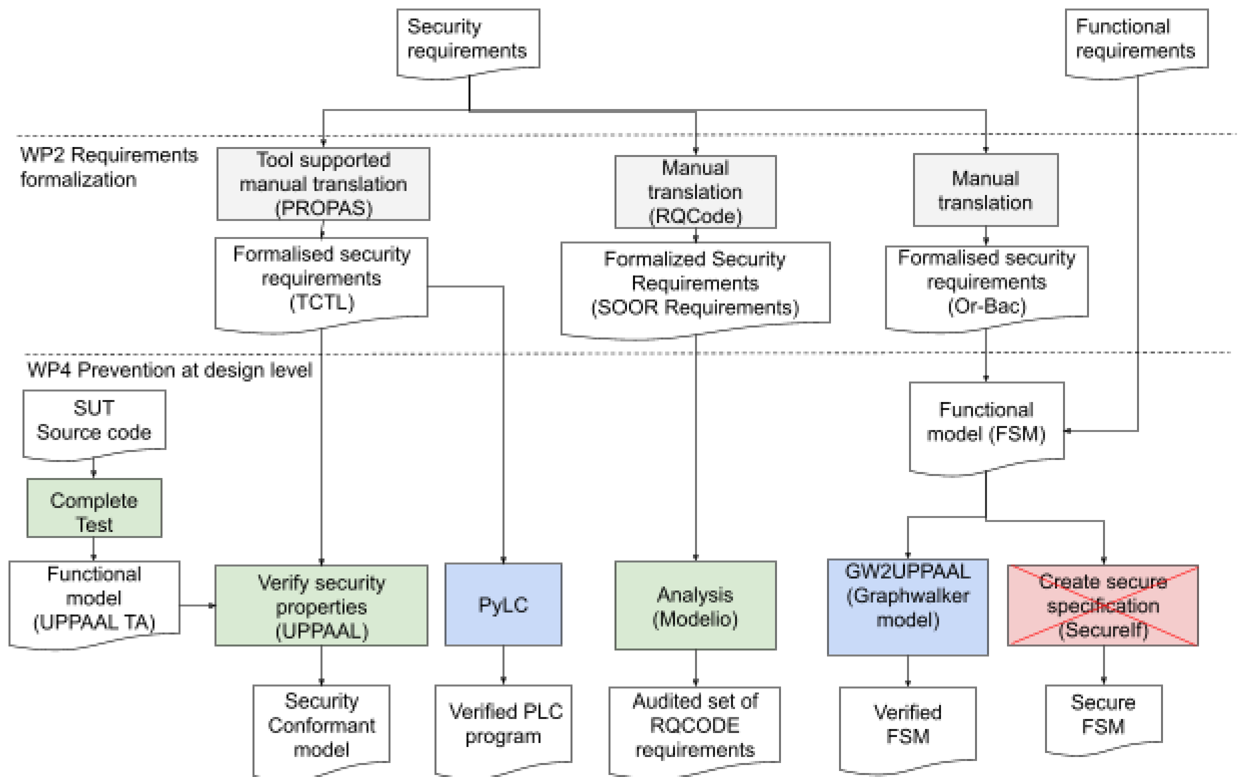


Figure 1. General overview of the usage of the tools used for prevention at design level.

We propose a new framework, called PyLC, that is capable of transforming PLC programs into Python code and generating tests that can effectively validate the transformed code. The results indicate that the framework helps in improving the testing and validation of safety-critical industrial PLC programs. Further, we present a new approach for automated analysis of a GraphWalker (GW) model by transforming it into a UPPAAL model and generating queries that are verified. The results of the preliminary evaluation demonstrate that it takes less time to create a model in GW and transform it into a UPPAAL model than creating a model directly in the UPPAAL GUI tool. Lastly, we point out that the SecureIF tool, proposed in Deliverable D4.1, has been discontinued due to its unsuitability for the case study artifacts available in this project. The tool can generate a secure specification of a system by combining security requirements formalized as Organisation-based access control (OrBac) rules with a behavioral specification of the system extracted from functional requirements in the form of an Extended Finite State Machine (EFSM). The tool will not be further described in this deliverable; however, one can find more details about it in Deliverable D4.1.

In the next sections, we overview in more detail the relevant features for creating secure-by-design specifications for each of the tools mentioned above as well as the benefits of using them within our approach and the dangers or pitfalls of not using them. We discuss the planned use in the context of the VeriDevOps use cases and we provide details on the licensing and how the tools can be employed in design. For each presented tool we also discuss the perceived benefits and drawbacks of applying the tool in the project. The hypotheses on the perceived benefits of these tools will be validated in the later stages of the project by evaluating the tools on the ABB and FAGOR use cases. The results of the evaluation will be presented in the deliverables of WP5.

2. UPPAAL model-checking tool suite

2.1. General description

The UPPAAL model-checking tool suite is an integrated environment for modeling, validation, and verification of real-time systems [2]. It uses a network of extended Timed Automata, called UPPAAL Timed Automata, to specify the behavior of the system. Although UPPAAL is not a contribution of this project, we decided to present it here since it is used for verification by the CompleteTest tool (as shown in Figure 1) and it can be potentially used by other tools (e.g., GW2UPPAAL and CompleteTest).

2.1.1. UPPAAL timed automata

A *timed automaton* (TA) is essentially a finite automaton (that is, a graph containing a finite set of nodes called locations and a finite set of labeled edges) extended with real-valued variables [3]. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, are initialized with zero when the system is started, and then incremented synchronously with the same rate. The behavior of the automaton is restricted by using clock constraints on edges. A transition represented by an edge can be taken when the clock values satisfy the guard which labels the edge. The clocks may be reset to zero when a transition is taken.

UPPAAL timed automata (UPPAAL TA) are an extension of timed automata with bounded integer variables and simple data types (aka, TA with data variables) [4]. The specification of a system using UPPAAL TA is defined as a closed network of extended timed automata that are called processes. The processes are combined into a single system by synchronous parallel composition like in process algebra. The state of an automaton consists of its current location and assignments to all variables, including clocks. Synchronous communication between processes is expressed by synchronization variables called channels. A channel ch relates a

pair of transitions in parallel processes where synchronized edges are labeled with symbols for input and output actions (denoted *ch?* and *ch!*, respectively)

An example of two UPPAAL TA is shown in Figure 2. This example is part of a railway control system distributed with UPPAAL. In this system, several trains can have access to a bridge but only one train can cross it at a given time. The trains require a certain time to stop and restart. The trains are modeled using the template in Figure 2-left. When approaching the bridge a train sends the *appr[id]!* notification, where id is the number of the train. After that it has 10 time units to receive a stop[id]? signal which allows it to stop safely, otherwise it will proceed to crossing the bridge. If a *stop[id]?* was received, the train will stop until a *go[id]?* signal is received and then it will start moving. The gate automaton (Figure 2-right) specifies the behavior of the gate: whenever a train signals its approach to the bridge via *appr[e]?* it is added to a queue. If the bridge is free the first train in the queue is allowed to pass via the *go!* signal and the bridge is occupied. When a train has left the bridge it notifies the gate controller via *leave[e]* and it is removed from the queue [2].

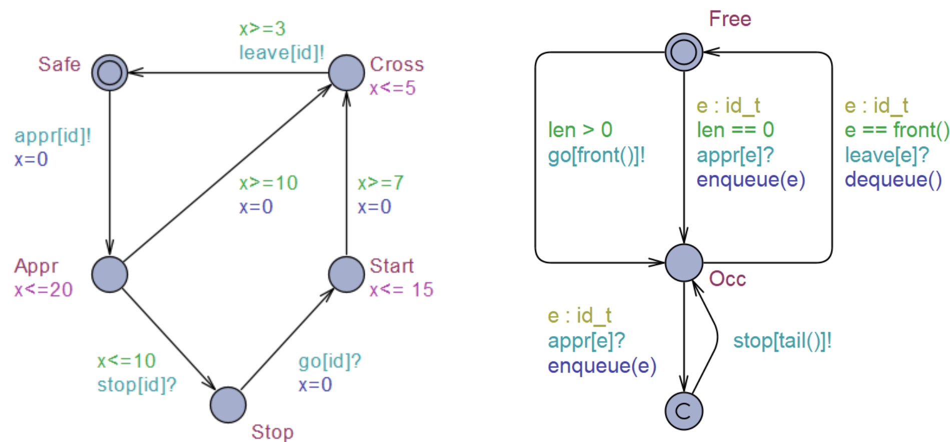


Figure 2. UPPAAL TA of a train from the train gate example [1]: left - Train TA and right- Gate TA

and it can be used to verify whether given properties of the system, specified in temporal logic, are violated or not. In the former case, a counter example (aka diagnostic trace) is presented and can be visualized in the UPPAAL simulator (described in the next section).

2.1.2. Tool features

The UPPAAL tool has a graphical user interface that includes a specification editor, a graphical symbolic simulator and verification tool.

The **UPPAAL editor tab** (Figure 3) allows one to specify the processes of the system as templates and to define shared and local variables, as well as functions. The example in Figure 3 shows the automaton of

the Train template. In addition, the editor provides syntactic consistency checks of the model to avoid common modeling mistakes.

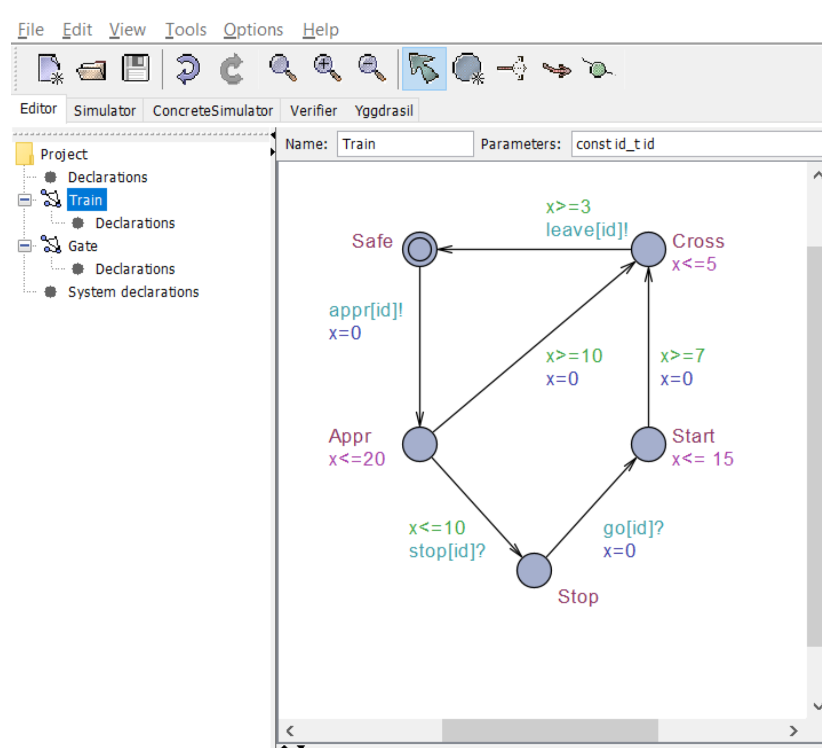


Figure 3. UPPAAL editor screenshot

The **Simulator tab** (Figure 4) in the graphical interface is a validation tool to examine possible dynamic executions of the system during the modeling stage and to visualize the diagnostic trace of executions generated by the verifier. The example in Figure 4 shows the train gate system in which the automata in Figure 2 have been instantiated into six train processes and a gate process.

