D4.2 Tools for Active Prevention During Development - Initial Version

Contract number:	957212
Project acronym:	VeriDevOps
Project title:	Automated Protection and Prevention to Meet Security Requirements in DevOps Environments
Delivery Date:	30/12/2021
Coordinator:	MDH
Partners contributed:	All
Release Date:	16.12.2021
Version:	01
Revision:	MDH, ABO, MI
Abstract:	The deliverable will report on technologies for active prevention through the use of automated test generation. The main focus in this deliverable will be on technologies for verifying that the implementation satisfies the security-related properties specified in the design phase. In addition, it will report on the tools for measuring the quality of the tests at specification and code level.
Status:	PU (Public)

Table of Contents

Executive Abstract

Tools for active prevention during development can be used for automated test generation targeting security aspects. This is a mature area that has seen a lot of research resulting in many test automation methods and tools. In this report, we first show a taxonomy that characterizes the methods for active prevention using automated test generation and instantiating this to the security context. The taxonomy can support researchers and practitioners to identify, compare and evaluate automated test generation methods in VeriDevOps. The resulting dimensions characterize automated test generation and its use in software security testing. We demonstrate the use of the taxonomy by applying it to several automated test generation tools from VeriDevOps. We overview these tools, in their initial version, and their applications to VeriDevOps use-case scenarios.

1. Introduction

If software security testing is severely constrained, this typically means that less time is devoted to manually designing highly effective test cases during development. As a solution to this problem, tools for active prevention (such as automated test generation techniques [1]) have been proposed to complement manual testing and allow some test cases to be created with less effort. To introduce such tools for active prevention using automated test generation, we present a taxonomy and instantiate it using different tools used in VeriDevOps. *We focus on active testing tools targeting software security [2]. Active testing tools avoid known categories of bugs that allow explicit attack techniques.* These are well-understood *software implementation bugs* that can show substantial disruption in the behavior when an attacker initiates. In addition, *these can break the software's security goals based on particular security requirements*. These bugs aimed to be detected by test generation tools are also called implementation vulnerabilities.

Over the last decades, it has been a problem for both software professionals and researchers to develop effective, applicable, and practically relevant test generation techniques and tools [1], [3]. With so many approaches in automated test generation, there is a risk of not assessing and adopting properly these in practice, which makes it harder for practitioners to choose and use these tools. Many software testing standards relate to testing techniques and test automation (e.g., ISO/IEC/IEEE 29119[1], OMG UML Testing Profile[2], TTCN-3[3]). Even if these standards are relatively recent, they lack the depth needed to deal with the sheer number of techniques used by automated test generation approaches. In addition, the software security aspects are not tackled in an explicit manner. We propose a generic process that can be used for active prevention.

2. A Generic Test Generation Process and Taxonomy for Active Prevention

The process of automated test generation aims to find suitable test cases using a description of the test objectives that guide towards a certain desirable security-related property. These test cases could contain parameters to start the software, a sequence of steps and inputs, and the timing when these steps should be supplied. In some cases, test cases might need to contain some other type of information for a complete execution and evaluation of the system-under-test.

[1] https://www.iso.org/standard/45142.html
[2] https://www.omg.org/spec/UTP/1.0/PDF
[3] http://www.ttcn-3.org/

Since we did not limit to techniques generating test data fully automated, we focused on the only secondary study [1] on automated test generation techniques regardless of their input software artifacts. Based on this study, we chose to scope our focus to the following test generation categories: structural testing, model-based testing, combinatorial testing, random testing and search-based testing applied to software security. We reflected on our own experience in automated test generation and used existing taxonomies for security model-based testing [4], [5]. In Figure 1.1, a typical setting for active testing tools is identified based on the testing process outlined by Utting et al. [4], [5] and extended using the methodology categories for automated test generation [1].

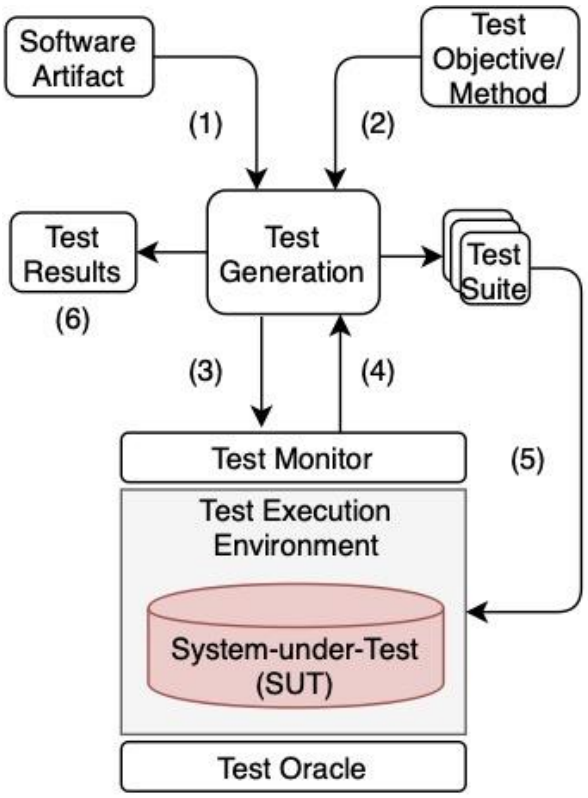


Figure 1.1 The Generic Test Generation Process in VeriDevOps. A test objective, in this case, relates to different software testing goals targeting software security.

A generic process of automated test generation proceeds as follows:

- Step 1. A software artifact is used or created for the purpose of guiding the test generation. A software artifact is either a specification of what the System-under-Test (SUT) should do or the actual code of the SUT in different forms (e.g., source, executable code).
- Step 2. A test objective and method formally encodes the test criteria and describes how the test generator should choose the resulting tests. This can relate to the structure of the

software artifact (e.g., code or model coverage), random test goals, or to vulnerability-based objectives.

- Step 3+4. A test suite is generated by running the software over many possible executions using a certain method. Each method needs to monitor if the test objectives are met, which can be achieved during test generation or just for test execution and evaluation of the test results. This can be invasive (e.g., at code level) or non-invasive by focusing on the available external interfaces.
- Step 5. Once Steps 1 to 4 are completed, a test suite is executed by running the software in an online or offline manner.
- Step 6. The test results are based on the test evaluation that compares the actual outputs of the SUT with the expected outputs as provided by the oracle such that the test results are generated.

