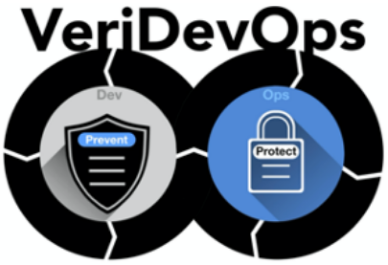


VeriDevOps

Automated Protection and Prevention to Meet Security

Requirements in DevOps Environments

D4.1 Tools for prevention at design level - initial 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:	30/06/2021
Coordinator:	ABO
Partners contributed:	MDU, MI, SOFT
Release Date:	30.6.2021
Version:	02
Abstract:	This deliverable overviews initial 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 D4.2.
Status:	• PU (Public)

Revision History

VERSION	DATE	DESCRIPTION	AUTHOR
02	15/09/2022	Version Updated based on project officer's comments	VeriDevOps consortium
01	30/06/2021	Final version	VeriDevOps consortium
.01	03/03/2021	Initial version created	ABO

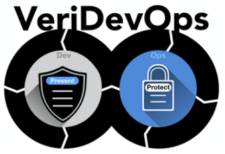
Executive Abstract

This deliverable overviews initial 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 D4.2.

This is an initial version of selected tools. Based on the evaluation results in use cases this list will be extended with additional tools. An updated version of these tools will be described in the final version of this deliverable, that is D4.4.

Version 01 of this document was released on 30.06.2021. The current version 02 is an update of the previous version, which improves the description of the connection between the methods and tools presented in version 01 and the software artefacts produced in work package 2. This connection is described in the Section 1 Introduction and captured in Figure 1. 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. Additional updates have been performed in each section to harmonise the description of the methods with the overview presented in the introduction.

Table of Contents



1. Introduction

In the context of VeriDevOps, security specifications play a central role in generating different artefacts needed later on for prevention at development in WP4. The first deliverable of this WP, namely D4.1, lists initial methods and tools for creating system specifications that satisfy the security properties imposed by security requirements. This deliverable lists several technologies that will be used in the context of the industrial use cases, as shown in Figure 1.

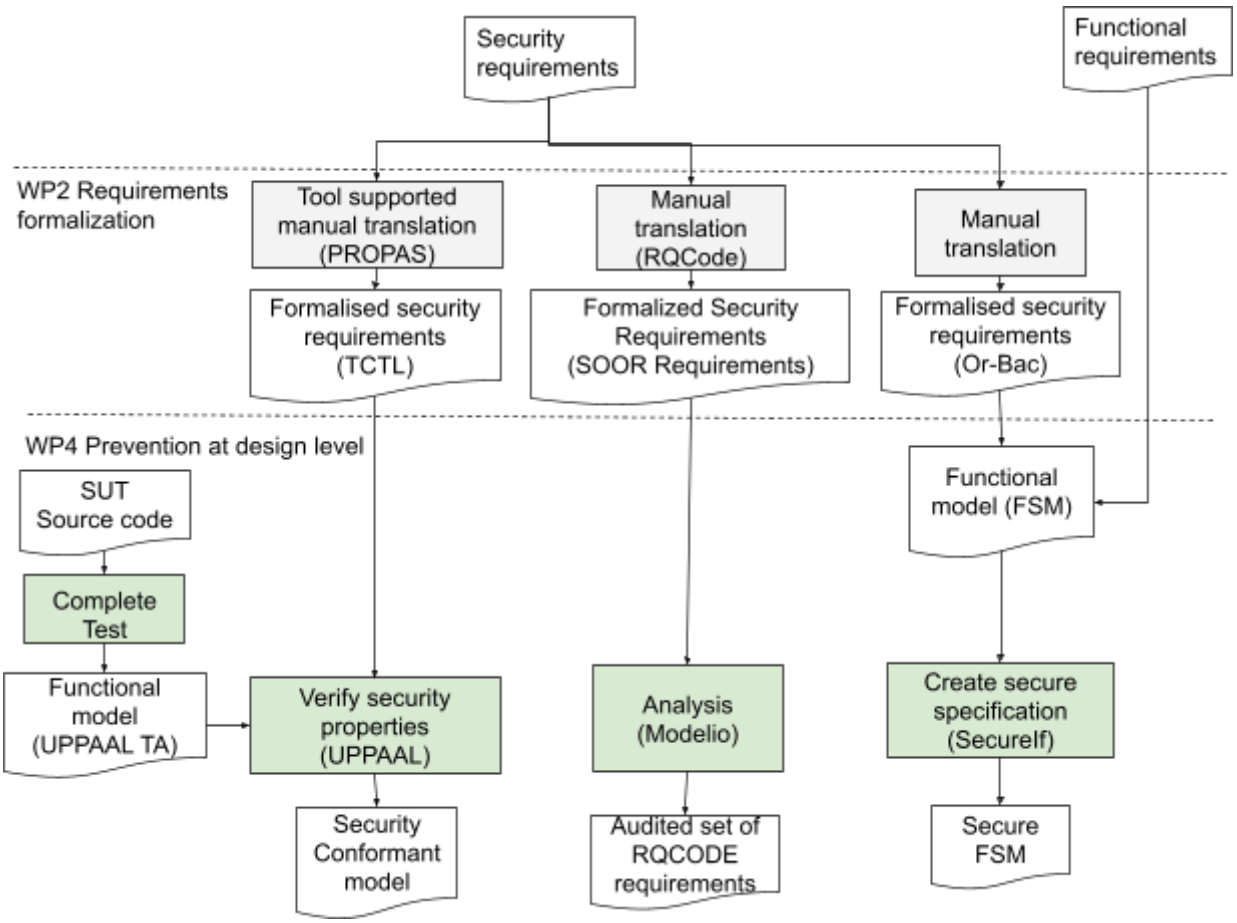


Figure 1. General overview of the usage of the tools used for prevention at design level.

These tools mentioned in Figure 1 use formal or semi-formal semantics to specify the expected behaviour 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 *formalised security requirements* developed in WP2. These formalised 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 formalised security requirements: Timed Computational Tree Logic (TCTL), Seamless Object Oriented Requirements (SOOR) and, respectively, Organisation-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.

Four tools are used for enforcing the specification of the system at design time:

- **CompleteTest** is used for extracting formal specifications from Function Block Diagrams of PLC systems and using such specifications later on to generate tests.
- **UPPAAL model checker** - CompleteTest uses UPPAAL Timed automata (UPPAAL TA) to represent the system under test and as a back-end to verify that the system specification satisfies security requirements specified as Timed Computational Tree Logic (TCTL) queries.
- **Modelio** is a modelling tool which is used to model and implement executable security requirements using the Seamless Object Oriented Requirements (SOOR) paradigm. The tool allows to specify requirements as code, analyse and execute them.
- **SecureIF** is a tool that can generate a secure specification of a system by combining security requirements formalised as Organisation-based access control (OrBac) rules with a behavioural specification of the system extracted from functional requirements in the form of an Extended Finite State Machine (EFSM).

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 (purpose, features, interfaces)

The UPPAAL model-checking tool suite is an integrated environment for modelling, validation, and verification of real-time systems [1]. It uses a network of extended Timed Automata, called UPPAAL Timed Automata, to specify the behaviour 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., SecureIF).

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 labelled edges) extended with real-valued variables [2]. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, are initialised with zero when the system is started, and then incremented synchronously with the same rate. The behaviour 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) [3]. 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 synchronisation variables called channels. A channel ch relates a pair of transitions in parallel processes where synchronised edges are labelled 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 modelled 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 behaviour 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 [1].

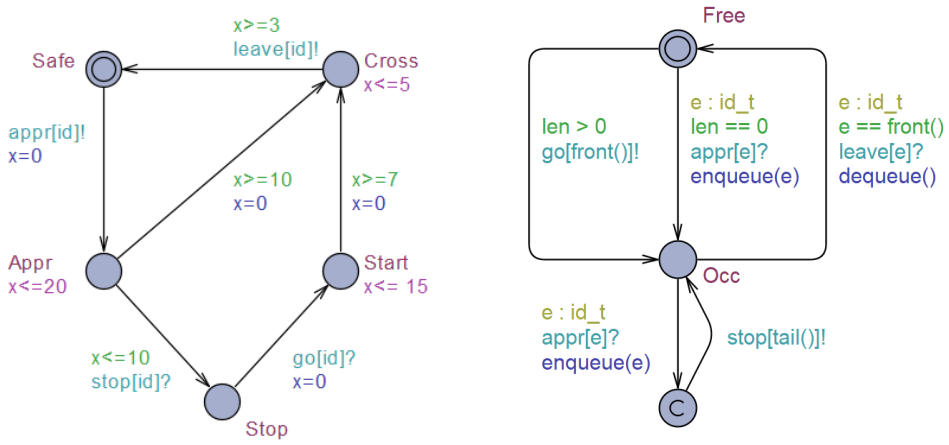


Figure 2. UPPAAL TA of a train from the train gate example [1]: left - Train TA and right- Gate TA

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 visualised 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 modelling mistakes.