The simulator provides several panels, as follows:

- currently enabled transitions from which one can manually select the next steps to be executed
- a trace panel recording which transitions have been executed in the model and the corresponding states
- a simulation control panel from where the simulation can be run automatically based on random choice made by the tool according to the enabled transitions
- variables panel listing the global and local variables and their values in each state
- a process panel showing all the processes of the system, the current locations for each process and the edges taken during a transition
- and a message sequence chart panel showing the order of synchronizations exchanged between different processes.

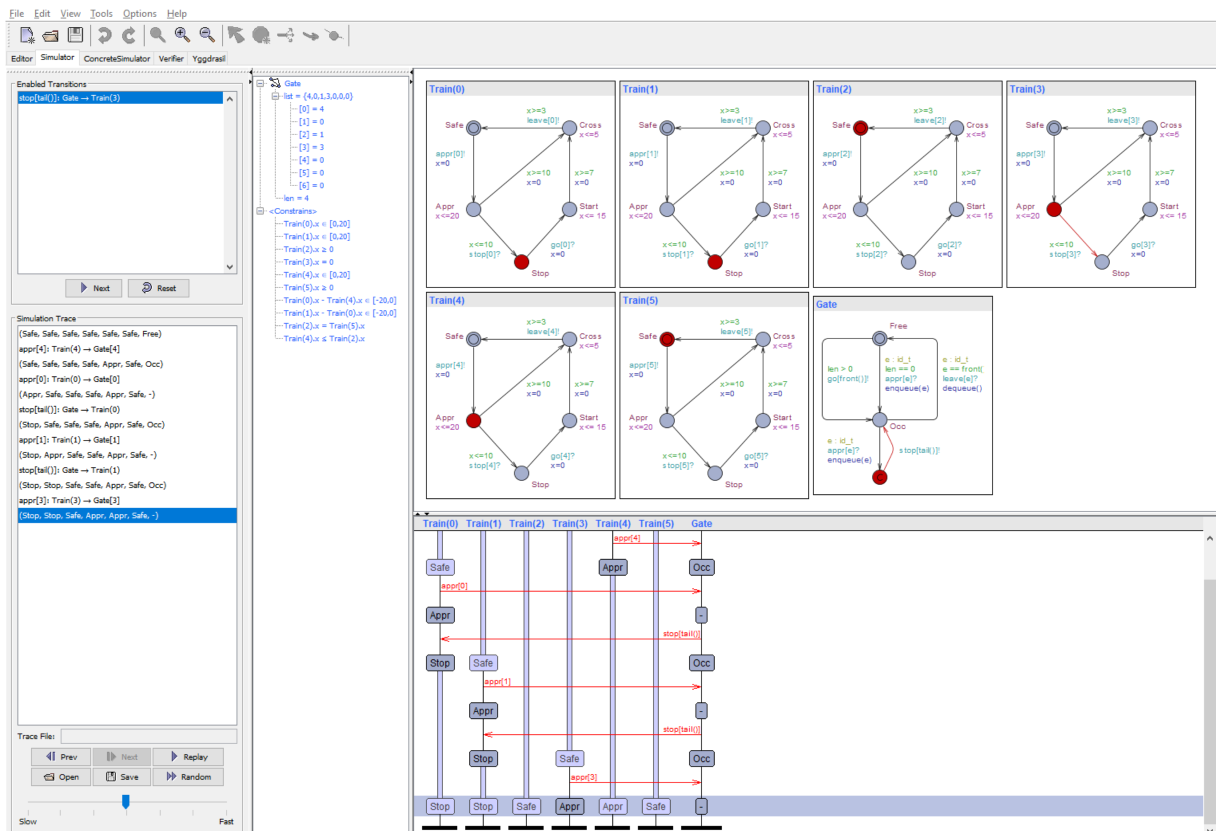


Figure 4: Symbolic simulator tab of UPPAAL

The third tab is a **concrete simulator** that was originally used in UPPAAL-Tiga [5]. This simulator allows the user to simulate a system with concrete values of clocks, which is more intuitive than with the symbolic simulator. This simulator is shown in Figure 5.

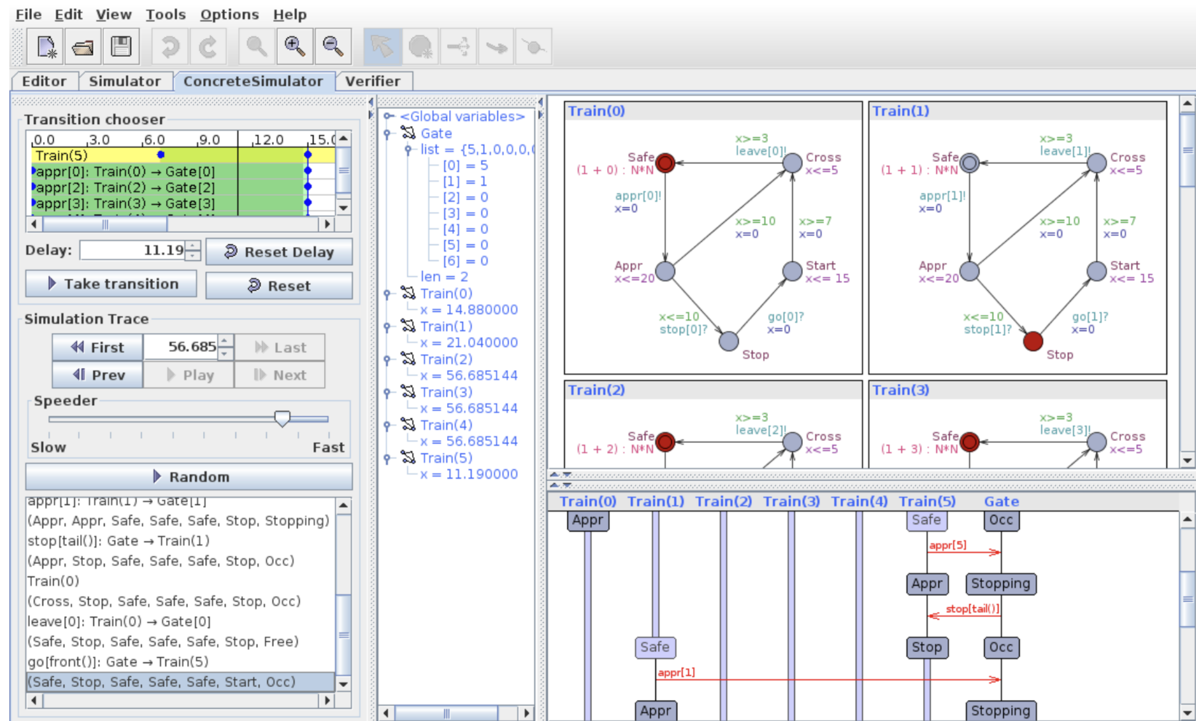


Figure 5. The concrete simulator in UPPAAL

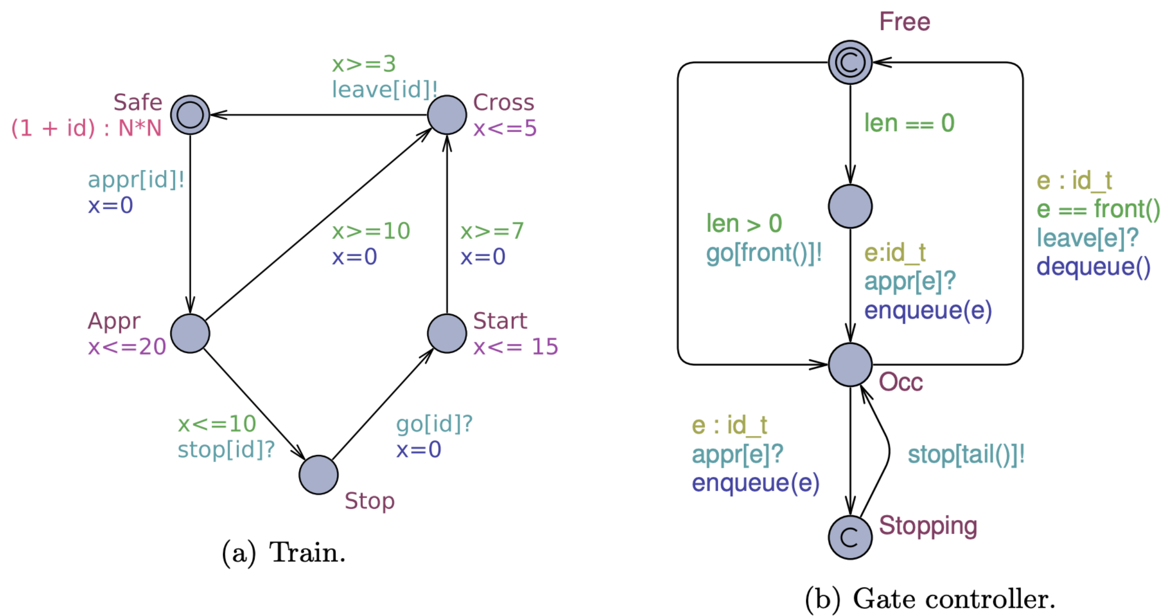
UPPAAL also provides a **model checking feature**, either via the graphical user interface (Verifier tab) (Figure 6) or as a command line tool, called *verifyTA*. The query language used in UPPAAL is a subset of timed computation tree logic (TCTL) [6]. It consists of state formulae to verify the states and path formulae to check safety, liveliness, and reachability properties using paths of the model.

In case the verification of a property fails, the verifier can produce a counter-example trace showing the sequence of states and transitions in the model that violated the property. The trace can be visualized in the UPPAAL Simulator for further debugging.

```
File  Edit  View  Tools  Options  Help
Editor  Simulator  ConcreteSimulator  Verifier  Yggdrasil

Overview
E<> Gate.Occ
E<> Train(0).Cross
E<> Train(1).Cross
E<> Train(0).Cross and Train(1).Stop
E<> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop)
A[] forall (i : id_t) forall (j : id_t) Train(i).Cross && Train(j).Cross imply i == j
A[] Gate.list[N] == 0

Check
Insert
Remove
Comments

Query
E<> Gate.Occ

Comment
Gate can receive (and store in queue) msg's from approaching trains.

Status
Established direct connection to local server.
(Academic) UPPAAL version 4.1.19 (rev. 5649), September 2014 -- server.
(SUT.x>=period && input1_var != input2_var) --> output_var==1
Verification/kernel/elapsed time used: 0.016s / 0s / 0.013s.
Resident/virtual memory usage peaks: 7,984KB / 27,036KB.
Property is satisfied.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.1.19 (rev. 5649), September 2014 -- server.
E<> Gate.Occ
Verification/kernel/elapsed time used: 0s / 0s / 0.002s.
Resident/virtual memory usage peaks: 8,104KB / 27,636KB.
Property is satisfied.
```

Figure 6. Verifier GUI of UPPAAL

UPPAAL SMC [7] is an extension of the tool UPPAAL, which supports statistical model checking (SMC) of hybrid automata (HA). Instead of exhaustively exploring the state space of the model, statistical model checking randomly executes the model with respect to a given property and applies statistical analysis to estimate the satisfaction of that property. HA in UPPAAL SMC are similar to UPPAAL TA, and extend the latter with a set of continuous variables whose derivatives are described by ordinary differential equations (ODE). In UPPAAL SMC, the HA have a stochastic interpretation based on: (i) the probabilistic choices between multiple enabled transitions, and (ii) the nondeterministic time delays that can be refined based on probability distributions, either uniform distributions for time-bounded delays or (user-defined) exponential distributions for unbounded delays.