Test generation approaches can be quite different, but all of them have common underlying dimensions that can be quite helpful when adopting these techniques in a certain software development project when tackling software security. Given the generic test generation process shown in Figure 1.1, we identified five dimensions corresponding to these steps (i.e., Software Artifact (Step 1), Test Generation (Steps 2, 3 and 4), Test Execution (Step 5) and Test Oracle (Step 6). Even though there can be other steps in software security testing used in practice, we argue that the identified steps are most commonly conducted when using automated test generation in VeriDevOps and the industries targeted by this project. This taxonomy of automated test generation approaches has five categories shown in Figure 1.2. This gives an overview of the taxonomy where the tree leaves indicate options that are not incompatible (for example, some approaches may use more than one generation objectives).

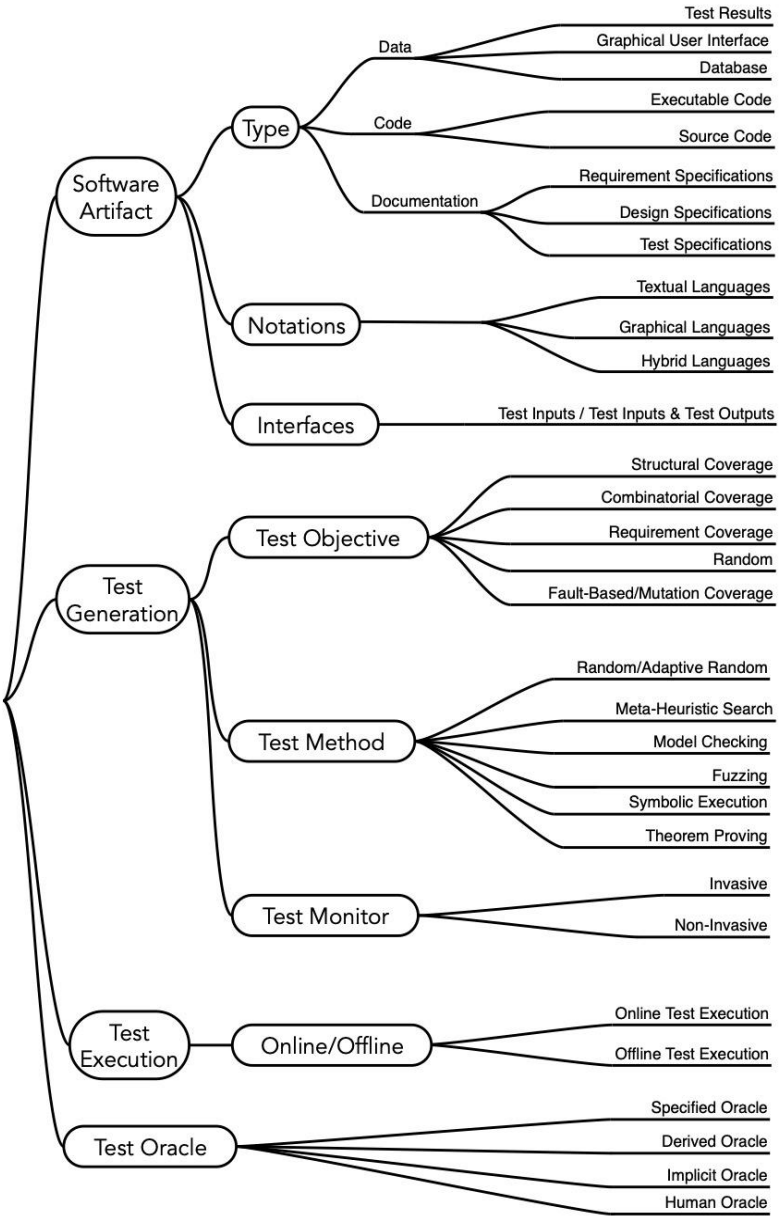


Figure 1.2 Overview of the Test Generation Taxonomy used in VeriDevOps

2.1. Software Artifact

Creating the software artifact is reflected by the three dimensions within the software artifact category: type, notation, and interface. These are used in the software artifact type categorization [6] and the model-based testing taxonomy [5], [6]. The first dimension refers to the type of software artifact used as an input to the test generation process. Software is multidimensional and consists of a variety of artifacts related to data (i.e., test results, graphical user interface, database), code (source or executable), and documentation (i.e., requirement, design, and test specifications). The second

dimension refers to the notation style used to describe the software artifact. Many different notations have been used for representing the expected or the actual behavior of software artifacts and their related security aspects. To differentiate between language styles [7], we group them into textual, graphical and hybrid languages. The last dimension relates to the input-output interfaces [5], [6], which answers the following question: does the software artifact specify only the test inputs (or the sequence of test inputs), or does it specify the input-output behavior of the SUT? This is an "x/y" tree leaf alternative that indicates incompatible alternatives. For example, combinatorial test generation tools will be used to represent test inputs and do not specify the expected outputs.

2.2. Test Generation

Three dimensions under this test generation category define the test objectives used to select the generation of test cases as well as the monitoring of these objectives. Since this category partially reflects the test generation dimensions in the model-based testing taxonomy [8] [5], we use their categorization for reflecting which kinds of test objective criteria and monitors they support. These criteria are structural coverage, combinatorial coverage, requirement coverage, random, and fault-based/vulnerability/mutation coverage. In addition, we classify the test generation method that is used to derive test cases using graph search algorithms, meta-heuristic techniques, model checking, symbolic execution, theorem proving, and fuzzing. In addition, the test monitor dimension refers to the types of monitoring needed to evaluate if certain test objectives are met during test generation. This dimension is based on the concretization adaptor in the process of model-based testing [5] and the needs of different automated test generation methodologies and can be of two types: invasive and non-invasive. Automated test generation can use instrumentation at code and internal interface levels for invasive monitoring. However, when this is not needed or is not practically possible, automated test generation uses a non-intrusive instrumentation activity that can always be measured static or using external interfaces. For example, the coverage information can be obtained from the SUT source code for code coverage-based test generators. In many cases (e.g., embedded systems), this kind of instrumentation could impose an overhead that may alter the program execution behavior. Therefore other non-invasive monitoring techniques can be used.

2.3. Test Execution

The test execution dimension is concerned with how test case generation relates to test execution. This dimension is also used in the model-based testing taxonomy [5]. Test execution is done either online or offline from the test generation. Some tools support both (e.g., model-based testing tools such as GraphWalker).