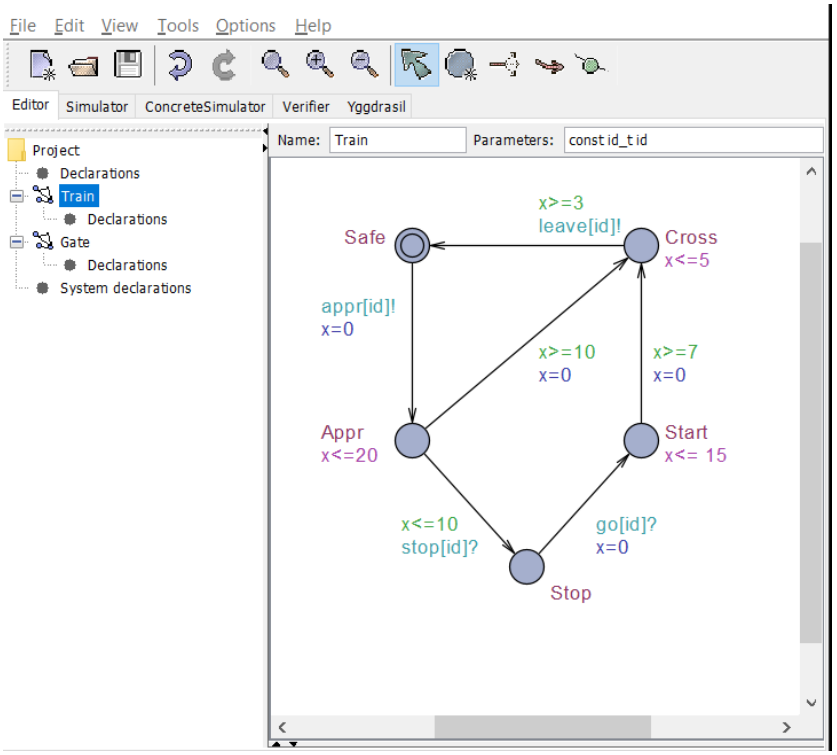


Figure 3. UPPAAL editor screenshot

The **Simulator tab** (Figure 4) in the graphical interface is a validation tool to examine possible dynamic executions of the system during the modelling stage and to visualise 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.

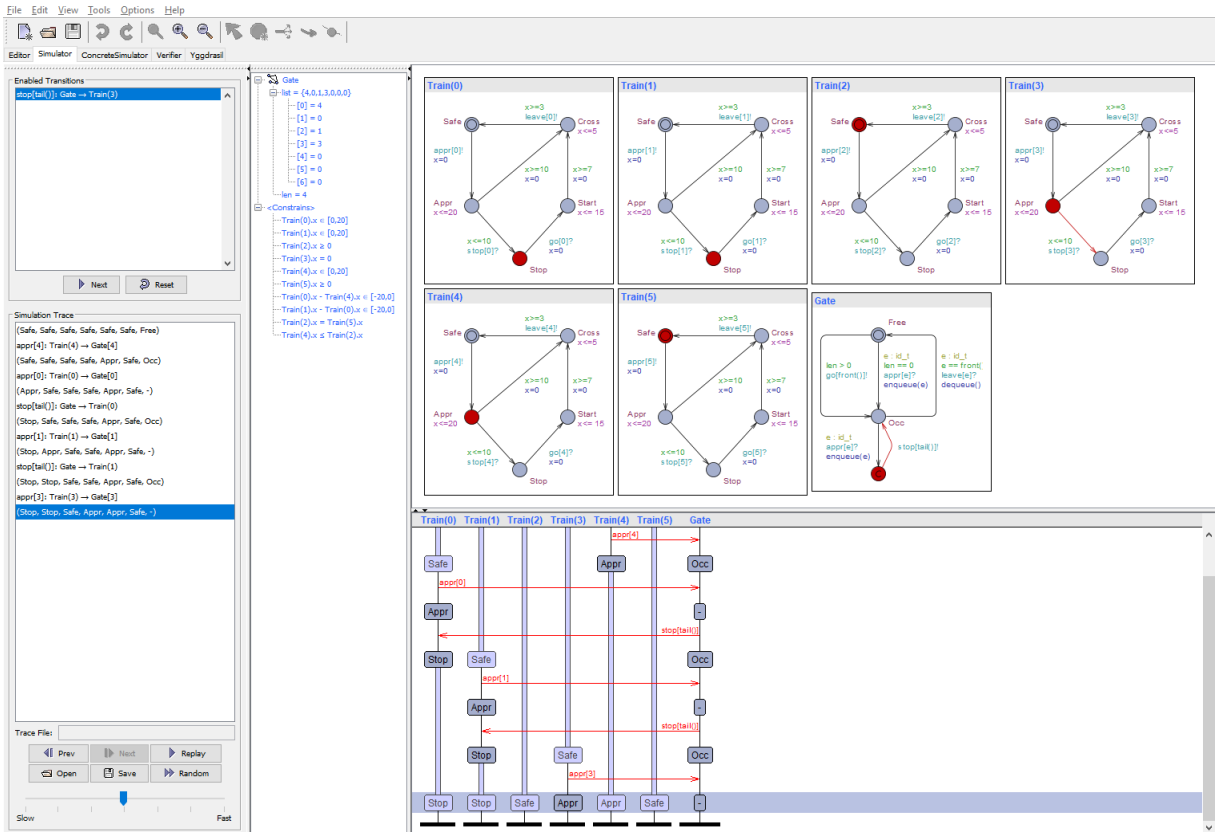


Figure 4: Symbolic simulator tab of UPPAAL

The third tab is a **concrete simulator** that was originally used in UPPAAL-Tiga [4]. 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.

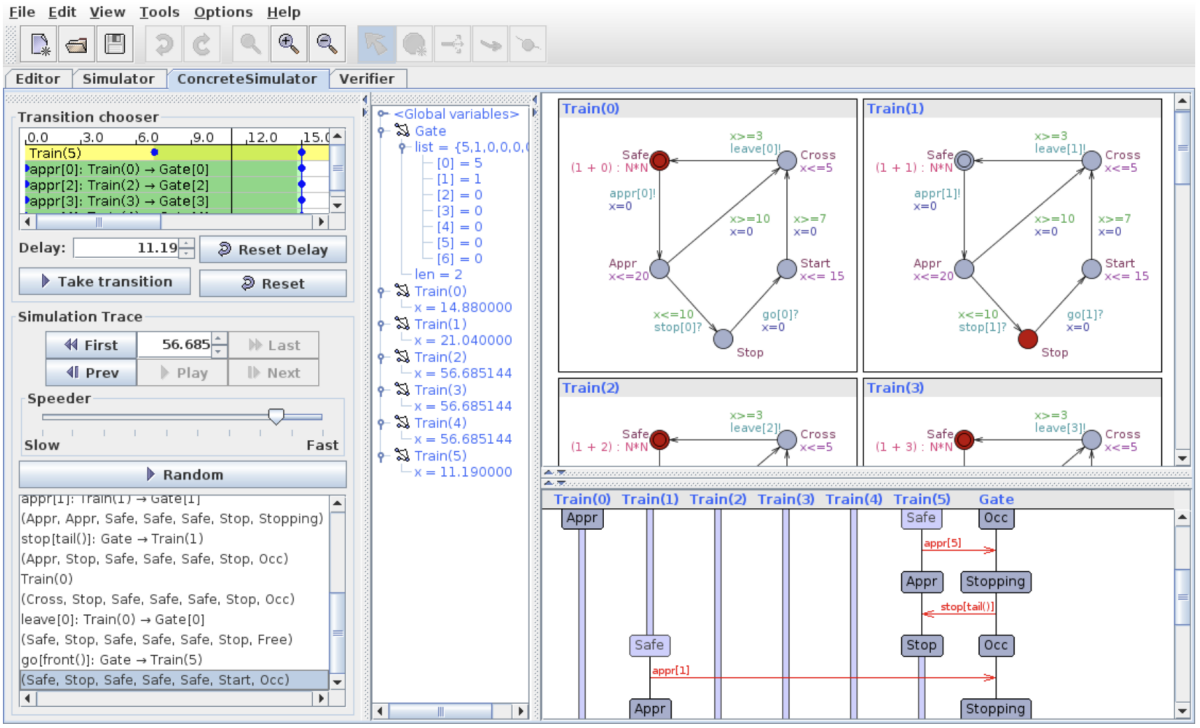


Figure 5. The concrete simulator in UPPAAL

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) [5]. 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 visualised in the UPPAAL Simulator for further debugging.