(a) Train.

(b) Gate controller.

Figure 7. UPPAAL SMC train-gate example.

As in UPPAAL, a model in UPPAAL SMC consists of a network of interacting Stochastic Timed Automata (STA) that communicate via broadcast channels and shared variables to generate Networks of Stochastic Timed Automata (NSTA). Figure 7 shows the NSTA template for a train and a gate controller. UPPAAL SMC allows the user to specify an arbitrary (integer) rate for the clocks on any location. In addition, the automata support branching edges where weights can be added to give a distribution on discrete transitions. It is important to note that rates and weights may be general expressions that depend on the states and not just simple constants. We mention here that there are timing constraints for stopping the trains in which it is not possible to stop trains instantly. The interesting point in SMC is to define the arrival rates of these trains. The location Safe has no invariant and defines the rate of the exponential distribution for delays (trains delay according to this distribution).

UPPAAL SMC supports an extension of weighted metric temporal logic for probability estimation, whose queries are formulated as follows: Pr[bound] (ap), where bound is the simulation time, ap is the formula that supports two temporal operators: “Eventually” (<>) and “Always” ([]). Such queries estimate the probability that a property is satisfied within the simulation time bound. Probability comparison (Pr[bound](ψ1) ≥ Pr[bound](ψ2)) and hypothesis testing (Pr[bound](ψ) ≥ p0) are also supported. Figure 8 shows the verifier of UPPAAL SMC and Figure 9 shows how it generates the cumulative probability distribution of an example property using statistical model checking.

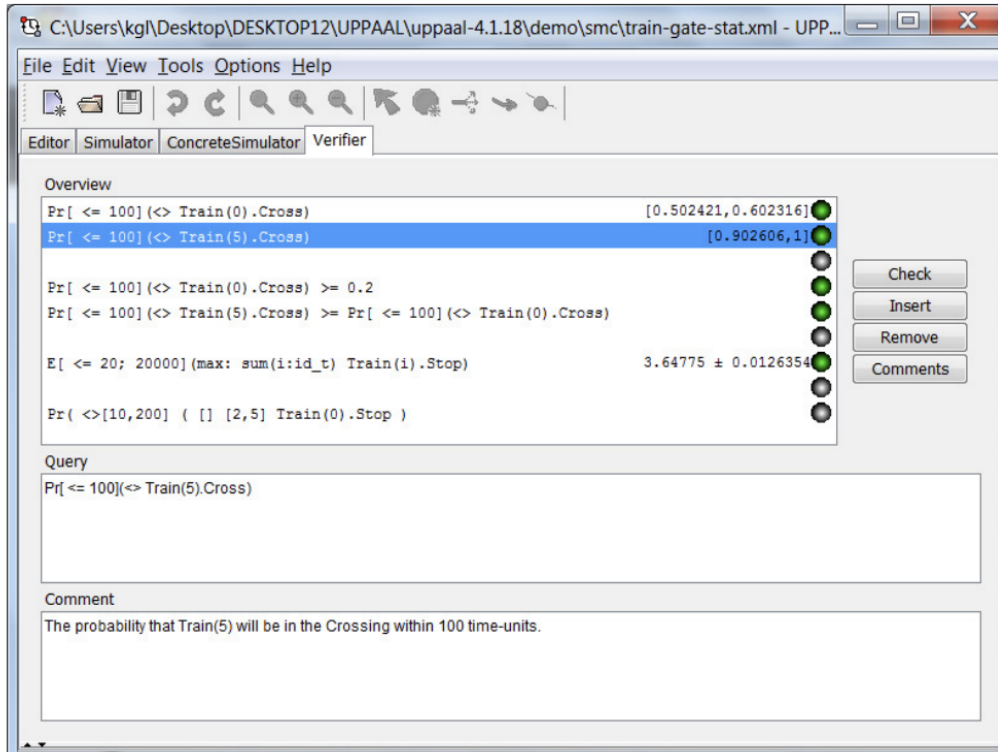


Figure 8. The Verifier of UPPAAL SMC and generating the cumulative probability distribution of an example property using statistical model checking.

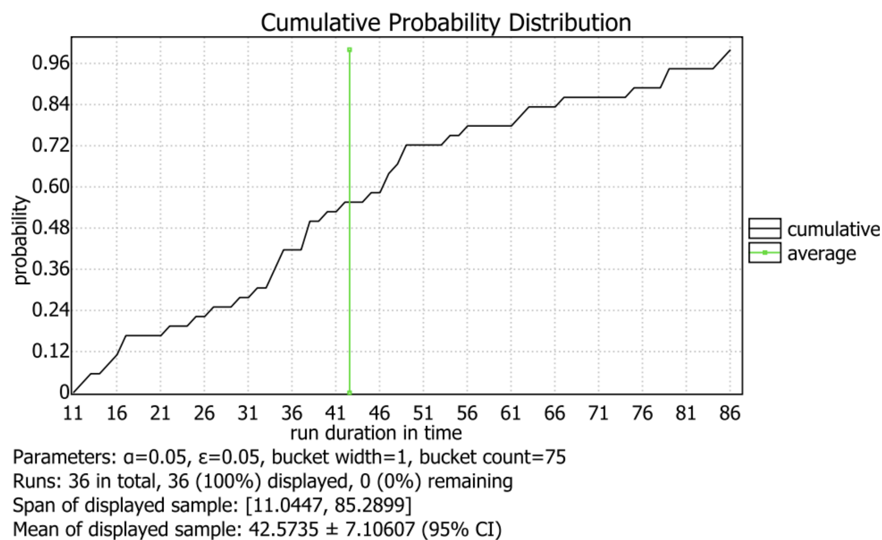


Figure 9. The generation of the cumulative probability distribution of an example property using statistical model checking.

The **benefits** of using UPPAAL SMC over the traditional UPPAAL verification engine include: handling of dynamical behaviors, discrete probabilities, a stochastic interpretation for timed delays and even dynamic process creation. By developing, first, a UPPAAL model, one can, with just a few changes, benefit also from UPPAAL SMC features, and gain statistical and probabilistic measures in addition to firm results. The potential **drawbacks** of using UPPAAL and its extensions lie in the slow learning curve and industrial-grade testing capabilities, as well as in poor scalability.

2.1.3. Relation to VeriDevOps and use cases

UPPAAL and its extensions will be used in this project in conjunction with the CompleteTest and GW2UPPAAL tools to model the functional specification of the system under test and to verify that the specifications satisfies the security requirements imposed by the requirements. The verified specifications and the TCTL queries will be used for security test generation later on in the project as will be detailed in Deliverable D4.2.

The relation to the use cases will be discussed in Section 3.

2.1.4. How to get it, install it, licensing

The UPPAAL toolkit is free for non-commercial applications for academic institutions that deliver academic degrees. UPPAAL 4.1 (development snapshot)[1] is the current development release of the academic version, this build includes UPPAAL SMC. To download and install (or upgrade to) the current version of UPPAAL:

1. Choose the version from the download area.
2. Fill in the license agreement and press the "Accept and Download" button.
3. Download the zip-file containing the installation files.
4. Unzip the downloaded zip-file. This should create a number of files, including: *uppaal.jar*, *uppaal*, and the directories *bin-Linux*, *bin-Win32*, and *demo*. The bin-directories should all contain the two files *server*(.exe) and *verifyta*(.exe) plus some additional files, depending on the platform. The directory demo should contain some demo files with suffixes .xml, and .q.
5. Make sure you have at least Java 11 configured on your system. The UPPAAL GUI will not run without Java installed. Java for Windows and Linux can be downloaded from adoptopenjdk.
6. To run UPPAAL on Linux systems run the startup script named *uppaal*. To run on Windows systems, just double-click the file *uppaal.jar*.

[1] https://uppaal.org/downloads/

2.2. New compared to D4.1

We extended the use of UPPAAL for combining model-based analysis and testing. The results revealed that the time taken by UPPAAL to create and execute the model is negligible compared to a manual transformation. As the complexity of the model increases, the time taken to transform these models manually increases with the number of vertices and edges. In contrast, UPPAAL now is extended by the use of a new tool, called GW2UPPAAL, that can automate this modeling step and can reduce the time needed to verify the created models and the security requirements.

3. CompleteTest - Model Generation and Vulnerability Detection using Model Checking

3.1. General description

CompleteTest [8] is a method in which the model is annotated and the properties to be checked are expressible as a single sequence. In contrast to other approaches, CompleteTest provides an approach to generate test cases for different code coverage criteria that are directly applicable to industrial control IEC 61131-3 software. In CompleteTest, the UPPAAL model-checker is used for automatic test generation based on code and mutation coverage criteria. For a detailed overview of testing with model checkers, we refer the reader to Fraser et al. [9]. One important part of this method is the model generation capability. The rest of the method is used for test generation (as shown in Figure 10).

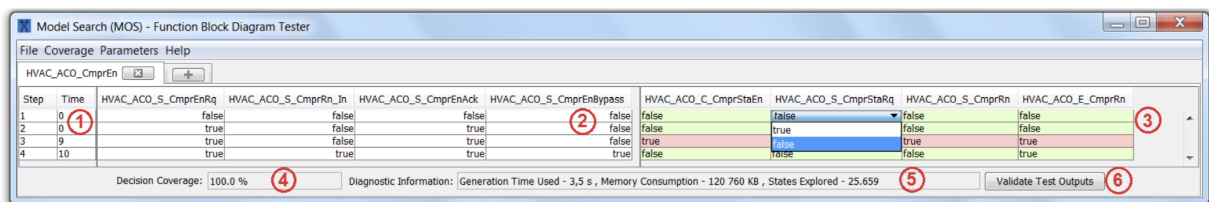


Figure 10. Graphical Interface of the CompleteTest Method and its Toolbox

The translation scheme (from a program to a timed automata model) is included when using the importing function of CompleteTest and is outlined in Figure 11. CompleteTest is automatically calling the UPPAAL model checker for verification purposes. In practice, the timed behavior of a Function Block Diagram (FBD) [10] program is defined as a network of timed automata, extended with data input and output variables. We first perform an automatic transformation of the FBD program to a timed automaton that obeys the read-execute-write semantics of the FBD program, hence preserving

the semantics of FBDs without altering its structure. Next, we specify the execution of each block and construct a complete timed automata model by the parallel composition of local behaviors.

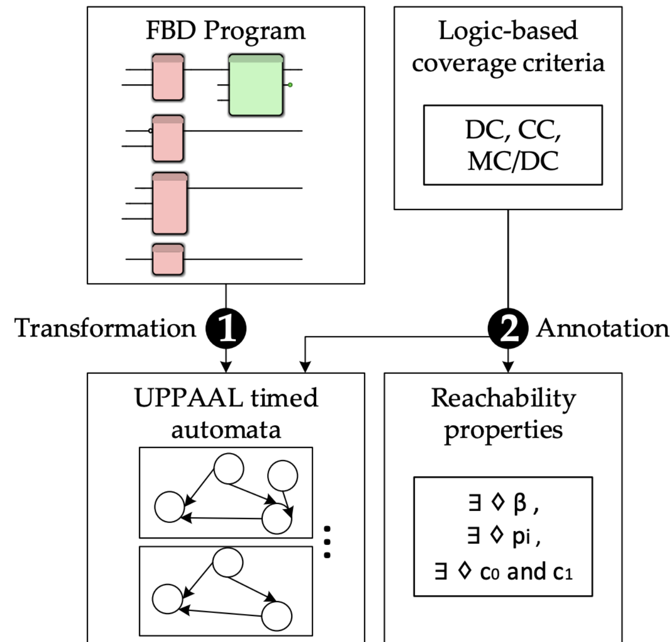


Figure 11. Model Transformation Methodology for CompleteTest.