2.4. Test Oracle

The last category relates to how automated test generation tools determine whether a given test case is acceptable or not. This should not be confused with the abstract information contained in the security requirement and design specification. An oracle is an implementation of a specification and is used to judge the correctness given the generated test data. We use the categorization of test oracles of Barr et al. [9] in which the test generation tools can use specified oracles (formally specified models), derived oracles (derived from the software artifacts or system executions), implicit oracles (by relying on general, implicit knowledge to distinguish between a system's correct and incorrect behavior), and human oracles when a human being is checking the results of the generated test cases. For example, metamorphic testing [9], [10] is using derived oracles out of metamorphic relations that must hold among different software executions.

3. Classification Criteria For Active Software Security Testing

Felderer et al. [4] focused on the following security aspects to select security test cases and the available evidence of using these approaches: filter criteria and evidence criteria. The first criteria include the specification of the system security model, the security model, and explicit test generation and selection criteria. In combination, these artifacts determine security-specific system traces that are of interest for software security testing purposes.

Furthermore, these criteria address specific techniques of security vulnerability testing, for example, risk-based testing, fault-injection, fuzz testing, and vulnerability coverage. In addition, evidence criteria relate to the industrial applicability and utility of automated test generation approaches. Figure 1.3 gives an overview of these criteria and their dimensions.

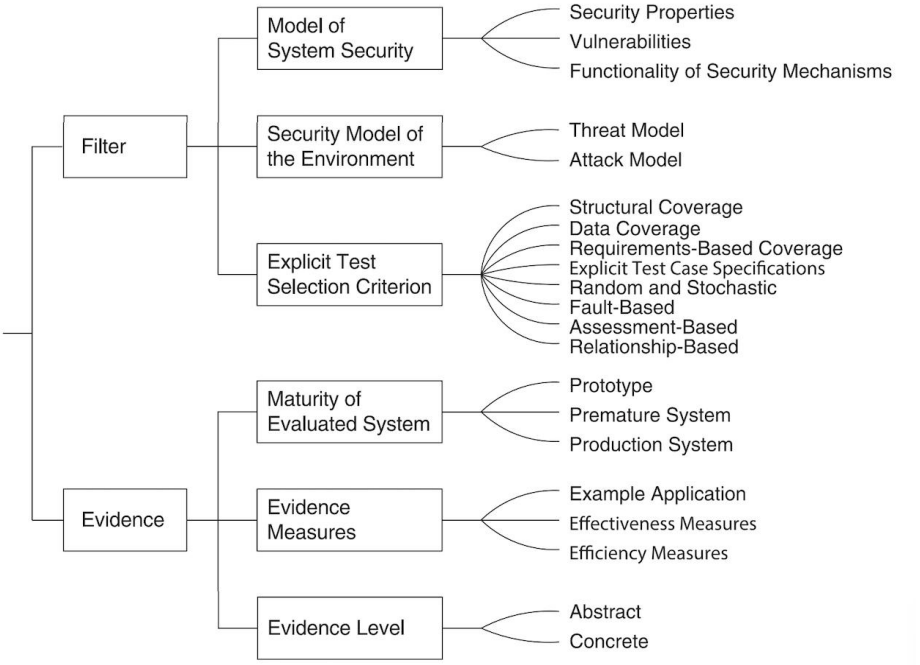


Figure 1.3. Security testing classification criteria [4] connected to our taxonomy.

System security models (shown in Figure 1.3) are partial system artifacts that focus on security aspects. These models consist of security properties, vulnerabilities, and functionality of security mechanisms. Examples for models of vulnerabilities include security-related mutation operators or logical mutation operators connected to software security testing.

Security artifacts of the environment are related to the security aspects related to system behavior's causes and potential consequences. These could be of two types (as shown in Figure 1.3): threat models describing threats and attack models describing sequences of actions to exploit vulnerabilities. Attack models may be testing strategies such as fuzzing that generate test cases.

The test selection criterion relates to the aspects of test generation mentioned in Section 2.3. For example, in mutation security software testing, the mutants are similar to vulnerabilities, and the generated tests simulate attacks.

Evidence criteria (and shown in Figure 1.3) are used to show how much evidence is available concerning the practicality of an active software security testing approach. As evidence has several aspects, these criteria incorporate maturity of the evaluated test generation approach and the evidence measures when applying these in practice. The criterion maturity of the evaluated active security testing approach captures the degree of deployment of the system under test used to assess the test generation approach (e.g., prototype, premature system, and production system). In addition, we use evidence measures to specify the qualitative or quantitative assessment criteria, e.g., fault detection. The number of found vulnerabilities or mutations is of interest to security testing objectives.

4. Active Prevention Tools and Their Application

4.1. CompleteTest - CODESYS Edition for Test Generation

CompleteTest [11] is a technique in which the input model is annotated with coverage information, and the checked properties are generated as a single timed test sequence. Unlike other methods, CompleteTest delivers a strategy to develop test cases for various code coverage criteria (as described in Section 2) instantly appropriate to industrial control IEC 61131-3 software. In CompleteTest, the UPPAAL model-checker is used for automated test generation using code and mutation goals. For a thorough summary of testing with model checkers, we direct the reader to Fraser et al. [12].

CompleteTest is a tool for automated test generation founded on model checking and timed automata. CompleteTest can use code coverage measures to develop test cases (as shown in Figure 1.4), provide a method for the user to choose a program, create tests for a sample of coverage criteria, visualize the developed test information, and define the correctness of the outcome made for each generated test by directly comparing the actual test result with the expected output (as delivered manually by the tester). CompleteTest joins as a client to the model checker and confirms properties against the model. A trace parser gathers a diagnostic trace from the model checker. It outputs an executable test case including inputs, current outputs, and timing details (i.e., the time parameters). In addition, CompleteTest was expanded to support mutation testing (as covered by the test generation taxonomy in Sections 2 and 3), a technique for automatically generating faulty implementations of a program to improve the fault detection ability of a test case. These faults are related to logical faults that an attacker can use to enable specific attack techniques.