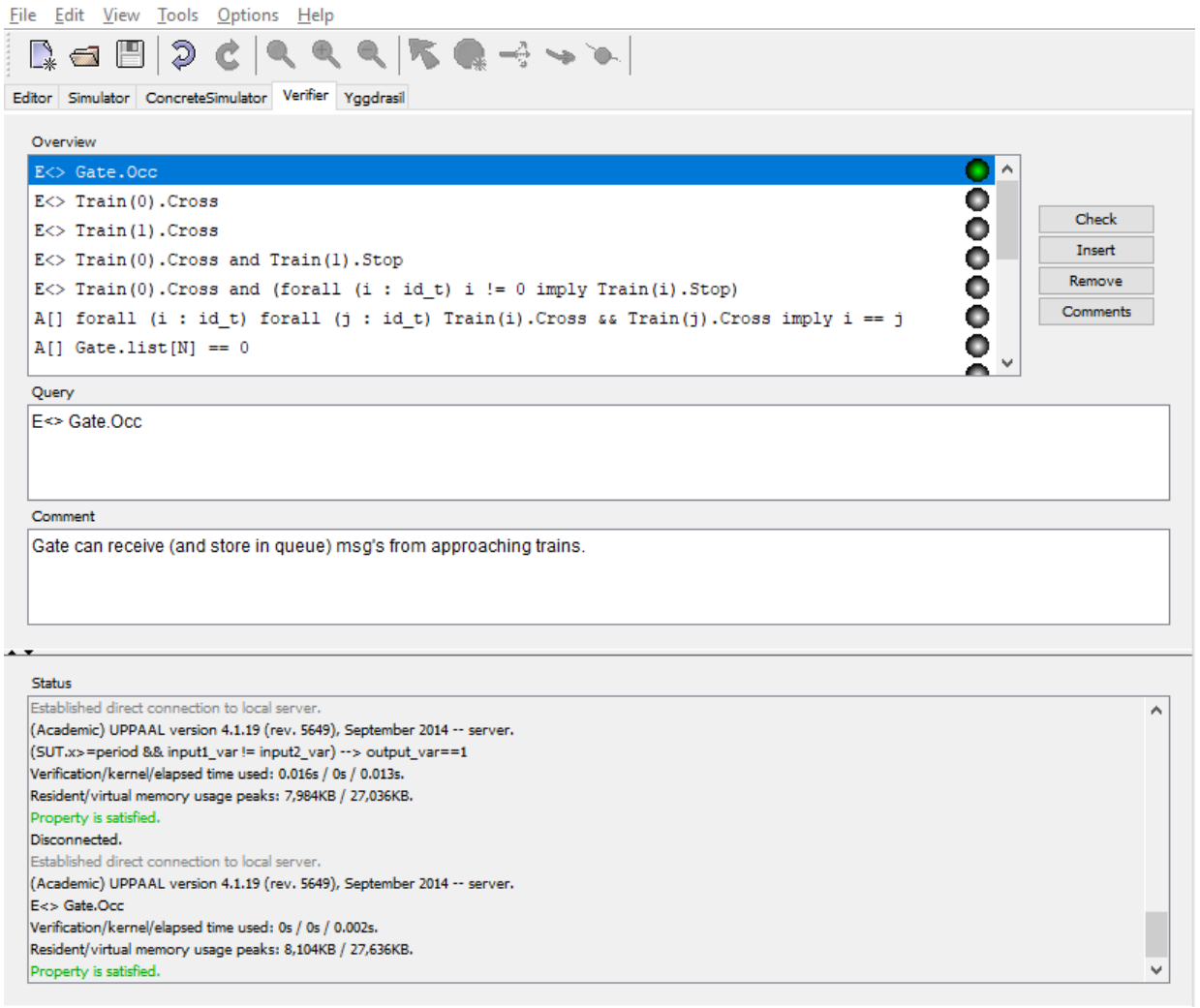
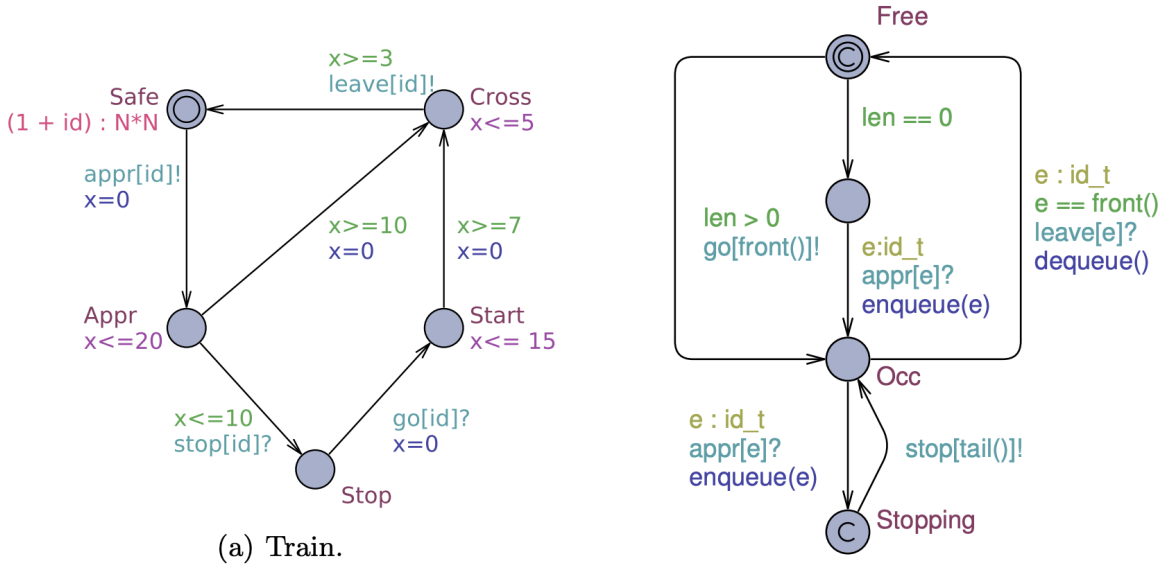


Figure 6. Verifier GUI of UPPAAL

UPPAAL SMC [6] 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.

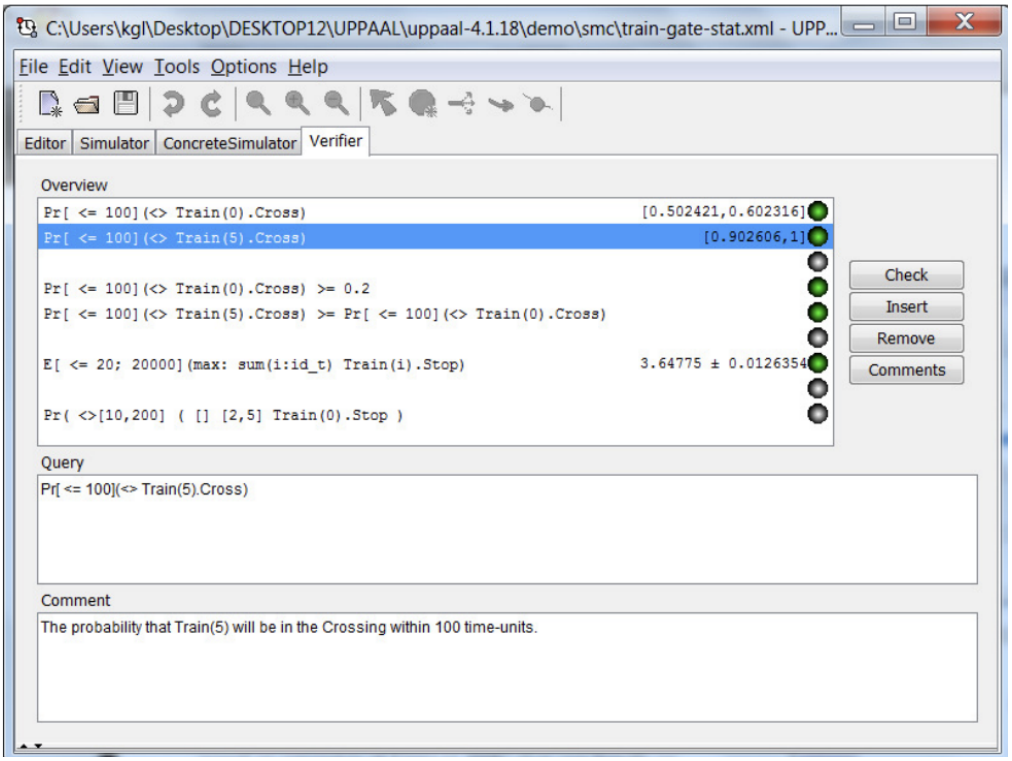


Figure 8. The Verifier of UPPAAL SMC and generating the cumulative probability distribution of an example property using statistical model checking.