A generic timed automata network of an example FBD program (Compressor Start Enable program) together with its cycle scan (plcSupervision()) and Input/Output models is shown in Figure 12. To introduce resets in the model, we annotate the cycle scan with a reset transition leading to the initial ReadInputs location. On this transition all variables and parameters (excluding encoded internal variables) are reset to their default value. This reset is hardcoded into the PLC supervision for any modeled FBD program in UPPAAL, being an atomic communication between all timed automata.

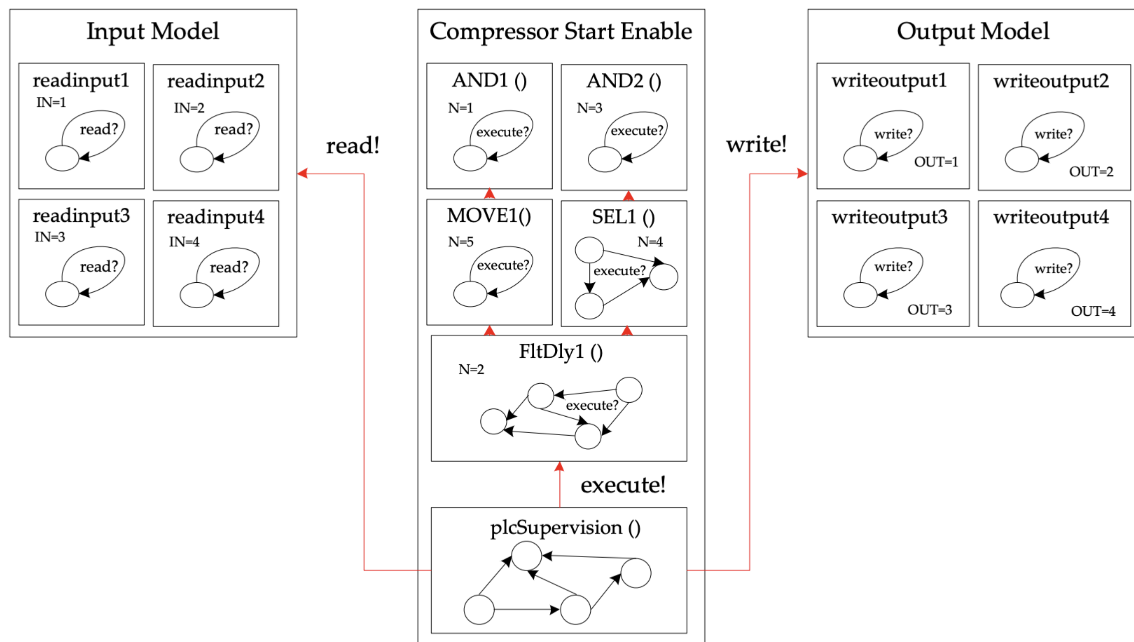


Figure 12. Overview of the Timed Automata Network obtained from CompleteTest for an example FBD program.

Once the model is generated, we can automatically check properties by characterizing a logic coverage criterion as a temporal logic property or any security requirements specified as UPPAAL TCTL properties. We use the PROPAS (The PROperty PAttern Specification and Analysis) tool (described in D2.2 in WP2 and outlined in the overall method in Figure 1) to specify these security requirements from textual descriptions of security threats and vulnerability scenarios. Requirements can be written manually or generated from requirement patterns using PROPAS and then used as formalized security requirements. By using a translated FBD program, we use logic coverage or security requirements related to FBD interfaces to directly annotate both the model and the temporal logic property to be checked. Security requirements obtained from the formalization using PROPAS are used in their TCTL form. For instance, the TCTL property could be: AG(input1 *>10 -> $AF_{<=1}$(SUT.state1)).* Such properties can be written manually or generated from requirement patterns using PROPAS. In the end, we can use such properties in our method to discover vulnerabilities early on when the design is available and can be transformed from the system under test. During design, we propose the annotation with auxiliary data variables and transitions in such a way that a set of paths can be used as a finite test sequence. In addition, we propose to describe the temporal logic properties as logic expressions satisfying certain logic coverage criteria. Informally, our approach is based on the idea that to get logic coverage and security requirements of a specific program, it would be sufficient to (i) annotate the conditions and decisions in the FBD program, (ii) formulate a reachability property for logic coverage and security requirements, and (iii) find a path from the initial state to the end of the FBD program.

3.2. Relation to VeriDevOps and CaseStudies

CompleteTest will be used on the ABB use case by targeting the integration with the CODESYS[2] development environment during design generation for FBD programs. It focuses on prevention at the design level using FBD IEC 61131-3 programs by automatic checking of predefined logic properties related to the interfaces of these FBD programs. As a result of the transformation, we compose such local automata in parallel to a TA. The purpose of the transformation is to construct a target model by filling the TA with the corresponding behavior as explained. Since FBD programs allow the use of behavioral notations, we exploit this and specify the behavior by assigning a TA model to each element mapped from its corresponding FBD program (Scenario ABB_S2). Specifically, the main **benefit** of using this method is to verify the FBD level vulnerabilities using specific test requirements or from various coverage criteria related to security attacks. Without employing such a tool, the **drawback** is that one cannot verify at design time that certain vulnerabilities are not present in the specification or design.

Both FAGOR and ABB use cases are targeted, but it is believed that the majority of case scenarios would benefit from the methods and tools discussed in this Deliverable. The SMC modeling approach will consider the security requirements and policies as well as the control system and the attack model. An example of the **benefits** of using CompleteTest and the UPPAAL model checker for prevention at the design stage is that the verification results may be used to identify the vulnerabilities for possible design improvements and to suggest possible further additions of security constraints w.r.t., e.g., value and range constraints, the dependency between device states and process variables etc. When employing UPPAAL, we need to augment normal system modeling with an environment model to simulate the potential security attacks.

3.3. Detailed overview for relevant usage scenarios

Use Case Scenario 1 - Model Transformation (ABB_S1). To model check an FBD program we map it to a finite state system suitable for model checking. To cope with timing constraints, we have chosen to map FBD programs to timed automata.

Use Case Scenario 2 - Property Annotation and Model Checking (ABB_S2). We annotate the transformed model such that a condition describing a single test case can be formulated. This is a property expressible as a reachability property used in most model checkers.

[2] https://www.codesys.com/products/codesys-engineering/development-system.html

3.4. How to get it, install it, licensing

CompleteTest is an academic tool and it is currently in an early beta version. You can always grab the latest version here and use it freely[3], but only as part of your academic work. Once you have downloaded both the CompleteTest and UPPAAL, extract the *completetest.zip* archive and place *verifyta.exe* from the UPPAAL bin-Win32 folder to *verifyta\bin-Win32* folder of CompleteTest. After you have placed *verifyta.exe* to the correct folder, run the tool either by double-clicking on *CompleteTest.jar* or from a command line by typing:

java -jar CompleteTest.jar

This tool is developed in Java and it requires java version 1.7 to be present on the system. You can always check the version of java you have installed from a command line by typing:

java -version

We suggest having a look at examples located in the samples folder.

3.5. New compared to D4.1

The work on extending CompleteTest towards security test generation and program analysis has progressed. We assume the PLC has been configured for remote access. This allows the threat actor to directly access the PLC and execute malicious commands with no defensive controls to circumvent. This could be enacted by low-skilled threat actors. Attention to FBD code vulnerabilities has not been a great concern as that of network related ones. That is because companies, developers, and programmers assume that the programs that are running within the PLCs are safe and secure as long as there is no network intruder. In the work done to support prevention at design level, we aim to solve the challenge of FBD programs that can carry within their own destructive threats and vulnerabilities that can be exploited by hackers or regular disgruntled users. The vulnerabilities come from the way the code is written or designed. Potential security requirements at code level that are handled using this method:

- input operations can take place only at allowed times
- prevents attackers to execute operations outside the regular flow
- input validation - check unallowed input values are handled
- out of bound data: divide by zero, counter overflow, negative counter or timer preset, I/O scan overrun

[3] https://github.com/eduardenoiu/CompleteTest

- inputs or outputs are those that physically cannot happen at the same time; they are mutually exclusive.
- false negatives and false positives (shortcuts in logic).

4. PyLC [new]

4.1. General description

Many industrial application domains utilize safety-critical systems to implement Programmable Logic Controllers (PLCs) software. These systems typically require a high degree of testing and stringent coverage measurements that can be supported by state-of-the art automated test generation techniques. However, their limited application to PLCs and corresponding development environments can impact the use of automated test generation. Thus, it is necessary to tailor and validate automated test generation techniques against relevant PLC tools and industrial systems to efficiently understand how to use them in practice. In this paper, we present a framework called PyLC, which handles PLC programs written in the Function Block Diagram (FBD) and Structured Text (ST) languages such that programs can be transformed into Python. To this end, we use PyLC to transform industrial safety-critical programs, showing how our approach can be applied to manually and automatically create tests in the CODESYS development environment. We use behavior-based, translation rules-based, and coverage-generated tests to validate the PyLC process.

PyLC transforms the PLC program into Python which is an open-source dynamic programming language that Guido van Rossum invented in 1990. Python has gained massive popularity during the last 20 years. Based on the latest statistics of top programming languages in 2022, Python is the top programming language worldwide based on TIOBE and PYPL Index[4]. Python is chosen as the destination language in the PyLC translation framework because it has good compatibility with parsing XML files, a widespread format used when dealing with PLCOpen formats used in the PLC IDEs for file exchange. The Python Test Automation Framework (TAF) we have chosen in this work is Pynguin [11] , a state-of-the-art automated test case generation tool for Python programs that uses search-based algorithms. It supports four different well-known search-based test case generation algorithms, including MOSA [12], DYNAMOSA [13], MIO [14], and WHOLE SUITE [15]. It is also equipped with a random test generator named RANDOM , which works based on the RANDOOP algorithm [16]. We note here that Python is the only non-IEC 61131-3 programming language officially supported by CODESYS IDE (a well-known PLC IDE) and can be directly compiled inside the IDE.

[4] https://statisticstimes.com/tech/top-computer-languages.php

The overall translation work flow of PyLC consists of four main phases chained to each other and working sequentially to eventually enable automatic test generation for PLC programs via Pynguin. Figure 13 shows the framework's workflow, while more details of each phase are described in the related paper. The first step is transforming the PLC program into Python code by considering the Translation Rules and PLC code specifications based on the IEC 61131-3 standard (Steps 1 and 2). Then the generated Python code is fed into the Translation Validation module, which checks the correctness of the transformed PLC code in Python based on the three different unit testing mechanisms (Step 3). Finally, the Translation Validation module of the transformed code (Step 4) uses unit testing to ensure that the code is scrutinized for proper use in further analysis and test generation.