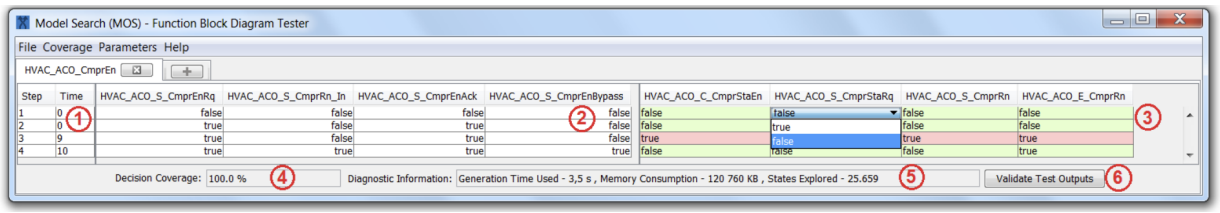


Figure 1.4 Graphical Interface of the Toolbox

A series of studies [13]–[16] based on industrial use cases shows the relevance of using automated test generation in practice. These results indicate that automated test generation is efficient in generating tests and scales well for industrial IEC 61131-3 FBD software. Automated test generation can reach comparable code coverage as manual testing conducted by human subjects but in relatively little testing time. These investigations support the conclusion that automatically generated tests are narrowly worse at uncovering defects and vulnerabilities in terms of mutation scores than manually created test cases. These results cover the Evidence category of the taxonomy shown in Section 3.

The primary purpose of the user interface layout is to fulfill the actual requirements of an industrial tester. Although there is an option for fine-tuning the configuration parameters of the

underlying UPPAAL model-checker, most of these are set to default values, making the tool directly ready for use upon startup. Figure 1.5 depicts menu options for the toolbox, listing chosen default values for the parameters and the coverage criteria. These activities cause the tool to develop a set of test cases that cover all branches. The attempt continues until all branches have been covered, or the tool runs for 10 minutes even if branches are still not encountered. We discovered that when the tool is applied to the existing programs, the model checker can generate tests in a couple of minutes.

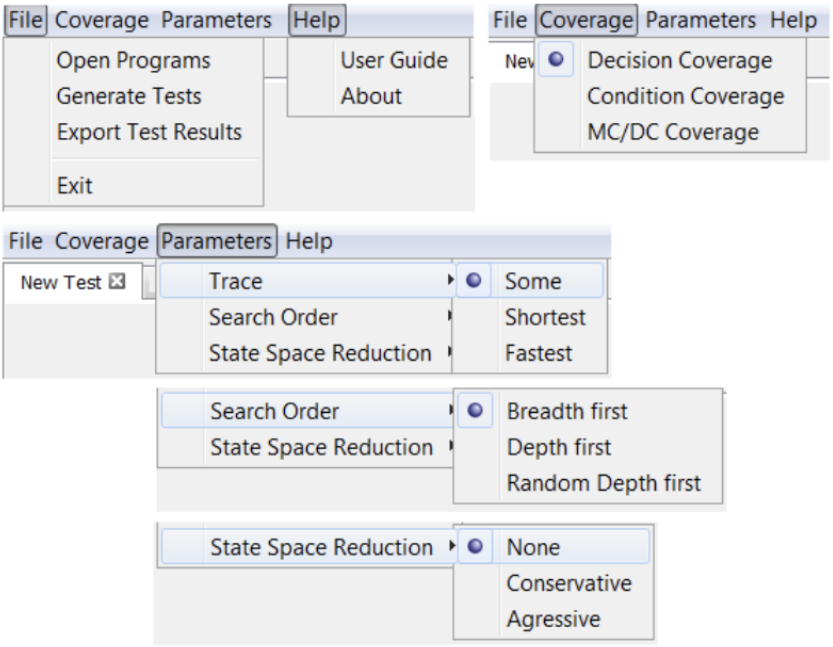


Figure 1.5 User Features of CompleteTest.

Application to the case studies in the VeriDevOps project: CompleteTest is used on the ABB use case by targeting the integration with CODESYS development environment during the unit and integration testing. It focuses on prevention at the design level using FBD IEC 61131-3 programs by the generation of test cases using adaptive random testing and fuzz testing targeting the detection of code vulnerabilities or faulty behaviors prone to security issues. The idea is to guide the randomized creation of new test inputs by feedback about the execution of previous inputs and avoid redundant inputs. In addition, we generate illegal inputs that can be used in the ABB case study as well as ways to measure effectiveness in terms of mutants detected.

Basic Test Model Generation and Test Generation using Random Testing and Fuzzing

Figure 1.4 shows the result of the tool for this use case executed. The figure illustrates several kinds of information displayed to the tester in a table with the test inputs and outputs (points 1,2,3 in Figure 1.4) and a set of additional data and actions (points 4, 5, and 6 in Figure 1.4). The numbered attributes required to be observed in this scenario are also shown in Figure 1.4:

1. Steps and Timing details concerning the specific test data supplied to the running program.
2. Developed test data required to achieve the highest coverage of the given program or using a random generation of test cases.
3. Editable area of the test data, where the tester can supply desired outputs for a specific set of inputs based on a specified behavior in the requirements. To support the efficient use of the area in the tool, expected results are supplied in a drop-down preference list for boolean values (true/false) or as a text area for other non-boolean values (integers, doubles, etc.).
4. Percentage of the code coverage achieved by using the generated tests.
5. Diagnostic data concerning the time spent generating tests, memory usage, and state-space size.
6. Optional step to compare expected values with the actual ones. Pressing the ”Validate Test Items” switch drives the test data entries to be colored with green where the desired values match the computed one and red where there is a mismatch. Any subsequent updates to the expected test data will automatically correct the coloring of that entry.

The initial set of test inputs is helpful to create an initial test case set for fuzzing. Then, depending on the scan cycle length, a variable number of inputs are generated within a specified time frame and given, we can cover the code. CompleteTest uses white-box fuzzing, where the FBD internal structure is analyzed to generate appropriate input values. We use a white-box fuzzing technique by using model checking as the search engine and execution environment (by choosing Random-Depth Search in Figure 1.5). It executes an FBD with concrete random input values. The CompleteTest-based fuzzer can ensure that these input values drive the program to a different FBD execution path, thus improving coverage.

Vulnerability Detection in FBD Programs

This scenario compares the expected values and computed values produced by the program. Let us assume a typical fault in a program, corresponding to a removed negated signal that can keep an output stuck at a certain value. Then we generated tests for both the original program. Random test generation and fuzzing in CompleteTest are used to discover software security vulnerabilities. The goal of running the FBD through a test generation campaign is to find bugs that violate the specified security policy. For example, a simplistic security policy employed by CompleteTest is to test only whether a generated sequence of test inputs—the test case— is crashing the system. Other policies will be developed. The specific mechanism in CompleteTest that decides whether an FBD program execution violates the security policy is called the CompleteTest vulnerability oracle.