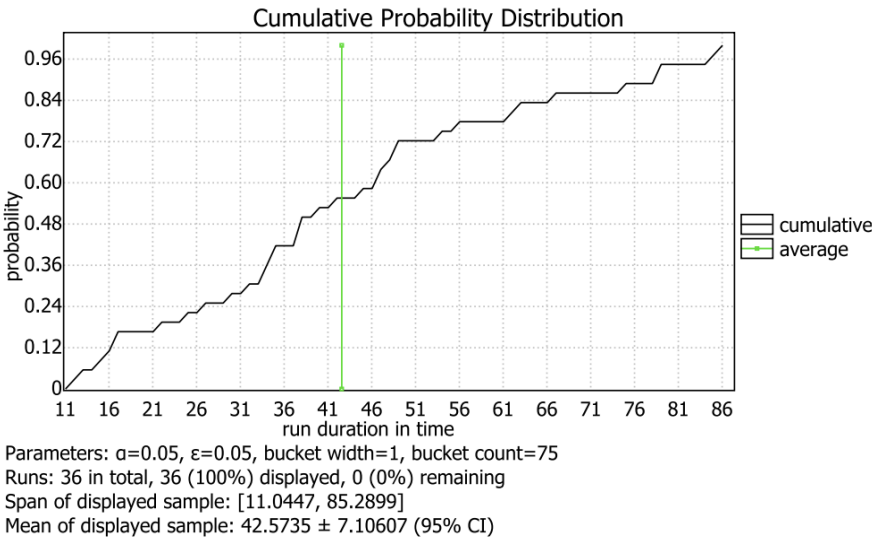


Figure 9. The generation of the cumulative probability distribution of an example property using statistical model checking.

The **benefits** of using UPPAAL SMC over the traditional UPPAAL verification engine include: handling of dynamical behaviours, 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 tool 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 licence 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/

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

3.1. General description (purpose, features, interfaces)

CompleteTest [7] 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 UPPAALl 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 [8]. 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).

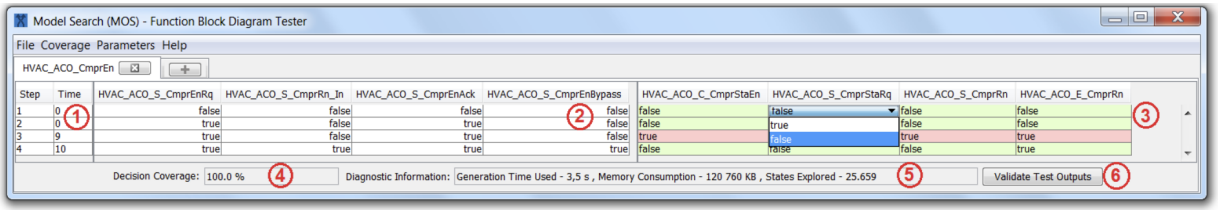


Figure 10. Graphical Interface of the CompleteTest Method and its Toolbox

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 behaviour of a Function Block Diagram (FBD) [9] 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 behaviours.

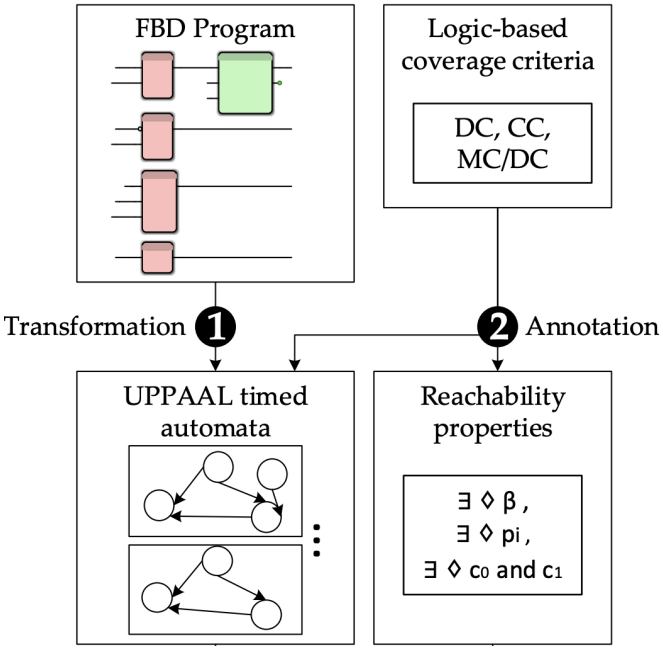


Figure 11. Model Transformation Methodology for CompleteTest.

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 modelled FBD program in UPPAAL, being an atomic communication between all timed automata.

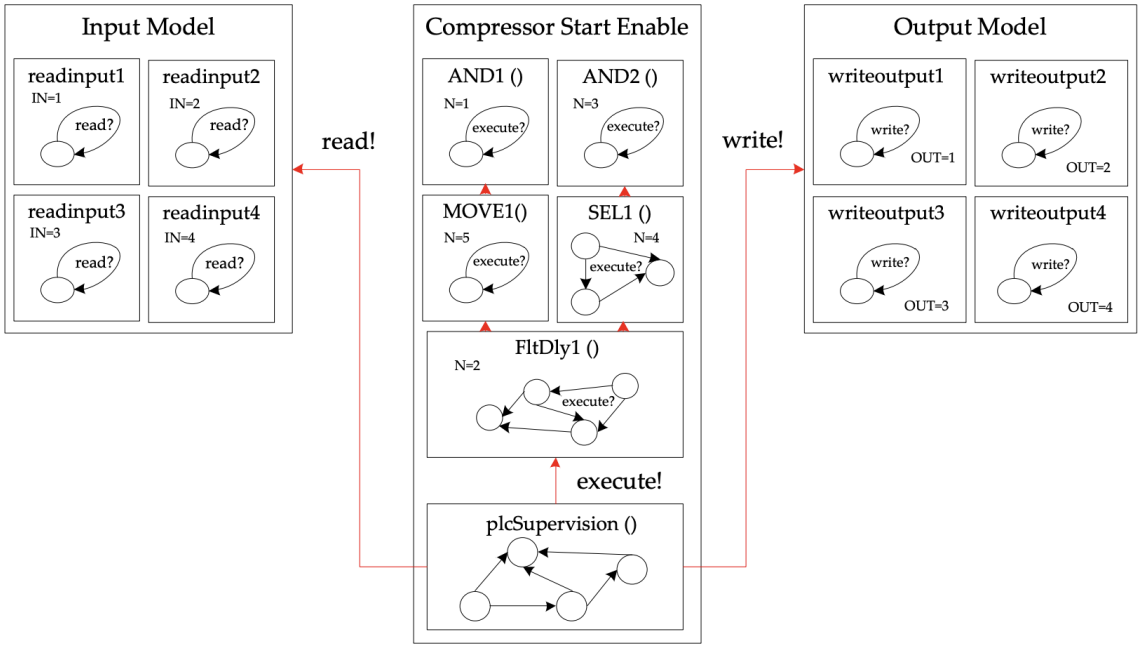


Figure 12. Overview of the Timed Automata Network obtained from CompleteTest for an example FBD program.

Once the model is generated, we can automatically check properties by characterising a logic coverage criterion as a temporal logic property or any security requirements specified as UPPAAL TCTL properties. We use the PROPAS 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 (The PROperty PAttern Specification and Analysis) and then used as formalised 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 formalisation 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 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 behaviour as explained. Since FBD programs allow the use of behavioural notations, we exploit this and specify the behaviour 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 will be targeted, but it is believed that the majority of case scenarios would benefit from the methods and tools discussed in this Deliverable. The SMC modelling 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 modelling with an environment model to simulate the potential security attacks.

For the FAGOR use case, the applicability of the tools is under investigation.

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.

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[2], 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.

4. Modelio

4.1. General description (purpose, features, interfaces)

Modelio is both an open source[3] and a commercial[4] modelling 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:

- Modelling and consistency check: UML [10], SysML [11], BPMN [12] 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 modelling. Modelio allows the usage of a specific (the most relevant) language combination but also checking related to specific language usage. For example, Figure 13 depicts data modelling of a web application.