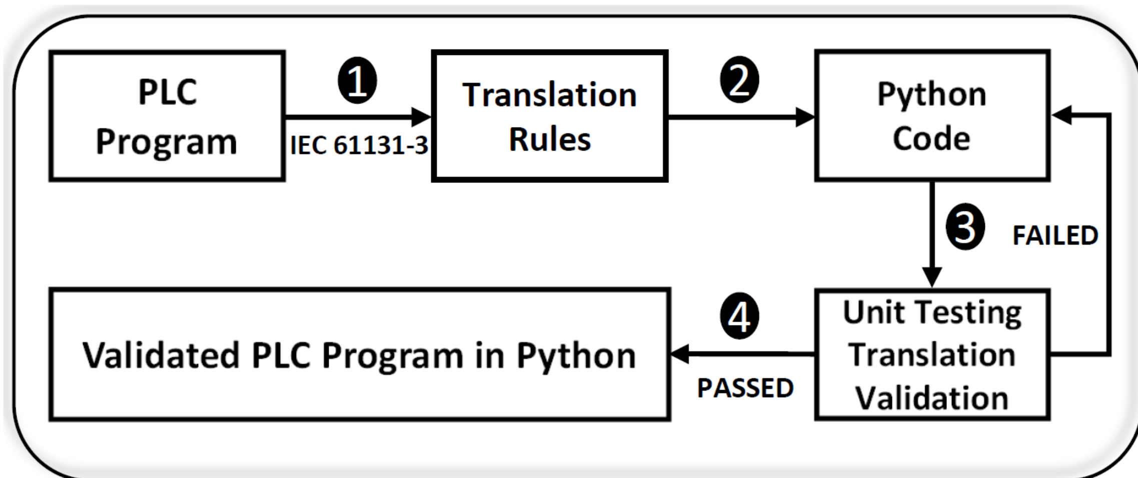


Figure 13: An Overview of the PyLC Framework, the Proposed Translation Mechanism for Translating a PLC Program into Python Code and Validating the Translation.

Translation Process. Our translation policy includes two common programming languages of IEC 61131-3: ST and FBD. Since ST is a textual programming language like Python, the transformation process is more straightforward. It includes translating each logical operator (e.g. AND, XOR, OR functions) into the corresponding operator in Python and mapping these together based on the network of the original PLC program. The rest of the section explains the transformation rules and validates the generated Python code. The translation process of our framework consists of 7 main steps, which can be observed in Figure 14. The translation process starts by analyzing the PLC program's inputs and outputs, transforming the input signals into Python function arguments, and considering the output signals as global variables in Python (Steps A, B). Then, the functionality of each interface Function and Function Block (FB) inside the PLC program (e.g. AND, XOR, TON) is analyzed based on their standardized functionality description in IEC61131-3 documentation (Step C). In the next step, the identified interface FBs are transformed into corresponding Python sub-functions that represent the same functionality based on the Block translation rules described in the rest of this section (Step D). After translating the blocks into sub-Python functions and feeding them with the

inputs as main Python function arguments, we analyze the network between different FBs, inputs, and outputs in the original PLC program to simulate these connections in the Python code and correctly map the elements to each other (Step E). The final step is identifying the execution order of the program elements inside the PLC program and implementing it in the translated Python code (Steps F, G).

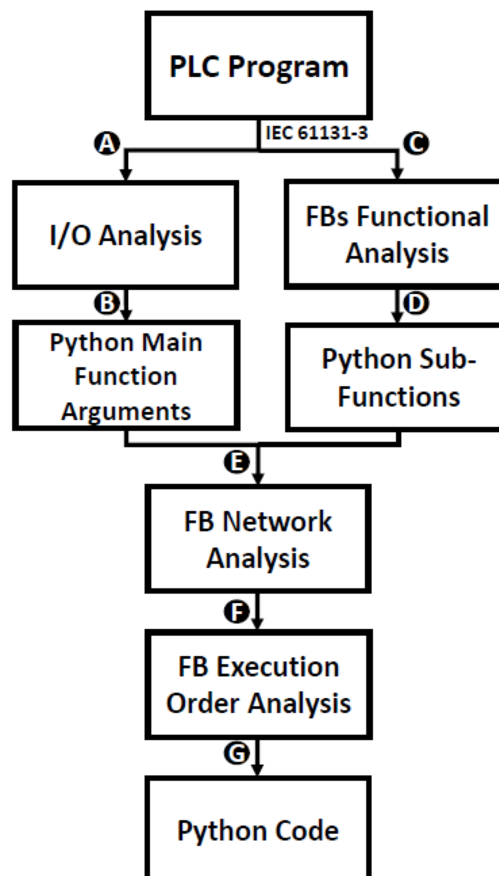


Figure 14: The Translation Workflow (TWF) Used in PyLC Framework for Translating a PLC Program into Python.

An overview of the translation rules we adhere to in the translation process is observed in Table 1. It is worth mentioning that every described step in this table is done by considering IEC 61131-3 specifications for the PLC program elements under translation. In other words, the translation mechanism is realized by using all the translation rules.

FBD/ST to Python Translation Rules		
Category	**PLC**	**Python**
Input(s)	Scanning PLC Program Inputs	Declaring the inputs as the main Python function arguments [a]
Output(s)	Scanning PLC Program Outputs	Declaring the outputs as global variables in Python [b]
Data Type	Identifying the data type of each I/O	Binding the data type of each PLC I/O to the corresponding data type in Python [c]
Data Range	Detecting I/O Variables Range	The accepted range of values for each PLC data type is declared using <, >, and = operators
FB Behavior	Analyzing the behavior of the FB based on the requirements	Implementing the FB behavior in Python as a sub-function with a dynamic range of inputs based on standardized ST and FBD implementation and specification in IEC-611313/CODESYS. [d]
FB Network	Analyzing the existing network between different FBs, Inputs, Outputs	Connecting the related Sub-function of each FB to other FBs, Inputs, and Outputs by a Python function call
Execution Order	Extracting the execution order of the program	Simulating the execution order by calling the main and sub Python functions in the correct order
Cyclic Execution	Identifying the cyclic execution delay time	Implementing the cyclic execution using a Python timer module equipped with a specific iteration(s) number

Table 1: Translation Rules (TR) of the Proposed PLC Program to Python Code Considering IEC-61131-3 Standard

Validation of the Translated Code. To validate the correctness of the translated code in Python, we propose a unit testing-based validation mechanism that consists of 3 different validation types, including 1) requirement-based testing, 2) translation rules checking, and 3) search-based test generation. To check the validity of the translated code, we generate and execute unit test cases that meet the requirements of each validation category. It should be noted that our proposed validation mechanism is not used to demonstrate the semantic equivalence of the source and target programs. Instead, we aim to validate the transformation through unit testing and conformance tests. Conformance tests are made to verify whether the PyLC results comply with the requirements imposed by the PLC program definition and the translation rules checks. The proposed translation validation mechanism consists of 8 main steps and can be observed in Figure 15. More explanation on each step of the hybrid Unit-Testing Validation Mechanism of the Translated PLC Code in Python can be found in the related publication [17].

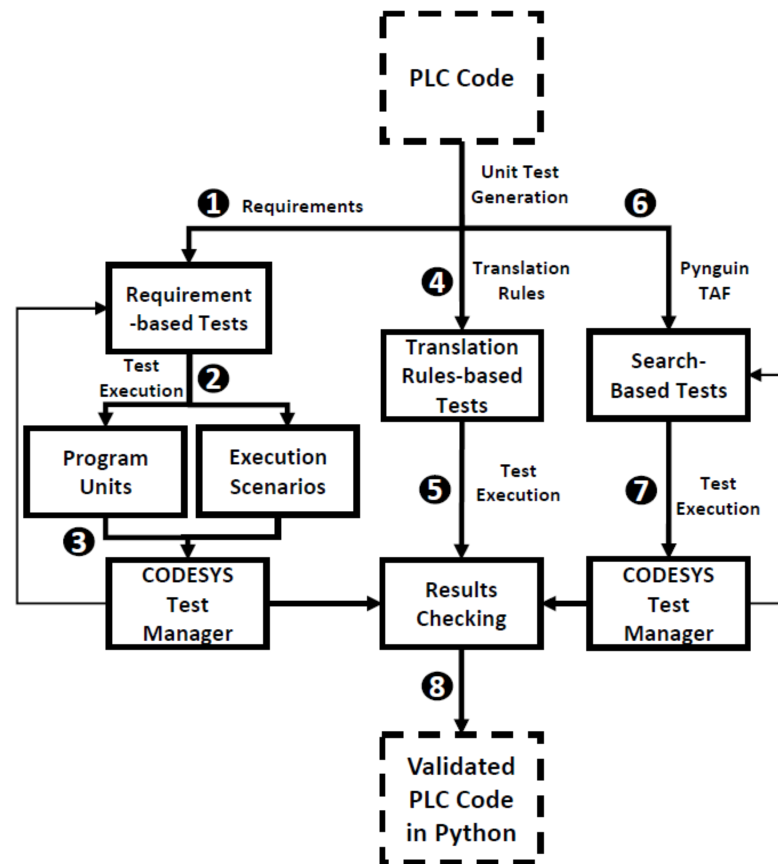


Figure 15: An Overview of The hybrid Unit-Testing Validation Mechanism of the Translated PLC Code in Python

At this stage, PyLC can only validate the correctness of the code translation from PLC to Python from requirements and behavior points of view but we plan to equip it with a powerful automatic static verifier of Python named Nagini [18] which works based on Viper verification infrastructure to validate the translation correction semantically as well.

4.2. Relation to VeriDevOps and CaseStudies

We consider ten different PLC programs to evaluate our proposed translation framework in real-world circumstances, including 6 ST and 4 FBD programs that are provided to us by a big automation company in Sweden.

The first validation mechanism we used is **Unit Testing Validation based on Requirements** which is behavior validation of the translated PLC programs into Python is done via requirements-based testing. It means that for each PLC program transformed into Python, the actual behavior of the translated PLC

program in Python is compared with the expected behavior in the original PLC program based on test cases covering all stated requirements. For six out of ten translated PLC programs (PRG5 to PRG10), both categories of the aforementioned requirement-based test cases are executed on the original PLC program in CODESYS IDE using CODESYS Test Manager. The result of executing these test cases on both Python and PLC environments is then compared. We find that the same test case execution status is obtained in CODESYS IDE, indicating the program's accurate translation using PyLC Framework according to the specific tested requirements. The reason behind excluding four PLC programs from this process is that these programs are designed to analyze some data directly from specific hardware cameras, and altering these inputs manually in CODESYS Test Manager is not feasible directly using unit testing.

The second validation mechanism we leveraged is **Checking PyLC Translation Rules** which is an investigation of the use of checks related to our translation rules. For each PLC program, we have designed several unit test cases that investigate the alignment of the translated programs to the proposed translation rules in PyLC. These test cases check if the transformation of certain PLC elements(i.e., input(s), output(s), data type, data range, FB behavior, FB network, execution order, and cyclic execution) produces valid elements in the translated PLC programs. We have developed test cases manually using the Python unittest tool.