Fault detection is calculated using mutation analysis. We used our FBD mutator tool implementation to generate faulty versions of the FBD programs. Mutation testing is the technique of creating incorrect program implementations to examine the fault detection ability of a test suite. A mutant is a new program version created by slightly modifying the original program. For example, a mutant is made by replacing an interface or block with another, negating a signal, or changing the value of a constant. The execution of a test on the resulting mutant may create a different output as the original one, in which

case we say that the test kills that mutant. A mutation score is calculated by automatically implanting mutants to estimate the mutant detecting capacity of the written test. We compute the mutation score using an output-only oracle (i.e., expected values for all of the test outputs) against the set of mutants. The following mutation operators are used during mutation analysis: Logic Block Replacement Operator, Comparison Block Replacement Operator, Arithmetic Block Replacement Operator, Negation Insertion Operator, Value Replacement Operator, Timer Block Replacement Operator.

4.2. SEAFOX CODESYS Edition for Combinatorial Security Testing

Combinatorial testing (as one of the test objectives shown in the taxonomy from Section 2) can be used for effective security testing. In VeriDevOps, modeling of vulnerabilities using a combinatorial approach is specific to the industrial domain and the identification of factors triggering vulnerabilities is not an easy task. The SEAFOX combinatorial testing tool has been developed in VeriDevOps and can take a PLCopen XML file as an input that contains information about the program to be tested in the CODESYS development environment[4]. Then the tool parses the file by extracting the needed information that will be used to generate test cases, such as, the name of the input parameters and their respective data types. Another option is to provide the parameters and data types in the tool manually. Further, SEAFOX tool creates test cases by generating the inputs using one of the available algorithms implemented in the tool: Random, Base choice, or Pairwise algorithm. These input values can then be exported as comma-separated-values in a CSV file where each line is a new set of input parameters representing a certain test case.

SEAFOX has a graphical user interface (GUI), as shown in Figure 1.6, where the user starts by loading a folder with the desired program or function block stored as an XML file. Alternatively, the parameter names and data types can be added manually. A limitation in the tool when importing an xml-file is that the file only can contain one function block with input parameters. If it contains two or more function blocks with input parameters, the tool will assume that all parameters can be combined in a test case. After choosing the file to import, input variables from the file are transferred to the tool and visualized in the middle container of Figure 1.6. Below this container, the user chooses which combinatorial algorithm to use – Random choice, Base choice, or Pairwise. Then, a range for each parameter as well as other additional information depending on the chosen algorithm needs to be set before the test cases can be generated. The ranges specify the values each parameter can take and is either a single value, a closed interval, or a combination of both. Examples of ranges and the notations used in the tool:

- 7= a single value of only 7.
- 1_3 = a closed interval (1 and 3 are included).
- 1_3;7 = a combination of both.

[4] https://www.codesys.com/

Once all this is set, test cases can be generated and displayed in the right container. Lastly, the user can export the test cases as a CSV-file by clicking the Save test button. An example of test cases generated using Base choice is depicted in Figure 1.6.

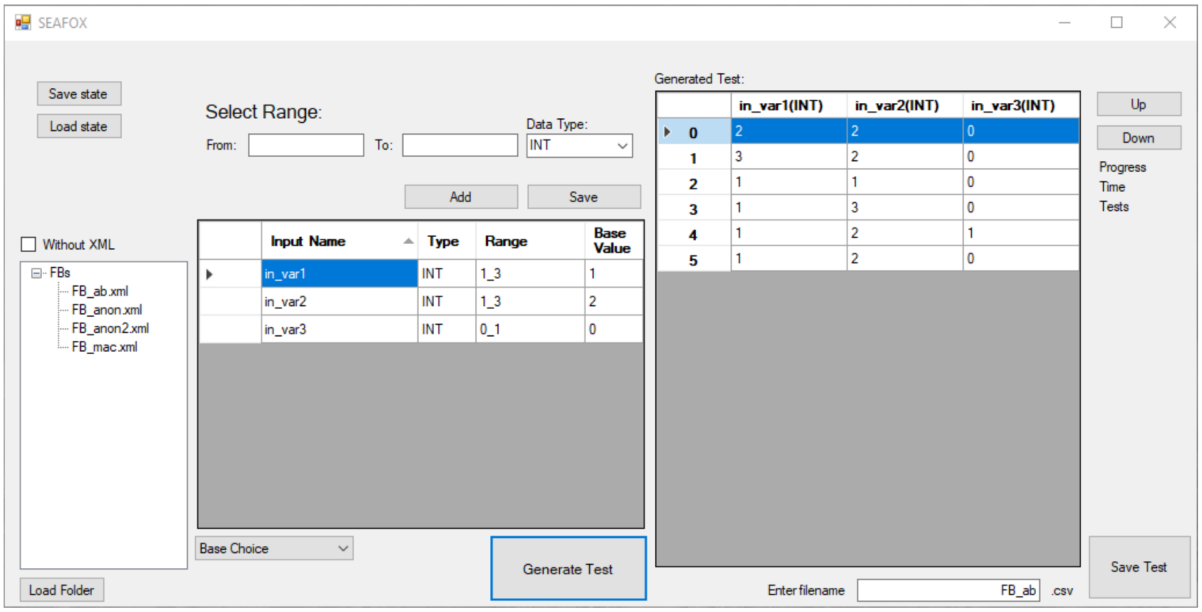


Figure 1.6. Base choice test suite generated in SEAFOX. The test cases have three parameters of integers, of which each has a range and base choice-value set as shown in the middle container in the image. The generated test cases are located in the box to the right.

When the SEAFOX tool has generated test case input values, they are saved in a csv-file. To be able to use these values for automated test case creation, SEAFOX is integrated with CODESYS by taking the content of the csv-file into the respective test cases that would be created based on a template from the previously exported PLCopen XML file from CODESYS. Test cases are executed using CfUnit tool.

Application to the case studies in the VeriDevOps project (1): SEAFOX is used on the ABB use case by targeting the integration with CODESYS development environment during the unit and integration testing for both functional and security testing. It focuses on prevention at the design level using FBD IEC 61131-3 programs by the generation of test cases using combinatorial testing targeting the combination of random testing and functional testing for detection of faulty behaviors prone to both functional and security issues based on certain requirements.