[2] https://github.com/eduardenoiu/CompleteTest
[3] https://www.modelio.org/
[4] https://www.modeliosoft.com/en/

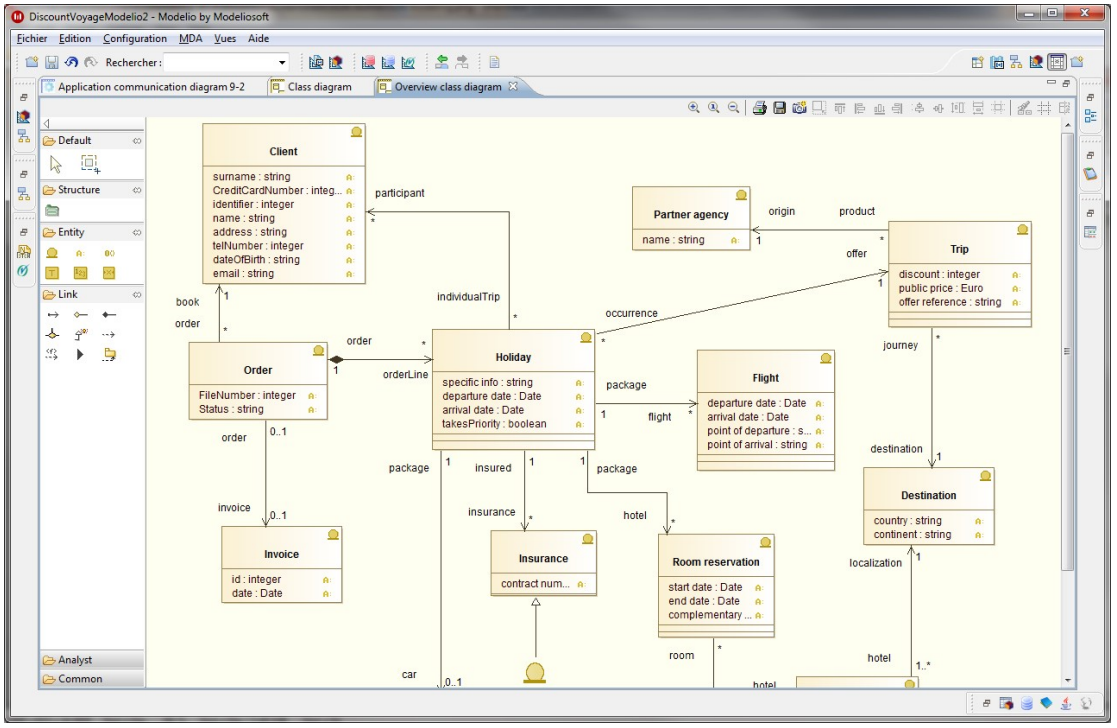


Figure 13. Example of the modelling under Modelio.

- 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 14).

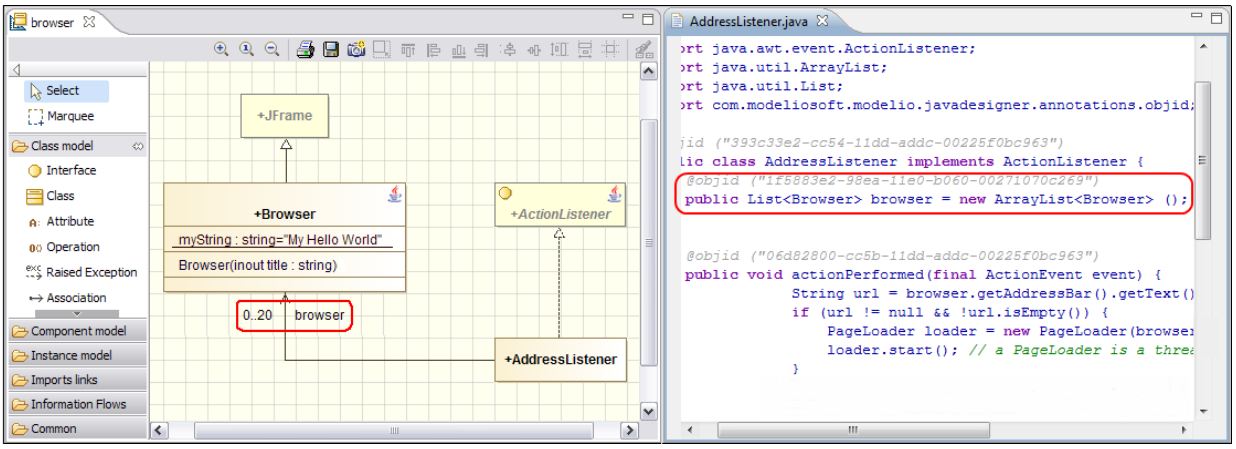


Figure 14. Example of the Java modelling (on left) and generated Java code (on right).

- Impact analysis and traceability: As already stated Modelio is able to provide several modelling 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 otr vulnerable for example (Figure 15) .

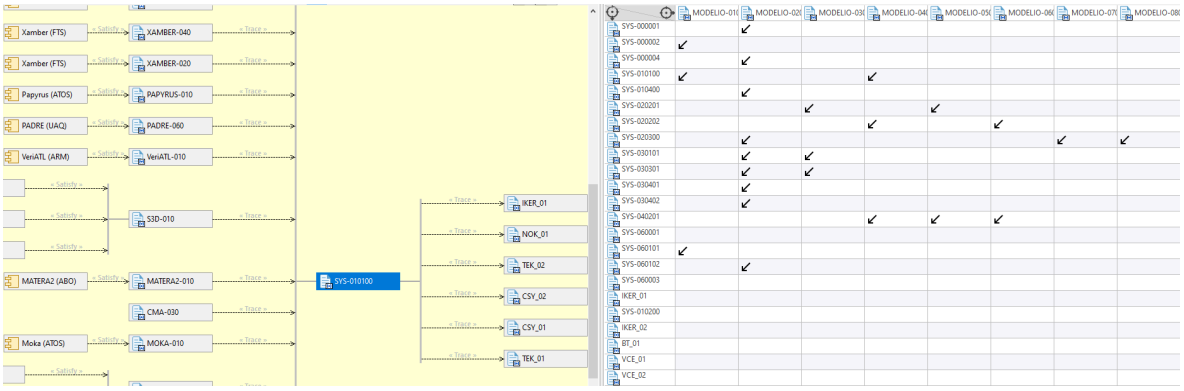


Figure 15. The Modelio link editor (on left) and the Modelio matrix (on right).

4.2. 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 [13] (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.

4.3. 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 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.

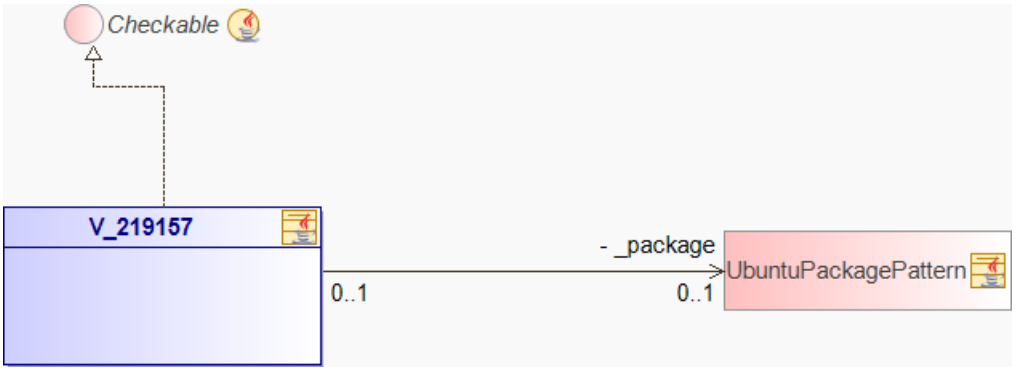


Figure 16. RQCODE UbuntuPackagePattern in UML with Modelio

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.

4.4. 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 modelling.