The third and final validation mechanism we used in validating the correctness of the translation in PyLC is **Validation using Pynguin Test Generation.** In this validation mechanism we leverage Pynguin, an automated search based testing framework for Python, within our framework. Among all of the supported search-based algorithms of Pynguin, we use DYNAMOSA (Pynguin's default algorithm) as our algorithm of choice for generating test cases. We have followed Pynguin's default configuration using DYNAMOSA, a test generation time budget of 10 minutes, and mutation analysis enabled. The results of automated test generation and execution on ten considered PLC programs of this study using Pynguin are shown in Table 2. As it is visible in the gathered results, all of the generated mutants for PRG1 to PRG 10 have been killed by the Pynguin TAF except the generated mutants for PRG5. We believe the reason behind this result is the limited time budget of 10 mins that we considered for automatic test generation using Pynguin TAF (i.e., we forced the mutation analysis process to stop immediately after 10 minutes and not all mutants are targeted).

Test Suite	Program	Number of TCs	Verdict	Test Generation Time(s)	Test Execution Time	Branch Coverage (%)	Covered Branches	Killed/ Survived Mutants
1	PRG1	7	5/7	5	0.16	100	16/16	72/0
2	PRG2	7	4/7	4	0.14	100	16/16	67/0
3	PRG3	6	4/6	609	0.13	80	27/34	164/0
4	PRG4	27	20/27	653	0.5	88.89	119/134	170/0
5	PRG5	2	2/2	601	0.03	77.78	6/8	5/4
6	PRG6	1	1/1	1	0.02	100	0/0	0/0
7	PRG7	4	2/4	601	0.13	86.67	12/14	18/0
8	PRG8	7	3/7	601	0.14	75.86	21/28	26/0
9	PRG9	7	5/7	610	0.23	86.96	19/22	40/0
10	PRG10	6	5/6	606	0.12	88.24	14/16	18/0

Table 2: Results of Automatic Test Generation/Execution for Translated PLC Programs using Pynguin TAF

The results of generating and executing test cases for the translated PLC programs into Python using PyLC show that this method is feasible for validating the transformation and test generation during the development of PLC programs. However, using other search-based algorithms and increasing the test generation budget, especially for large programs such as PRG4, might increase the obtained code coverage and improve the mutation analysis results. In the end, we execute the generated test cases on the original PLC programs in CODESYS IDE to investigate whether their execution in the original PLC environment produces the same results. Executing the test cases in CODESYS IDE has been done via CODESYS Test Manager.

4.3. Detailed overview for relevant usage scenarios

We have successfully applied our PyLC approach to transform and validate PLC programs. We also evaluated the applicability and efficiency of our proposed framework by applying it to the different industrial PLC programs. However, a significant threat to the validity of our experiments is the question of the representativity of the programs used. While our case study does not cover the whole range of possibilities of program transformations, these programs are still distinct from one another and of different sizes.

We believe the PyLC can be evolved by becoming fully automated by parsing in CODESYS the PLC program and using the test manager to generate and execute test cases without user intervention to minimize the manual overhead. Another direction for future research, is to equip PyLC with a formal verification mechanism, to increase correctness assurance. The final contribution for future work can be investigating the performance of the different search-based algorithms in generating more effective test cases for evolving PLC programs.

The planned future version of PyLC can enable automated search-based test generation and execution for both code development and DevOps operations teams in industry. The involved engineers in aforementioned teams can use PyLC to automatically generate and execute meaningful search-based tests on the code under development. Moreover, PyLC brings automated Mutation Analysis to the world of PLC testing which can help the testers to have a better analysis of the faulty parts of the code under development. Considering the current manual state of the practice for PLC testing in industry, using efficient PLC TAFs such as PyLC can boost up the testing process and save a considerable amount of time and energy for industrial automation companies.

5. GW2UPPAAL [new]

5.1. General description (purpose, features, interfaces)

We propose a hybrid approach that can perform an automated analysis of a GW model by transforming it into a UPPAAL model and generating queries that are automatically verified by running "verifyTA" without actually running the GUI of UPPAAL to perform model checking. An initial evaluation shows that the time taken by our tool for transforming these models is consistently lower when compared to manually creating the model and properties in UPPAAL and checking these using the GUI. However, checking other properties corresponding to software requirements requires manual intervention to generate queries. Thus, our proposed approach is the first step toward combining model-based testing with automated analysis and verification tools, which can be further modified to create a more realistic and complex set of properties.

For this purpose, we developed a hybrid approach that transforms a model obtained from an Model-Based Testing (MBT) automation tool into a model compatible with a state-of-the-art model checker. In addition, we automate the process of model checking by generating some queries to verify the model. To achieve this, we used a well known open-source MBT tool named GraphWalker (GW)[5] . Models in GW are created in the form of directed graphs. This tool lacks the capability to automatically analyze and verify if the model corresponds to certain requirements. For this purpose, we are using a state-of-the-art model checker called UPPAAL to perform model checking. This integrated tool environment allows developing models as a network of timed automata and can verify specific properties on these models.

[5] https://graphwalker.github.io/

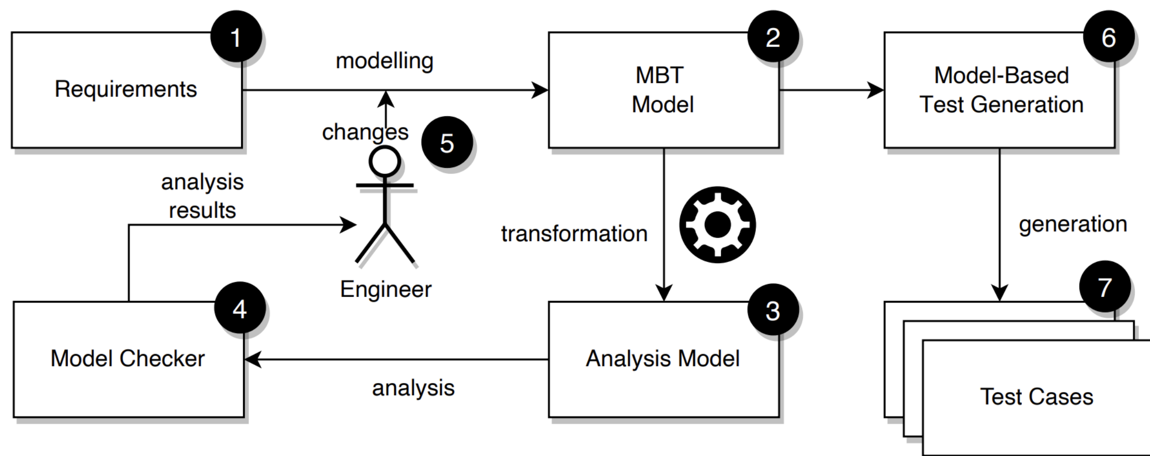


Figure 16. The overall method for combined model-based analysis using model-checking and the relation to testing at design level.

In Figure 16, an overall method for model-based analysis and test generation is identified. A generic process of combined model-based analysis and test generation proceeds as follows:

- Step 1. Requirements artifacts are used or created for the purpose of guiding the test generation and analysis. In our case, the requirement artifact is either a specification of what the System-under-Test (SUT) should do in different forms (e.g., finite-state model).
- Step 2. A model is obtained using a testing tool that can be used for modeling an MBT model objective. The first step of modeling involves a human understanding the requirements and exploring the requirements specification document. For example, an FSM-based model consists of nodes and directed edges. The nodes represent the state of the system, whereas edges represent the requests/decisions when a certain event occurs.
- Step 3. In this step an automatic transformation is needed to map the test model to an analysis model used for model checking. For example, in the case of a finite-state model, guards, actions and variable declarations are used to generate a formal model.
- Step 4. Given a formal model of the system , a model checker can be used to analyze the model given certain formalized requirements, for example, as a temporal logic formula. The model-checker returns an answer, and in some situations a model trace.
- Step 5. Based on the analysis results, the engineer could make certain changes to the original model and can continue using the MBT model for test generation.
- Step 6. Using model-based test generation that encodes the test criteria and describes how the test generator should choose the resulting tests, one can generate test cases based on certain goals (e.g., model coverage, random test goals).
- Step 7. A test suite is generated by running the model over many possible executions using a certain model based test generation tool.

For more details on the implementation and usage of GW2UPPAAL we refer the reader to the study of Tiwari, Iyer and Enoiu [19] .

5.2. Relation to VeriDevOps and CaseStudies

We evaluated the tool with models containing single as well as multiple diagrams from different sources including functional, safety and security requirements. Multiple diagrams are essentially part of a single model but are divided into multiple diagrams for better understanding and ease of use. These diagrams are connected by using shared vertices. A shared vertex is a vertex in the diagram that can be shared between multiple diagrams so that the test runner can execute the whole model, where the shared state is defined. This tool transforms it into a single model by flattening all the multiple diagrams. This is needed since UPPAAL does not directly support multiple diagrams in the same way as GW does. GW2UPPAAL is used on the ABB use case by targeting the integration with the requirements provided for FBD programs. It focuses on prevention at the design level using IEC 61131-3 programs by automatic checking of predefined logic properties related to the interfaces of these programs. The purpose of the transformation is to construct a target model by filling the model with the corresponding behavior as explained. Since IEC 61131-3 programs allow the use of behavioral notations, we exploit this and specify the behavior by assigning a TA model to each element mapped from its corresponding IEC 61131-3 program (Scenario ABB_S2 and Scenario ABB_S1). Specifically, the main **benefit** of using this method is to verify the model level vulnerabilities using specific test requirements related to security attacks and safety concerns. Without employing such a tool, the **drawback** is that one cannot verify (both analysis and test generation) at design time that certain faults and vulnerabilities are not present in the specification or design.

5.3. Detailed overview for relevant usage scenarios

The combined MBT and analysis technique implemented in GW2UPPAAL is shown in Figure 17. It is divided into the following five usage scenarios:

- 1) The test designer creates a GW model, which is exported as JSON.
- 2) In the next step, the JSON file is then imported into the tool by providing the file's location before executing the tool (JAR) in the command.
- 3) While running the GW2UPPAAL tool, the UPPAAL model in XML format is generated and used for analysis.
- 4) The generated UPPAAL model is then imported and executed by starting verifyta and providing the name of the generated model. This model also contains the queries to check the reachability and deadlock properties tested by executing "verifyta". 5) The results of this

verification is then visually displayed and a test engineer will use these to analyze the model in the UPPAAL simulator to adapt the model in GW before test generation. Apart from automated analysis, the generated model can also verify manually created queries to gain more confidence in the developed model before testing.