The usage scenario of this tool is explained in the form of steps, as shown in Figure 1.7. After selecting programs and specifying input ranges, the first step is to load these programs (written in FBD and ST programming languages) into CODESYS IDE and then create the necessary test suites using CfUnit for the FB that were to be tested (step 3). Each test suite is able to hold several test cases, where the input parameters for those test cases were generated through the SEAFOX tool (step 4), and the expected

outputs were provided manually. In the next step, the exported csv-file from SEAFOX and the file with the test suite from CODESYS are processed within the script to create a new test suite containing test cases with all test case inputs from the csv-file as a PLCopen XML file (step 5). This file was then imported back to the CODESYS programming environment (step 6), where tests were executed with the selected testing tool CfUnit (step 7).

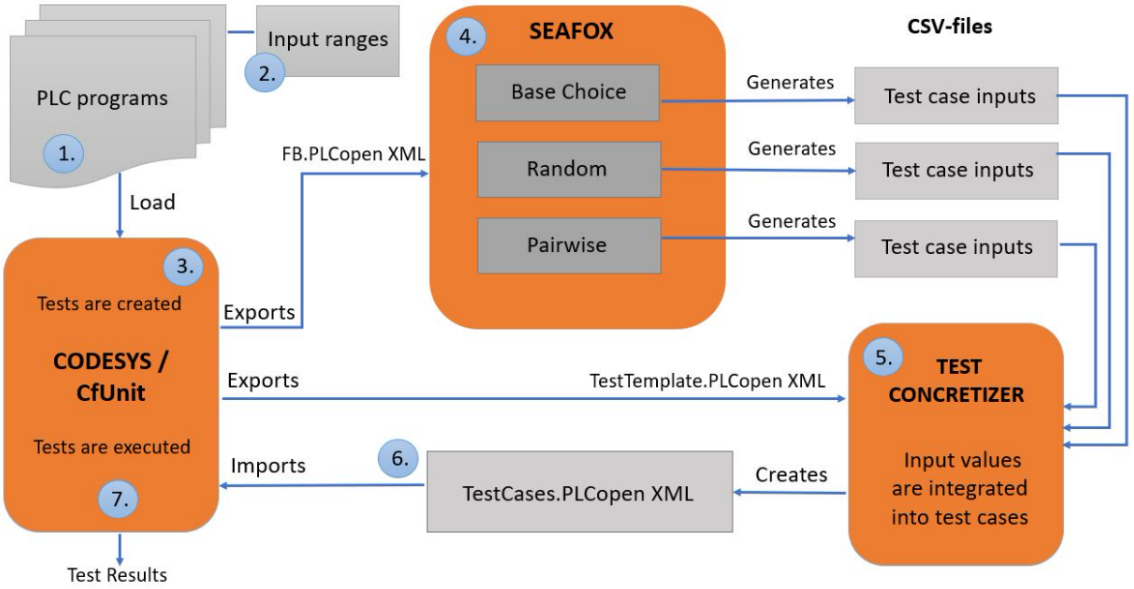


Figure 1.7. An overview of the integrated toolchain for combinatorial test modelling and test execution in CODESYS for IEC 61131-3 and its usage scenarios.

Test cases in CODESYS IDE were written following the instructions by CfUnit. CfUnit provides an assertion method that checks if the condition in the assertion is True. If it is False, an assertion error is created. One test case per test suite was created as a template that was later used with the script and the exported file from SEAFOX to generate all the test cases for the test suite. All input variables in the test case are initialized with a dummy value for enabling the script to set the test case inputs (see Figure 1.8).

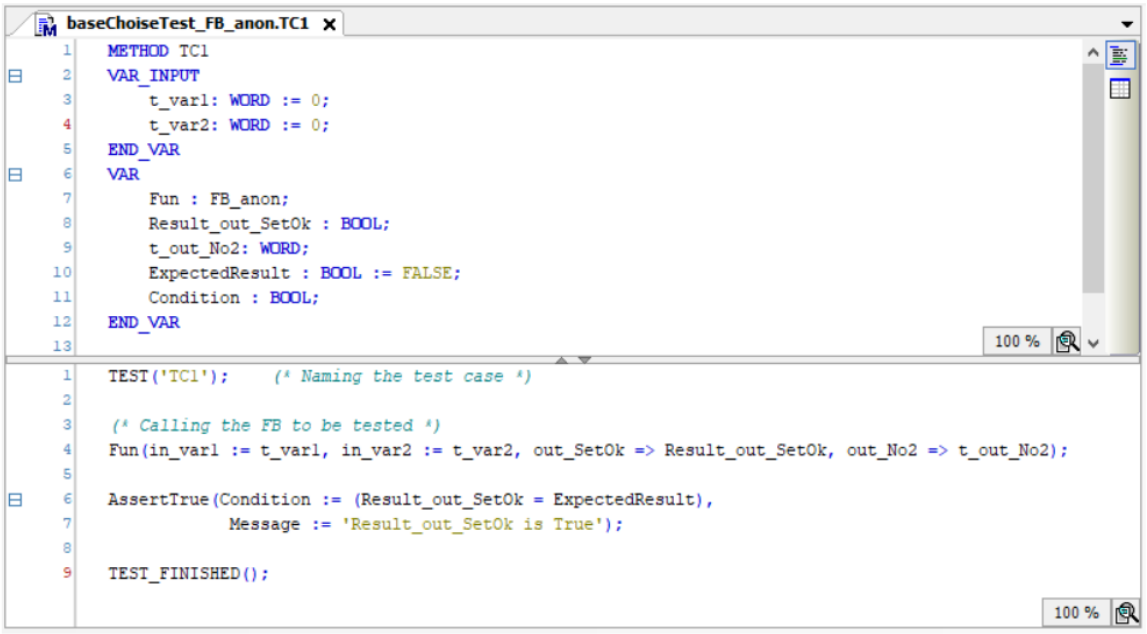


Figure 1.8. A test case in CODESYS IDE. The input variables are initialized with a dummy value that will later be replaced with a proper test case input through the execution of the script. An assertion that compares the actual and expected result is used to measure coverage.

Application to the case studies in the VeriDevOps project (2): SEAFOX CODESYS Edition is an open source tool[5] that supports PLCOpen XML and *.EXP files to be imported from CODESYS. The supported algorithms are Random, base-choice and pairwise. The source code of the test concretizer is available in a GitHub repository.

For the merging process to begin, the user first needs to specify the names of the files used in the script:

- The csv-file, holding all the input values
- The PLCopen XML file that will be parsed and used as a template for the new test case creation.
- The output PLCopen XML file will be created holding all the newly generated test cases.