4.4.1. 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 you first run Modelio, you will get a welcome page with useful hints (Figure 16):

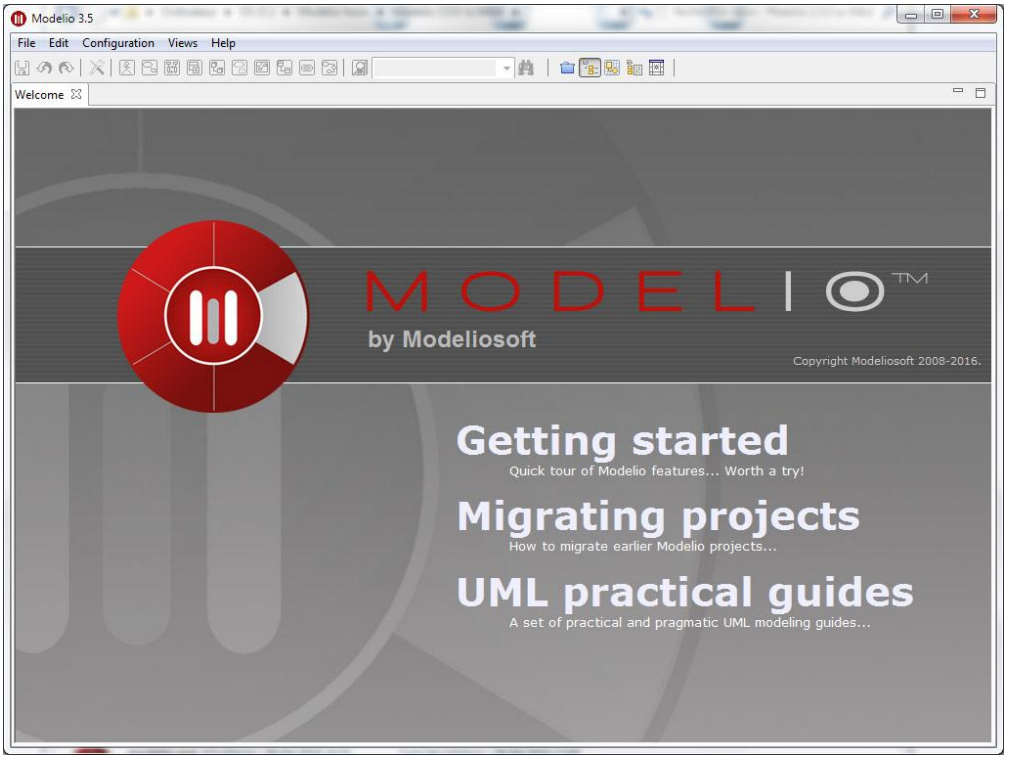


Figure 16. The Modelio welcome screen

Then one can create the first project.

4.4.2. 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.

5. SecureIF: Secure EFSM generation

SecureIF is a tool that can generate a secure specification of a system by combining security requirements formalised as Organization-based access control (OrBac) rules with a behavioural specification of the system extracted from functional requirements in the form of an Extended Finite State Machine (EFSM). Differently from CompleteTest, where the specifications are extracted from the source code, in SecureIF the specifications are created from functional requirements.

5.1. General description

5.1.1. Introduction

Since security is considered as a critical issue especially in dynamic and open environment systems, the system behaviour has to be restrained by a security policy that ensures that a certain level of security is always maintained. A security policy is a set of rules that regulates the nature and the context of actions that can be performed within a system, according to specific roles. As an example, such a policy can tackle the interactions between a network infrastructure and Internet or managing accounts and rights in an operating system or a database.

The main problem is that it is quite difficult to verify whether a system implementation conforms to a certain security policy. However, if one cannot ensure this conformance, global security cannot be guaranteed anymore. Most current work only concentrates on defining meta-languages in order to express security policies and provide unambiguous rules. In this section, we present typical examples of such generic policy description models. Indeed, they do not depend on the functional specification of the system. They suggest concepts to describe the security policy independently of the system implementation.

Once the security policy is formally specified, it is essential to prove that the target system implements this policy by (i) injecting this policy in the system considered or (ii) specifying formally the target system and generating proofs that this system implements the security policy or (iii) by considering several strategies of formal tests. In this section, we will investigate how to specify a secure model of a system target of a security evaluation. This secure model can be used for different purposes like model checking or model-based test generation etc.

5.1.2. The Key idea

In this work, we propose an approach that makes it possible to generate a secure specification of a system that can be used later for testing purposes. This approach manipulates three different inputs:

- A functional specification of the system based on a well-known formalism: Extended Finite State Machine EFSM.
- A specification of the security policy based on the Or-BAC [14] model that we wish to apply to this system.
- And finally an implementation of the system.

We want to obtain a new specification of the system that takes into account the security policy (we can call it: secure functional specification), and then to generate tests to check whether the implementation of the system conforms to the secure functional specification. Figure 17 describes the basic idea of the integration approach.

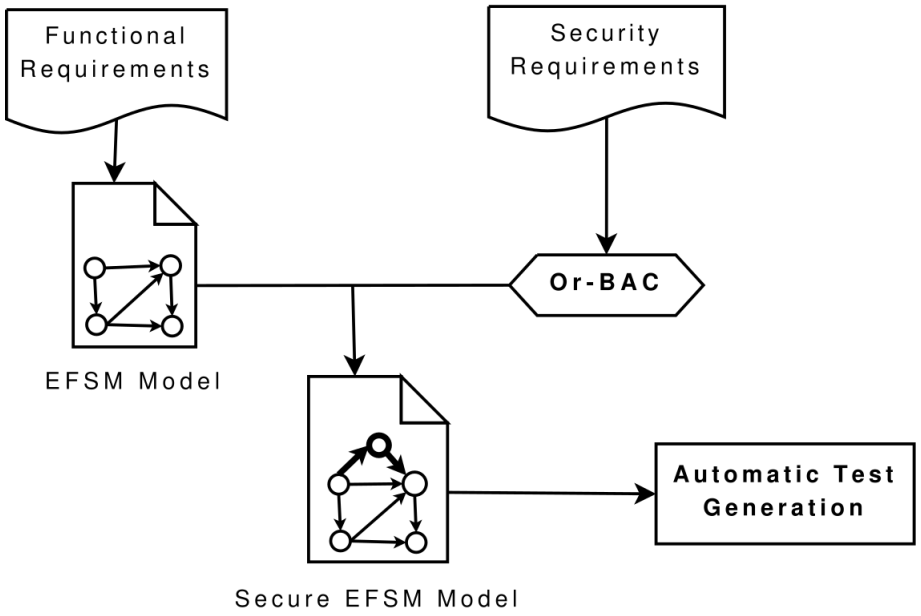


Figure 17. The Approach Basic Idea

Our approach is innovative since we propose to integrate security rules within the functional specification of a system. In general, the functional specification of a system and its security requirements are described in two different models and even implemented in two different modules that interact massively to ensure the system security respect. Thus, in our approach, we describe how modalities such as prohibitions, authorizations and obligations can be integrated in an EFSM based functional specification of a system by restricting predicates or by adding transitions and states to obtain a unique secure system model.

Checking the consistency of the security policy is out of the scope of this work. We assume that this issue has been checked. There are several techniques to achieve this goal, for instance the MotOrBAC tool [15].

5.1.3. Basics for Security Rules Integration

Assumptions

In this section, we define the relation between a security policy and the specification of a system and give the assumptions we rely on. First, we consider two parts in our approach: the initial system and the security policy. Initial system refers to the system functionalities with no security consideration. We assume that the specification of this system is verified with respect to an initial context and some functional requirements. Later, a new context can evolve to meet security considerations. Within this

new one, the initial system is not valid anymore since it cannot satisfy new requirements. It has to be completed with a security policy, to fit the new context.

In this approach, we propose to automatically integrate the security policy rules into the initial specification in the form of an EFSM, by way of a specific methodology. We take as an assumption that the specification of the security policy is correct. This means that we do not search within the policy for conflicts or redundancies. On the other hand, we consider that the rules have to be provided by an expert of the organisation, who guarantees their completeness and soundness. Finally, the last assumption concerns the semantics of the policy description language. We consider that it can always be translated in the form of an alphabet acceptable by the formalism. That means we consider that no information is lost while deriving the formal model from the set of rules.

Now that the assumptions are defined, we can introduce the basic concepts and describe our algorithm.

The EFSM Formalism

In order to model the initial system as well as the security policy, we choose to use the Extended Finite State Machine (EFSM) formalism. This formal description is used not only to represent the control portion of a system but also to properly model the data portion, the variables associated as well as the constraints which affect them.

Definition 1: An Extended Finite State Machine *M* is a 6-tuple *M = < S, s0, I, O, x, Tr >* where *S* is a finite set of states, *s0* is the initial state, *I* is a finite set of input symbols (eventually with parameters), *O* is a finite set of output symbols (eventually with parameters), *X* is a vector denoting a finite set of variables, and *Tr* is a finite set of transitions.

Definition 2: A transition *tr* is a 6-tuple *tr = <si, sf, i, o, P, A>* where *si* and *sf* are the initial and final state of the transition, *i* and *o* are the input and the output, *P* is the predicate (a Boolean expression), and *A* is an ordered set (sequence) of actions.

The Or-BAC Syntax

In this work, we choose to rely on the Or-BAC syntax to express the security policy (as an input). This choice is motivated by the fact that Or-BAC is an access and usage control model at the same time. It allows an organisation to express its security policy without any ambiguity. For this purpose, Or-BAC defines two abstraction layers. The first one is called abstract and describes a rule as a *role* having the permission, prohibition or obligation to perform an *activity* on a *view* in a given *context*. A *view* is a set of objects to which the same security rules apply. The second level is the concrete one. It is derived from the abstract level and grants permission, prohibition or obligation to a user to perform an *action* on an *object*.