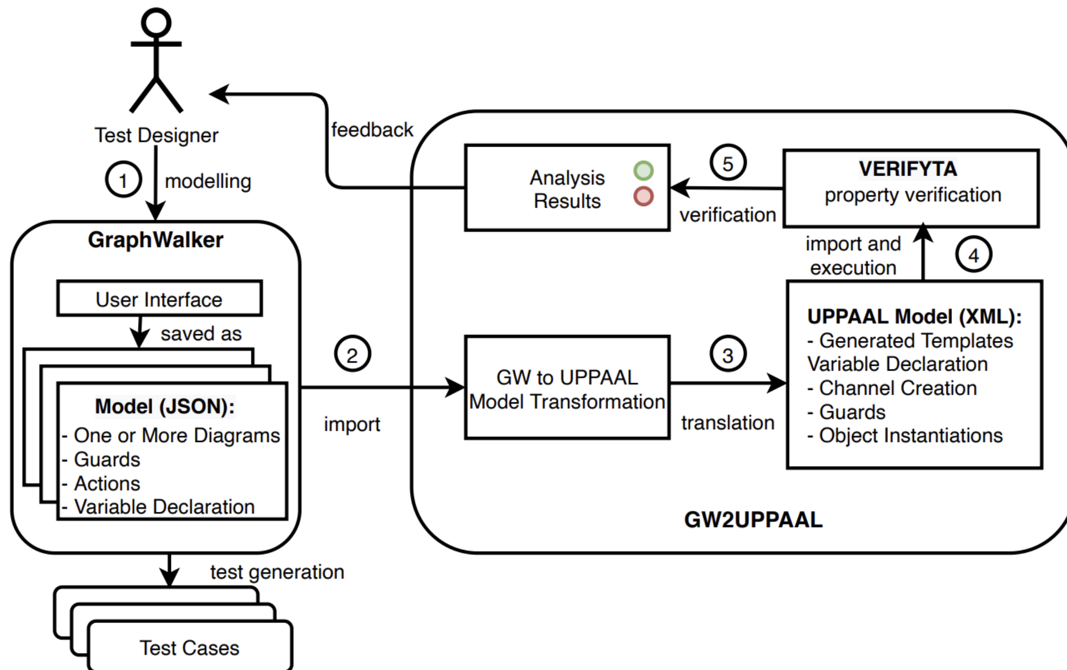


Figure 17. The overall architecture of GW2UPPAAL tool and the interaction with GraphWalker for test generation.

Use Case Scenario 1 - Model Transformation (ABB_S1). To model check an GW model we map it to a finite state system suitable for model checking.

Use Case Scenario 2 - Model Checking (ABB_S2). We generate properties expressible as a reachability property used in most model checkers.

5.4. How to get it, install it, licensing

Our tool transforms the GW JSON into UPPAAL compatible XML file to import the model in UPPAAL and perform model checking. The approach is divided into four steps. The programming language used for transformation is JAVA. The tool is available in Github[6].

Please follow the below instructions:

[6] https://github.com/eduardenoiu/GW2UPPAAL

- Use the ToolOutput folder as the folder where the UPPAAL models will be exported.
- Navigate to the ToolOutput folder and open the terminal/command prompt.
- Type the command: java -jar GW2UPPAAL.jar [path to out-put file folder] [name of the output file that the tester wants to give] [path to input GW JSON file] [Name of the JSON file that the tester wants to transform].

6. Modelio

6.1. General description

Modelio is both an open source[7] and a commercial[8] modeling environment (that supports UML2, BPMN2, MARTE and SysML among others). Modelio delivers a broad-focused range of standards-based functionalities for software developers, analysts, designers, business architects and system architects. Modelio is built around a central repository, around which a set of modules are defined. Each module provides some specific facilities dedicated to specific needs.

Three functional sets of modules seem to be the most relevant in our context. These functional sets are the following:

- Modeling and consistency check: UML [20], SysML [21], BPMN [22] are a subset of the long list of standards supported by Modelio. The most relevant languages, in VeriDevOps context, seem to be the one's related to Requirement, System, and Test modeling. Modelio allows the usage of a specific (the most relevant) language combination but also checking related to specific language usage. For example, Figure 18 depicts data modelling of a web application.

[7] https://www.modelio.org/
[8] https://www.modeliosoft.com/en/

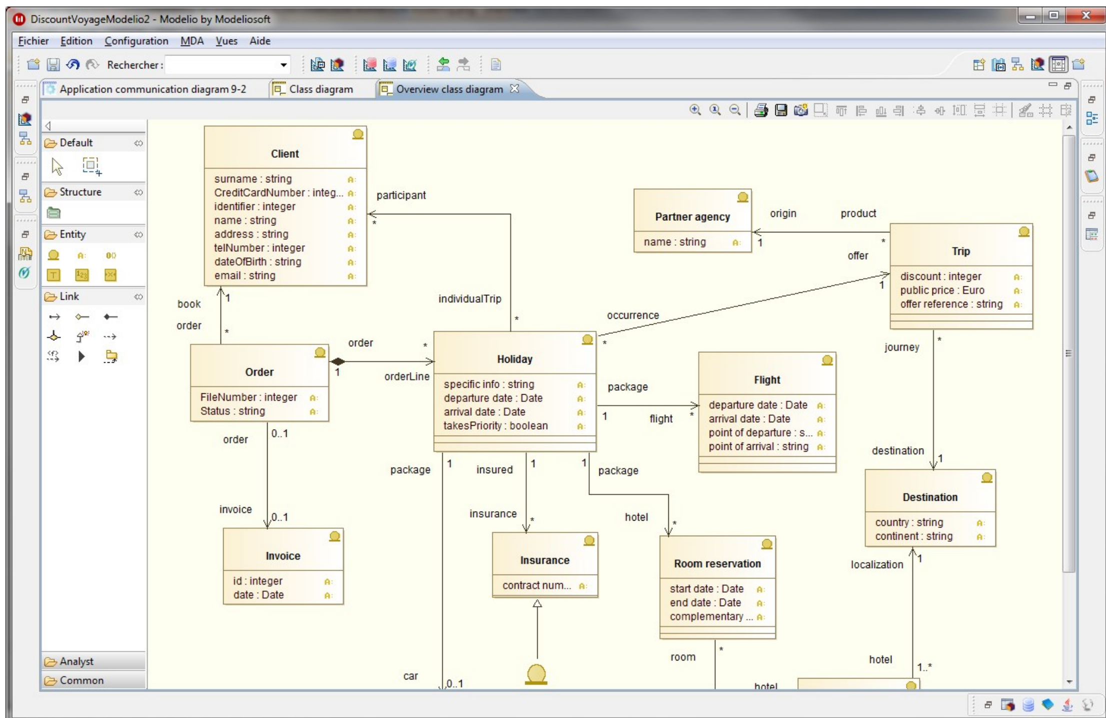


Figure 18. Example of the modeling under Modelio.

- Text generation: Modelio has powerful code generation and reverse engineering modules for Java, C# and C++ language. Moreover, it is able to generate documentation in several formats (e.g., HTML or OpenXML) which can be stored in the Modelio repository (Figure 19).

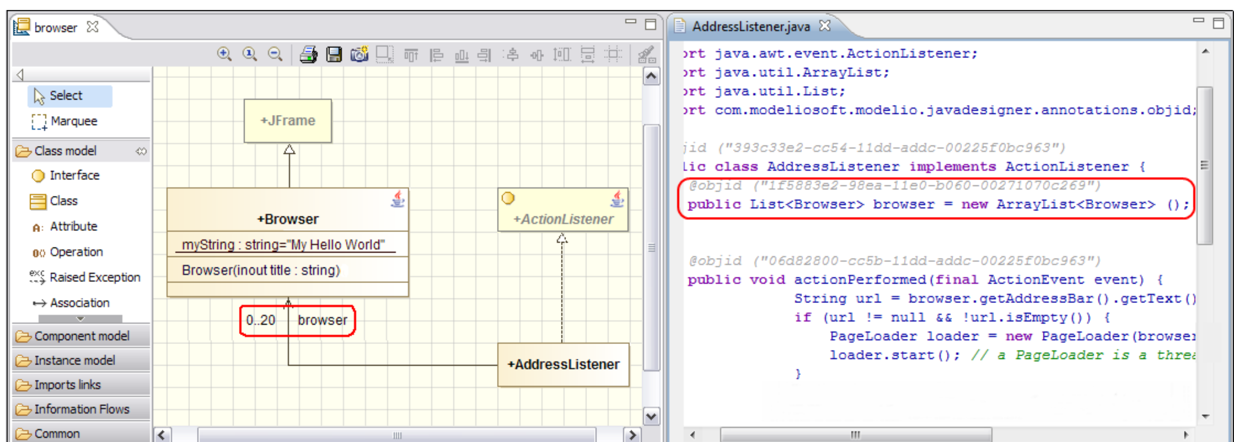


Figure 19. Example of the Java modeling (on left) and generated Java code (on right).

- Impact analysis and traceability: As already stated Modelio is able to provide several modeling levels each of them targeting specific stakeholders. Traceability and impact analysis can help to determine the cost, in terms of security for example, of any changes if part of a model is modified. This mechanism helps discover the value of an entire model by clearly identifying which and how many model elements are the most costly vulnerable for example (Figure 20).

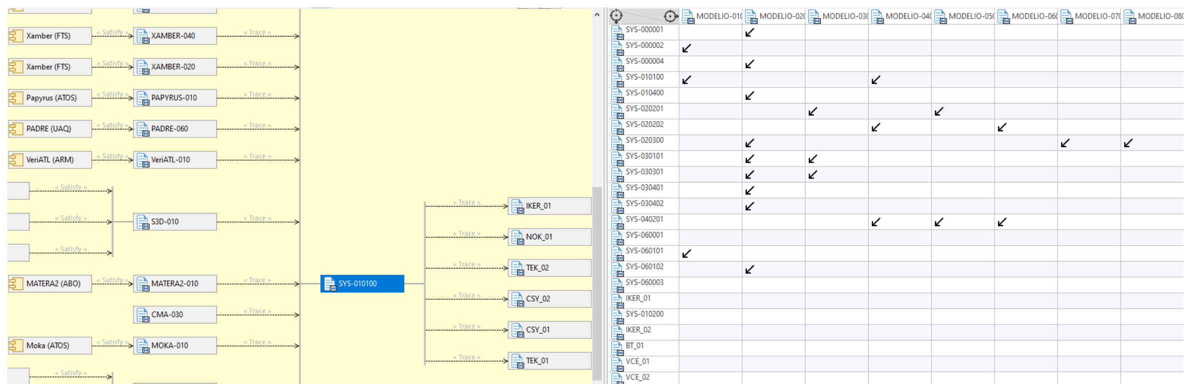


Figure 20. The Modelio link editor (on left) and the Modelio matrix (on right).

6.2. Relation to VeriDevOps and CaseStudies

In VeriDevOps context, Modelio mainly targets the FAGOR use case by automating:

- the detection of known vulnerabilities and attacks and providing automatic countermeasures or manual recommendations for decision support.
- the extraction of security recommendations (or requirements) and perform related tests.

The above are implemented in Modelio by employing the Seamless Object Oriented Requirements [23] (SOOR) concept which suggests the creation of requirements by following the Object-Oriented Programming (OOP) paradigm. In the latter, each requirement is specified as a class, including the requirement definition in a textual form, but may also include other types of representations such as Linear Temporal Logic (LTL) formula or simply a test case for requirements validation. The relations among requirements may be expressed in terms of associations and inheritance. The OOP analysis may be applied to argue about reusability, complexity or maintainability of the set of requirements. In VeriDevOps, we implemented SOOR in Java language. By reversing Java code with Modelio, we obtain the requirements specification in UML. Modelio provides the ability to verify syntax of UML models and provides recommendations for OOP analysis through audit rules and specific metrics.

In this project, Modelio is the only tool providing such capability for specifying security requirements using object-oriented concepts in an executable form. The following section details how

the VeriDevOps consortium plans to use Modelio in two of the FAGOR use case scenarios respectively FAG_S3 and FAG_S4, as detailed below.

6.3. Detailed overview for relevant usage scenarios

Alignment of the Edge Device configuration to the NIST Framework (FAG_S3)

The main goal of this Use case consists in having a Modelio extension able to automatically identify NIST framework security requirements and store them inside Modelio. For each stored requirement, create a model of the required test environment to be able to generate the test set needed.