[5] https://github.com/acn18/DVA331-SEAFOX-02

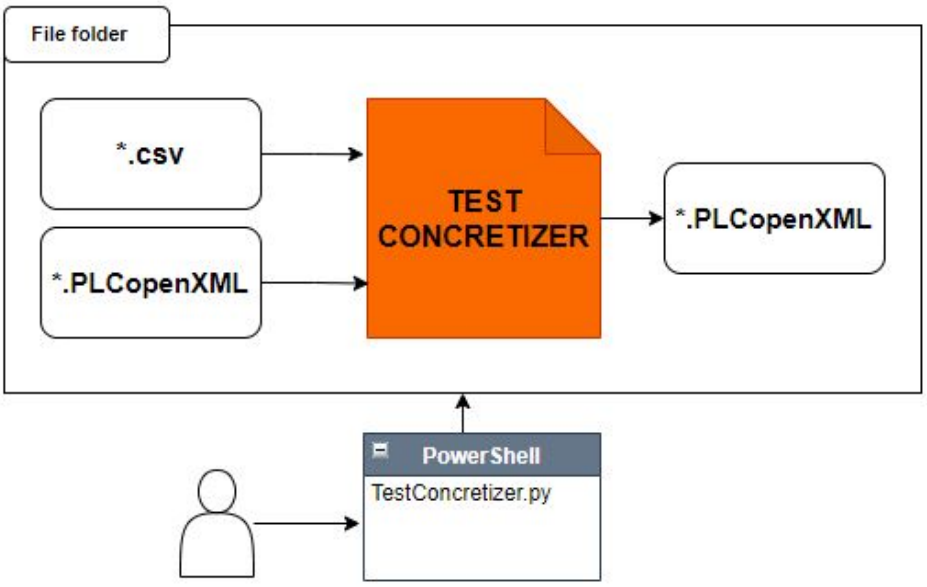


Figure 1.9. Test Integration of SEAFOX with CfUnit for CODESYS.

In order to execute the script (see Figure 1.9), all the previously mentioned input files and the script itself need to be located in the same file folder on the user's computer. Then the user needs to open the Command Prompt and specify the path to this folder and run the script or simply open the Windows PowerShell from the folder and run the python script using the python command. If the execution of the script went as expected, the number of created test cases would be printed out on the screen as well as the message of the successfully created .xml file.

Application to the case studies in the VeriDevOps project (3): We will further develop SEAFOX, by developing an automated methodology for requirements discretizing the input space based on functional requirements and devising attack models based on security requirements. In the end, the goal is to identify one or more combinations of input parameter values that would definitely trigger a vulnerability in the control software.

4.3. µUTA - a tool for model-based mutation testing

As discussed in previous sections, model-based testing (MBT) is an approach used to check whether the implementation of a system conforms to its specification[6]. The specification is typically represented as a behavior model and it is used for test design based on selected coverage criteria and test generation algorithms. When the tests pass, one considers that the implementation under test (IUT)

[6] MBT is one of the techniques that can be instantiated from the taxonomy for active testing tools presented in Section 2 and 3.

satisfies its specifications. If the tests fail, the error can originate either from the implementation (e.g, programming mistake, misunderstanding of requirements) or from the specification (e.g., incomplete or inconsistent specification, unclear requirements) and it has to be fixed accordingly.

However, in certain situations, one would like to evaluate how the IUT behaves when it is subject to unexpected inputs, either unspecified or malicious, and a traditional approach for that was fuzz testing [17]. Fuzz testing focuses mainly on generating invalid, unexpected or random inputs to computer programs with the goal of testing their robustness and security [18]. In fuzz testing, the main focus is on the input data to the program. However, testing how the program behaves when subject to correct data inputs but in the wrong order is also important for detecting inconsistencies in either the specification or in the implementation.

Specification mutation analysis is an approach used to design tests to evaluate the correctness and consistency of the specification or of the program [19]. When the mutation analysis is applied to the specification, a set of mutation operators create slightly altered versions of the specification, called *mutants*. Tests will be generated from the mutated specification in order to assess whether the IUT is accepting the tests. Accepting tests generated from a mutant specification successfully may indicate an unexpected behavior of the implementation.

Based on the verdict of the tests generated from a mutant specification, we can consider the following categories of mutants [20]:

- *killed*: the tests generated from a mutant specification fail against the implementation, under the precondition that the tests generated from the original specification have passed. These tests will basically confirm that the IUT does not conform to an incorrect specification;
- *alive*: all tests generated from the mutant specification pass against the IUT. Alive mutants can be divided into two groups:
    - *equivalent*: The mutant specification manifests the same behavior as the original model, even if they are syntactically different.
    - *non-equivalent*: The mutant specification does not have the same behavior as the original model; however, all tests generated from the mutant pass against the IUT. This indicates that the implementation is too permissive and is not able to detect invalid input behavior.

Deploying specification mutation analysis in the context of model-based testing has been previously referred to as *model-based mutation testing (*MBMT) [20].

µUTA is an in-house proof-of-concept tool developed at Åbo Akademi which automates the process of model-based mutation testing. The tool takes input system specifications represented using UPPAAL timed automata [21] like the one in Figure 1.10.

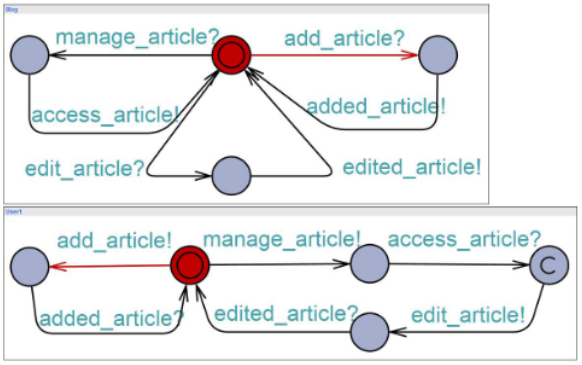


Figure 1.10 Excerpt UPPAAL TA model of user (environment) interaction with a Blog (SUT)

UPPAAL models are specified as a closed network of timed automata with shared variables, clocks and actions. Each automaton consists of a set of *locations* and *edges* with guards and clock updates. Synchronizations among channels are implemented via channels denoted with "!" for emitting and with "?" for receiving.