Thus, according to Or-BAC syntax, a typical security rule has the following form:

Obligation (S, R, A, V, C)

This rule means that within the system *S*, the role *R* is obliged to perform the activity *A* targeting the objects of view *V* in the context *C*. The principle is similar for permission and prohibition.

We argue that starting from such simple syntax, it is easy to apply our approach to other languages such as Ponder[5] or Nomad [16]. In that manner, our framework remains independent of any model for security description. The only restriction is that some specific complementary conditions have to be taken into account in the formal specification of the security rules:

- The activity and the rule context have to be described in the same language of the functional specification of the system.
- A rule context is divided into two parts: an *EFSM context* with conditions related to the position in the EFSM (eg. input=signal1) and a *variables context* with conditions related to variables values (eg. variable=value).
- If the roles and *variables context* are not already defined in the initial specification, precise definitions have to be added (type, default value, etc.). This is a very important issue in our methodology because in general, the context expressed in Or-BAC rules describe conditions related to features outside the initial system. That is why, it is necessary to take them into account in the secure system and thus define precise methodology to access to their valuations or values.
- As an assumption, an activity within an obligation is considered as a *new* partial EFSM which starts with an obligation state *OS* and ends with one or many end obligation states *EOS*. All the new variables and signals have to be defined.
- An activity related to permission or prohibition must correspond to one transition at once (can be on several distinct transitions but not on a sequence). Otherwise, we cannot determine the predicate to be restrained.
- The *EFSM context* is mandatory in an obligation.

5.1.4. Integration Methodology

In this section, we define algorithms to automatically integrate the security policy rules into the initial specification in the form of an EFSM, by way of a specific methodology. The process is twofold. At first, the algorithm seeks for the rules to be applied on each transition of the specification and derives a secure automaton from this set of rules and the initial transition. Then, it integrates the automaton within the initial specification. At the end of the process, this integration will generate a new specification that takes into account the security requirements.

[5] http://ponder2.net/

It is possible that the security policy defines some new concepts that cannot be directly apprehended by the initial specification. In particular, a rule can express an activity that does not exist in the specification (new role, different object, new action, etc.). In that case, the new activity must firstly be created in the specification. That means some new states might be created in order to make the EFSM accept the new elements. In the security policy, this is specified by an obligation.

The beginning of the algorithm is the same for the three kinds of rules (permission, prohibition and obligation). It parses the EFSM specification and for each transition, it identifies the rules that map the activity (which can be a state, an input, an output, a task or a logic combination of these features) and the *EFSM context* in the case of permissions and prohibitions, and only the *EFSM context* in the case of obligations. If no rule maps the transition, the default one will be applied. Once rules are identified for each transition, we can proceed to their integration. Notice that several rules may apply to the same transition. In this case, the algorithm is recursively applied to each relevant rule.

Permissions Integration

The permissions are the easiest modality to integrate. Indeed, by definition, a permission does not define what is *possible* to do but instead, what is *permitted* to do. Thus, permissions relate to activities which already exist in the initial system. Considering the EFSM, permission corresponds to one (or many) transitions. If the transition related to permission contains no predicate, a predicate has to be added. On the other hand, if a predicate is already defined in the specification, it only needs to be further restrained (the condition is stronger). Notice that a permission rule (as a part of a security policy) allows to restrain an action and not to relax it. For example, if the system gives the permission to a subject *Bob* to perform the action *act*, this means that all the other subjects are not allowed to perform this action (of course, only if another permission to another subject to perform the action *act* is specified within the system).

Algorithm 1 Permissions Integration

Require: The transition Tr. The set of k permissions $permission_i$ that maps Tr where $(i \in \{1, ..., k\})$. Each $permission_i$ applies to a $role_i$ and possibly to a $variables\ context_i$ (may be empty).

1: **if** ($\exists$ predicate P associated to Tr) **then**
2: $P := P \wedge (\vee_i(variables\ context_i \wedge role_i))$
3: **else**
4: create predicate $P := \vee_i(variables\ context_i \wedge role_i)$
5: **end if**

Figure 18 gives an example. In the left transition, the system can pass from S1 to S2 if P is true, performing the task T, A and X are respectively the input that triggers the transition and its output. If

the permission involves a role R, allowed to perform task T in the context C where C is a variable context, the transition will be modified by strengthening the predicate, as shown in the right transition.

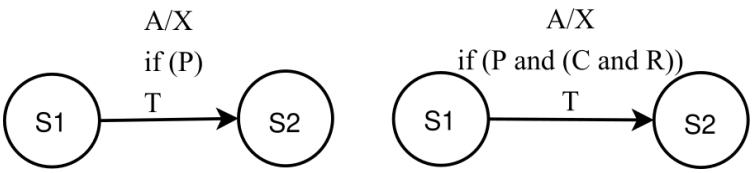


Figure 18: Permissions integration

Prohibitions Integration

Like permissions integration, prohibitions integration consists either of adding a new predicate or restraining an existing one (it becomes stronger).

Algorithm 2 Prohibitions Integration

Require: The transition Tr. The set of k prohibitions $prohibition_i$ that maps Tr where $(i \in \{1, ..., k\})$. Each $prohibition_i$ applies to a $role_i$ and possibly to a $variables\ context_i$ (may be empty).

1: **if** ($\exists$ predicate P associated to Tr) **then**
2: $P := P \wedge_i (\neg variables\ context_i \vee \neg role_i)$
3: **else**
4: create predicate $P := \wedge_i(\neg variables\ context_i \vee \neg role_i)$
5: **end if**

Here is an example in Figure 19. In the left transition, the system can pass from S1 to S2 if P is true, performing the task T, A and X are respectively the input that triggers the transition and its output. If the rule specifies that a role R is prohibited to perform task T in the context C, the transition will be modified by restricting the predicate, as in the right transition.

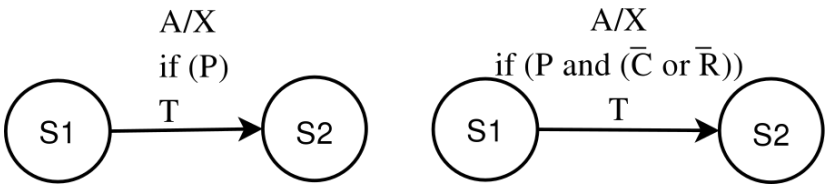


Figure 19: Prohibition integration

Obligation Integration

In our model, we take as an assumption that an obligation implies the creation of a new activity (if an activity already exists in the initial specification, it only has to be allowed or denied). This new activity describes a new functional feature of the system. To make this possible, the new activity has to be initially expressed through a partial EFSM so that the algorithm can perform an automatic integration

of the rule within the EFSM. In this manner, an obligatory activity is a partial EFSM which begins by a starting obligation state called *OS* and ends with an end obligation state called *EOS*.

Algorithm 3 Obligation integration

Require: The transition $tr =< S_1, S_2, A, X, P, t_1...t_n >$ that maps the obligation with an activity specified by the mean of an EFSM with OS as a first state and EOS as a last one. tr is denoted in the form of $<$*initial state, final state, input, output, predicate, actions*$>$

1: **for all** (transitions from OS) **do**
2: **if** ($\exists$ associated predicate Q) **then**
3: $Q := Q \vee (variables\ context \wedge role)$
4: **end if**
5: **end for**
6: determine the cut point $C_{ut}P_{oint}$
7: delete the transition tr
8: create transitions C_1, C_2 and C_3 such that
9: **if** ($C_{ut}P_{oint} == S_1$) **then**
10: $C_1 :=< S_1, OS, -, -, -, - >$
if ($\neg \exists\ EOS$ state in M) then $C_2 :=< EOS, S_2, A, X, P, t_1...t_n >$ end if
$C_3 :=< OS, S_2, A, X, \neg variables\ context \vee \neg role, t_1...t_n >$
11: **else**
12: **if** ($C_{ut}P_{oint} == A$) **then**
13: $C_1 :=< S_1, OS, A, -, P, - >$
if ($\neg \exists\ EOS$ state in M) then $C_2 :=< EOS, S_2, -, X, -, t_1...t_n >$ end if
$C_3 :=< OS, S_2, -, X, \neg variables\ context \vee \neg role, t_1...t_n >$
14: **else**
15: **if** ($C_{ut}P_{oint} == t_i$ where $i \in \{1, ..., n\}$) **then**
16: $C_1 :=< S_1, OS, A, -, P, t_1...t_i >$
if ($\neg \exists\ EOS$ state in M) then $C_2 :=< EOS, S_2, -, X, -, t_{i+1}...t_n >$ end if
$C_3 :=< OS, S_2, -, X, \neg variables\ context \vee \neg role, t_{i+1}...t_n >$
17: **else**
18: **if** ($C_{ut}P_{oint} == X$) **then**
19: $C_1 :=< S_1, OS, A, X, P, t_1...t_n >$
if ($\neg \exists\ EOS$ state in M) then $C_2 :=< EOS, S_2, -, -, -, - >$ end if
$C_3 :=< OS, S_2, -, -, \neg variables\ context \vee \neg role, - >$
20: **end if**
21: **end if**
22: **end if**
23: **end if**
24: minimize the resulting EFSM by deleting silent transitions (without input nor output nor action)

Thanks to the *EFSM context* of the obligation, the algorithm identifies the transition which will be split into two (pre/post transitions), to insert the partial EFSM of the obligation. Then, the algorithm needs to know how the components of this transition will be distributed relatively to the obligation (pre/post

transitions). This can be determined using the *cut point* that corresponds to the last component of the initial transition (state, input, task or output, but not a predicate) which maps the *EFSM context*. Each component until this *cut point* (included) will be attributed to the pre-transition (through the obligation) while other ones will be attributed to the post-transition. Finally, a last transition has to be added to bypass the obligation in the case the initial predicate is not satisfied (see Algorithm 3).