Vulnerability verification and correction suggestions (FAG_S4)

In this particular use case, VeriDevOps plans to define an action repository. Each action will be related to identified and known vulnerabilities which will be detected from product description. So according to a specific product description, a set of vulnerabilities will be identified conducting to a related set of actions to protect the product.

With regards to the above scenarios, the following Modelio related analysis should be applied. For example, in case an industrial PC runs on Ubuntu Linux distribution, STIG guidelines can suggest disabling a certain number of packages. For, example STIG recommendation V_219157 states:

"Removing the Network Information Service (NIS) package decreases the risk of the accidental (or intentional) activation of NIS or NIS+ services.".

This recommendation suggests removing the NIS package from the system. We implemented these and many other recommendations as a SOOR in Java - the process that we called RQCODE (Requirements as a code). While implementing the requirements and reversing them to UML with Modelio for analysis, we discovered a number of patterns. One of these patterns deals with disabling or enabling system packages. Separating the PackagePattern (Figure 21) from the actual implementation of disabling particular packages helps to maintain the system of requirements, reuse it in other distributions and control complexity and well formedness through Modelio.

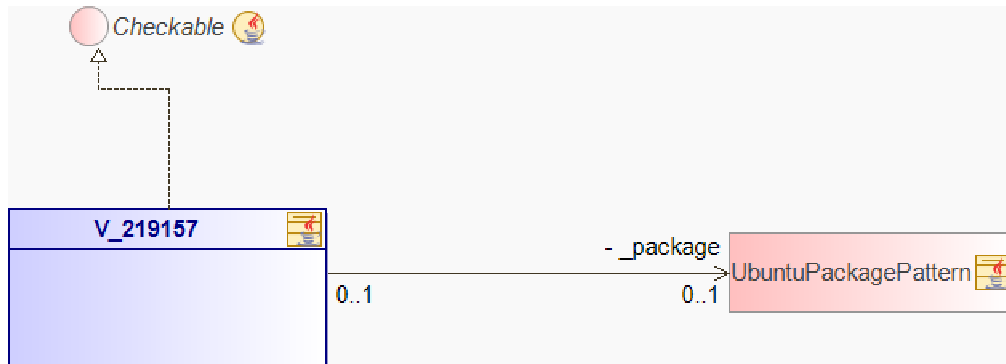


Figure 21. RQCODE UbuntuPackagePattern in UML with Modelio

In case of PackagePattern, it can be extended to CentOS distribution that uses a different package manager (e.g., *rmp* instead of *apt*) - changing one pattern class will help to run the same security recommendations in a new system. Thus the approach improves efficiency and reduces a duplication in the important security related routines.

In this scenario, reversing RQCODE in Modelio helps to manage requirements on the higher System Level, while providing capabilities for syntax check and audit of OOP constructions in requirements and recommendations.

6.4. How to get it, install it, licensing

Modelio exists in two versions 1) Open Source and 2) Commercial. The following sections explain how to get both of them. In VeriDevOps context, the consortium is using the commercial one mainly because it is the only one to provide Requirement modeling.

6.4.1. Modelio Open Source

For information on supported operating systems and required libraries, check system requirements. To install Modelio open source starts by:

1. Download Modelio.
2. Extract the package into the directory of your choice [1].
3. Start Modelio.

When Modelio starts, one will get a welcome page with useful hints (Figure 22):

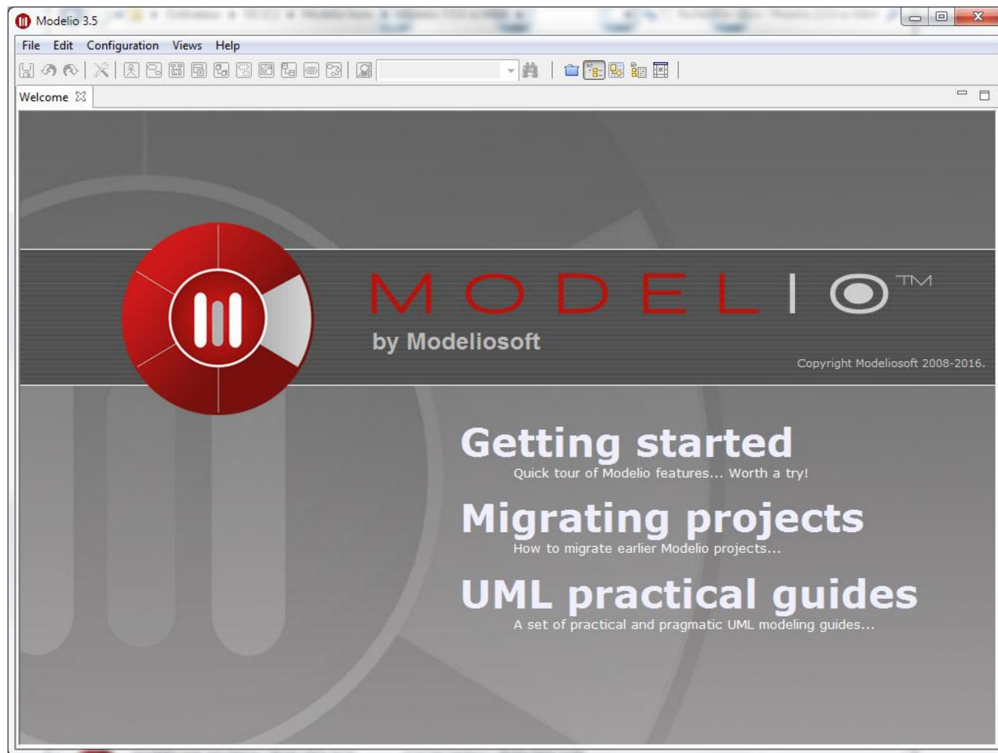


Figure 22. The Modelio welcome screen

Then one can create the first project.

6.4.2. Modelio Commercial

To get the commercial version of Modelio, if you are an educational institute please check the Modelio Academic Program otherwise evaluate it for ten days check Modelio Ten Days Evaluation.

6.5. New compared to D4.1

The VeriDevOps verification tools have been ported and tested with the latest version of the Java Designer modules on the Modelio 5.3.

7. Conclusions

This deliverable extends our previous approaches and presents several new ones to improve the modeling, testing, and analysis of industrial control systems. Firstly, the GW2UPPAAL tool is introduced to automate the modeling step and reduce the time required for the verification of security

requirements. Secondly, the CompleteTest approach is extended to address potential vulnerabilities in FBD programs, ensuring that they are not susceptible to attacks. Thirdly, the Modelio tool is updated to use the latest version of Java, improving its performance. Fourthly, the PyLC framework is proposed, enabling the transformation of PLC programs into Python code and generating effective tests to validate safety-critical industrial programs. Lastly, an approach for automated analysis of a GraphWalker model is presented, demonstrating that it takes less time to create a model in GraphWalker and transform it into a UPPAAL model than creating a model directly in the UPPAAL GUI tool. Overall, the proposed tools and approaches aim for improving the safety and security of industrial control systems.

References

[1] VeriDevOps project consortium, 'D4.1 Tools for prevention at design level - initial version', project deliverable, Apr. 2021.

[2] G. Behrmann, A. David, and K. G. Larsen, 'A Tutorial on Uppaal', in *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*, M. Bernardo and F. Corradini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236. doi: 10.1007/978-3-540-30080-9_7.

[3] R. Alur, 'Timed Automata', in *Computer Aided Verification*, Berlin, Heidelberg, 1999, pp. 8–22.

[4] 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: An Outcome of the FORTEST Network, Revised Selected Papers*, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 77–117. doi: 10.1007/978-3-540-78917-8_3.

[5] I. AlAttili, F. Houben, G. Igna, S. Michels, F. Zhu, and F. Vaandrager, 'Adaptive Scheduling of Data Paths using Uppaal Tiga', *ArXiv E-Prints*, p. arXiv:0912.1897, Dec. 2009.

[6] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, 'Symbolic model checking for real-time systems', in *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, 1992, pp. 394–406. doi: 10.1109/LICS.1992.185551.

[7] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, 'Uppaal SMC tutorial', *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 4, pp. 397–415, Aug. 2015, doi: 10.1007/s10009-014-0361-y.

[8] 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', *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 3, pp. 335–353, Jun. 2016, doi: 10.1007/s10009-014-0355-9.

[9] G. Fraser, F. Wotawa, and Paul. E. Amman, 'Testing with model checkers: a survey', *Softw. Test. Verification Reliab.*, vol. 19, no. 3, pp. 215–261, 2009, doi: 10.1007/s10009-014-0355-9.

[10] J. Karl Heinz and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems. Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*, 2nd ed. Springer Berlin, Heidelberg. [Online]. Available:

https://doi.org/10.1007/978-3-642-12015-2
[11] S. Lukasczyk, F. Kroiß, and G. Fraser, ‘Automated Unit Test Generation for Python’, in *Search-Based Software Engineering*, vol. 12420, A. Aleti and A. Panichella, Eds. Cham: Springer International Publishing, 2020, pp. 9–24. doi: 10.1007/978-3-030-59762-7_2.
[12] A. Panichella, F. M. Kifetew, and P. Tonella, ‘Reformulating Branch Coverage as a Many-Objective Optimization Problem’, in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Graz, Austria, Apr. 2015, pp. 1–10. doi: 10.1109/ICST.2015.7102604.
[13] A. Panichella, F. M. Kifetew, and P. Tonella, ‘Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets’, *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, Feb. 2018, doi: 10.1109/TSE.2017.2663435.
[14] A. Arcuri, ‘Many Independent Objective (MIO) Algorithm for Test Suite Generation’, in *Search Based Software Engineering*, vol. 10452, T. Menzies and J. Petke, Eds. Cham: Springer International Publishing, 2017, pp. 3–17. doi: 10.1007/978-3-319-66299-2_1.
[15] G. Fraser and A. Arcuri, ‘Whole Test Suite Generation’, *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013, doi: 10.1109/TSE.2012.14.
[16] C. Pacheco and M. D. Ernst, ‘Randoop: feedback-directed random testing for Java’, in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, Montreal Quebec Canada, Oct. 2007, pp. 815–816. doi: 10.1145/1297846.1297902.
[17] M. E. Salari, E. P. Enoiu, W. Afzal, and C. Seceleanu, ‘PyLC: A Framework for Transforming and Validating PLC Software using Python and Pynguin Test Generator’, in *The 38th ACM/SIGAPP Symposium On Applied Computing (SAC 2023)*, Apr. 2023.
[18] H. Chockler and G. Weissenbacher, *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Springer Nature, 2018.
[19] S. Tiwari, K. Iyer, and E. P. Enoiu, ‘Combining Model-Based Testing and Automated Analysis of Behavioural Models using GraphWalker and UPPAAL’, in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, 2022, pp. 452–456. doi: 10.1109/APSEC57359.2022.00061.
[20] ‘OMG Unified Modeling Language’. [Online]. Available: https://www.uml.org/
[21] O. Casse, ‘SysML: Object Management Group (OMG) Systems Modeling Language’, *SysML in Action with Cameo Systems Modeler*. pp. 1–63, 2017.
[22] OMG, ‘Business Process Model and Notation (BPMN)’. [Online]. Available: https://www.omg.org/bpmn/
[23] A. Naumchev, ‘Seamless Object-Oriented Requirements’, in *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, 2019, pp. 0743–0748.