Figure 20 shows an example of the process. In this case, the initial transition is <*S1, S2, A, X, P, T*>. The new activity is a partial EFSM with two states (OS and EOS) and one transition characterised by the input B, the task T' and the output Y. According to the EFSM context, the *cut point* is the Input A. By the following, the transition C1 (pre-transition) is defined by the input A and the predicate P. The transition C2 (post-transition) is defined by the task T and the output X. Obligation integration is shown in the Algorithm 3.

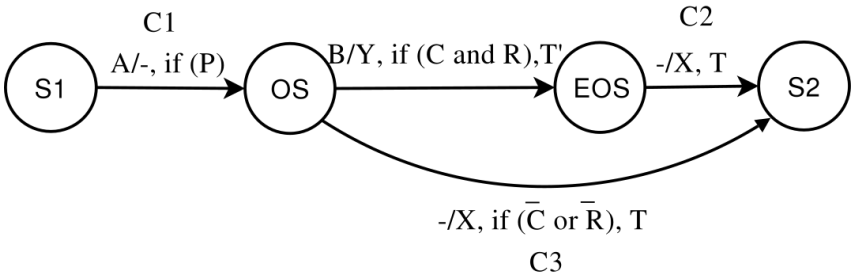


Figure 10: Obligation integration

Notice that in the case of an existing activity, two solutions can be applied. The first one is to create a copy of this existing activity (which is then in the form a partial EFSM). Our algorithm can be applied without any change. But the drawback of this solution is to copy the same transition many times which contributes to expanding the specification model size. The second solution consists of connecting the pre-transition to the existing activity (from its beginning denoted OS') and then connecting the existing activity (from its end denoted EOS') to the post-transition. We also need to create a new Boolean variable (with false as a default value) that is set to true after the pre-transition. Thanks to this variable, all the activities starting from the activity state OS' are insurmountable but the mandatory activity. Before the EOS' state, the new variable is reset to false.

Discussions

One can notice that the integration of rules requires few transformations on the EFSM. At worst, a permission/prohibition only implies to modify some transitions' predicates. In fact, the more costly operations are the obligations. As explained in the previous section, an obligation might lead to creating new states and consequently, several transitions. Nevertheless, even in terms of transitions, the complexity of the algorithm remains linear (in O(n)) since it is directly proportional to the number of rules in the policy. In practical experience, the modification of an initial system is not sizable since

the number of rules is not in general huge. In most transitions, only some changes in predicates are applied.

On the other hand, as a border effect, the order of the rules might have a great impact on the time consumed by the integration process. Indeed, the algorithm manages the rules one by one without analysing their relevance, since this role is devoted to the expert of the system. However, the complexity might slightly increase if the policy begins by resetting all rights in an open (everything is allowed by default) or closed (everything is forbidden by default) manner. This would be performed respectively by the following default rules:

$$\forall R, \forall A, \forall V,\ Permission\ (Website,\ R,\ A,\ V,\ --\ )$$
or
$$\forall R, \forall A, \forall V,\ Prohibition\ (Website,\ R,\ A,\ V,\ --\ )$$

R, A, V respectively refer to Role, Activity, and View aspect defined in OrBAC model.

5.1.5. Integration Result

The security rules integration allows us to obtain a new specification of the system that takes into account the security policy. This formal specification is described in the EFSM formalism that is a well adapted one to model communicating systems.

An implementation of this work has been done using IF formalism[6] that is an intermediate formalism based on EFSM. And it was possible to create a secure IF specification called SecureIF.

5.2. Relation to VeriDevOps and CaseStudies

The specification of several functionalities of VeriDevOps use cases is intended to be specified in Finite State Machines (or an extension of FSMs). Thus, the integration of security properties using this methodology or by being inspired by this methodology can allow to generate a secure model of the system that can be used to generate security oriented tests. This is basically the only tool in the project that allows the integration of security policies with the functional behavioural specification of the system under test.

This same idea can also be used by integrated attack behaviours in the model in order to generate negative tests denoting malicious behaviours.

[6] http://www-verimag.imag.fr/~sifakis/RECH/sdl_forum.pdf

5.3. Detailed overview for relevant usage scenarios

Test generation: Software Test and Automation (FAG_S1), From Requirements and Design to Test Specification (ABB_S2)

The main purpose of such methodology is to generate a secure model that can be the input of a test generation tool. But the same idea can also be adapted to generate a malicious model and perform penetration testing.

Security monitoring: Anomaly Identification and Handling of Safety and Security Requirements (ABB_S5), Known Anomaly and Vulnerability Identification (ABB_S6)

This secure model can also be a model that is used by a monitoring tool to extract security properties to be verified on traces or to be the reference for a pre-captured trace.

5.4. How to get it, install it, licensing

The security integration tool for an IF functional specification is a prototype solution developed by Dr. Wissam Mallouli during his PhD thesis. It is a linux based software. You can contact him by email for further details.

6. Conclusions

This deliverable briefly reviewed several tools for creating secure-by-design specifications. As mentioned in the introduction, this is an initial selection of tools to be evaluated on the use cases in the first phase of the project. The tools discussed provide important and yet complementary features, which cover different use case scenarios and make use of different specification and implementation artefacts available in each use case. For instance, in this project the CompleteTest and SecureIf tools are used for creating secure specifications of the system. On the one hand, CompleteTest creates the specification from PLC code and verifies its security properties using TCTL queries created from security requirements. On the other hand, SecureIf creates a functional model of the system starting from functional requirements and then it creates a secure specification by integrating security policies extracted from security requirements. At the other end, the Modelio-based approach allows one to express security requirements as executable object-oriented code which can be imported and analysed in the Modelio modelling tools. Also the tools have different purposes and applicability in the VeridevOps use cases. While CompleteTest and SecureIf are more suitable for state-based systems, the Modelio-based approach is suitable for non-state-based systems e.g., for enforcing systems based on technical security recommendations.

Additional tools will be eventually added on the list depending on the evaluation results in the future version of this deliverable, that is D4.4, which is planned for release in the second part of the project.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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

[10] 'OMG Unified Modeling Language'. [Online]. Available: https://www.uml.org/

[11] O. Casse, 'SysML: Object Management Group (OMG) Systems Modeling Language', *SysML in Action with Cameo Systems Modeler*. pp. 1–63, 2017.

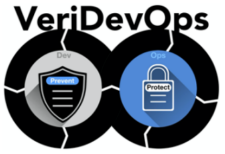
[12] OMG, 'Business Process Model and Notation (BPMN)'. [Online]. Available: https://www.omg.org/bpmn/

[13] A. Naumchev, 'Seamless Object-Oriented Requirements', in *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, 2019, pp. 0743–0748.

[14] K. Guesmia and N. Boustia, 'OrBAC from access control model to access usage model', *Appl. Intell.*, vol. 48, no. 8, pp. 1996–2016, Aug. 2018, doi: 10.1007/s10489-017-1064-3.

[15] A. Kassid and N. El Kamoun, 'Evaluation of a Security Policy Based on OrBAC Model Using MotOrBAC: Application E-learning', in *Advances in Ubiquitous Networking*, Singapore, 2016, pp. 129–139.

[16] W. Mallouli, A. Mammar, and A. R. Cavalli, 'Modeling System Security Rules with Time Constraints Using Timed Extended Finite State Machines', in *2008 12th IEEE/ACM International Symposium on*



Distributed Simulation and Real-Time Applications, 2008, pp. 173–180. doi: 10.1109/DS-RT.2008.22.