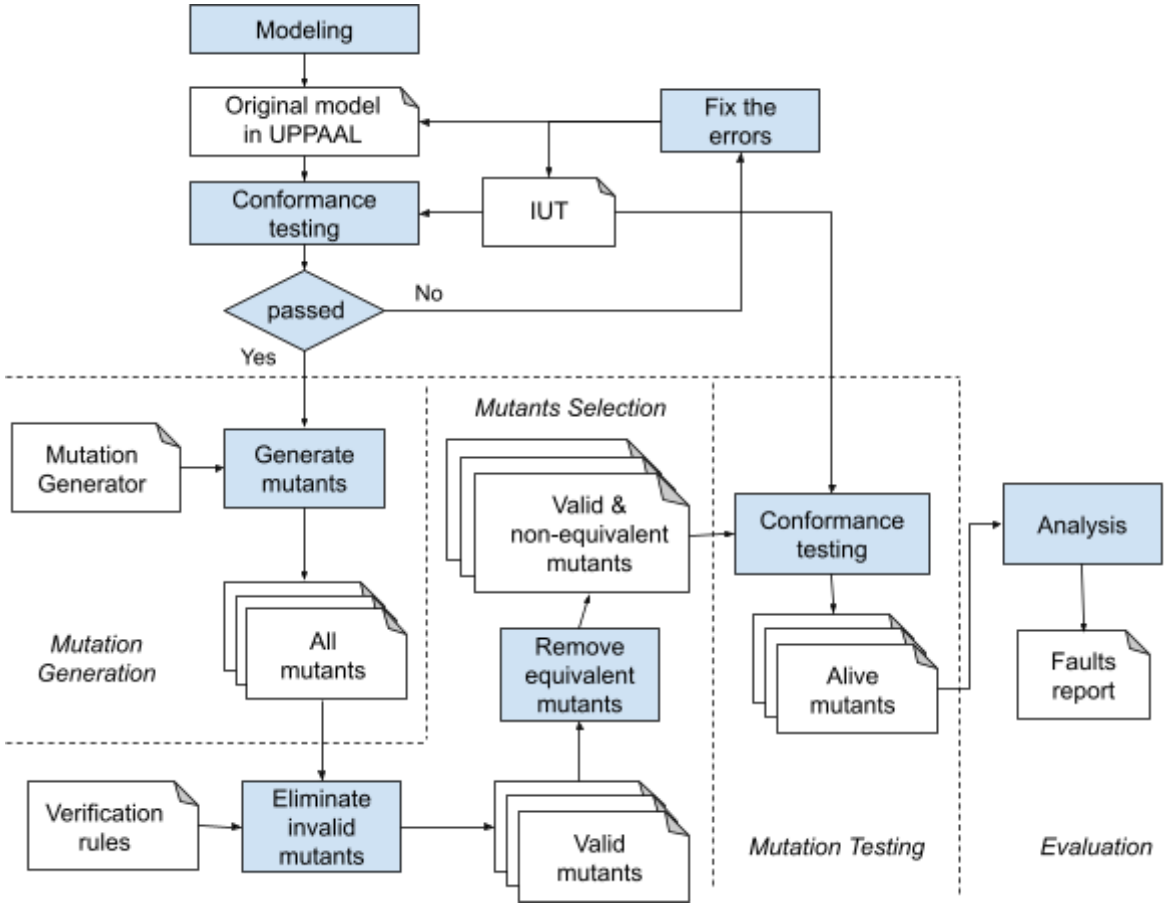
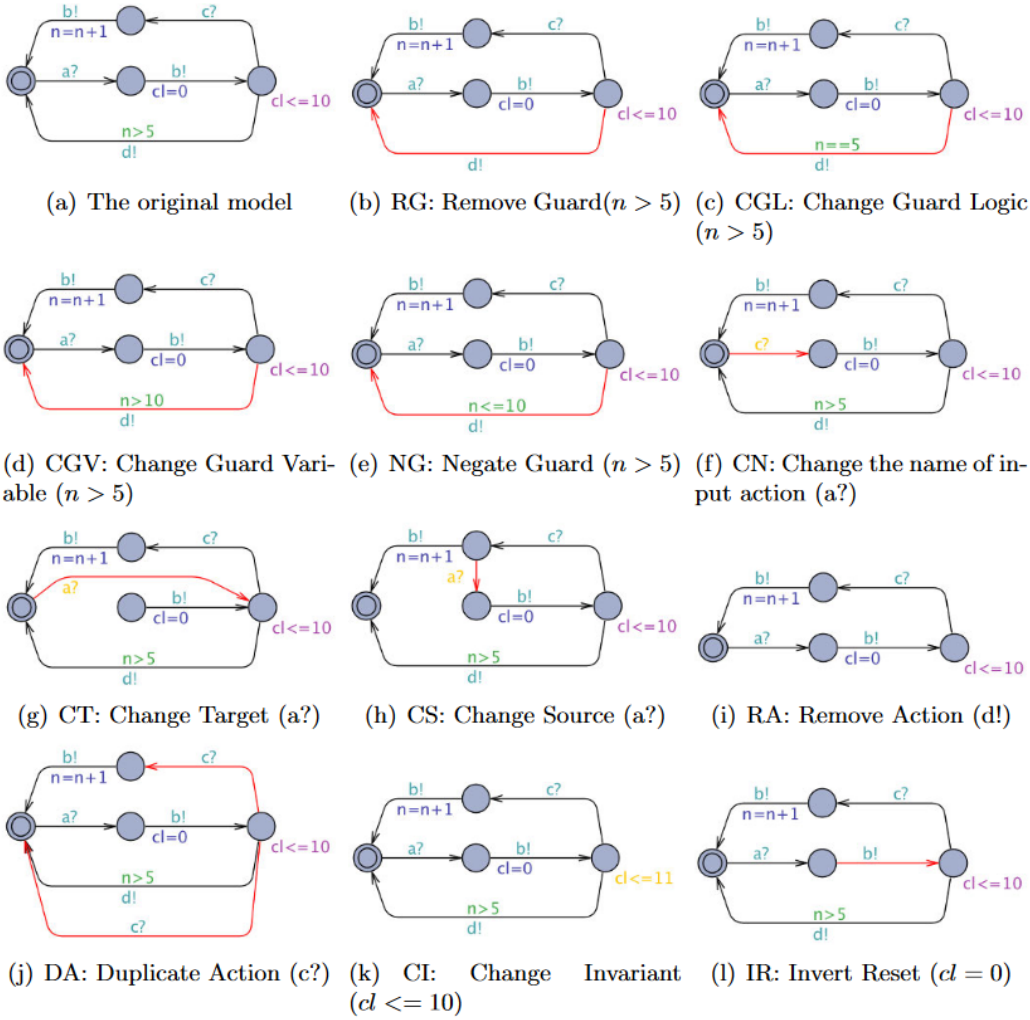


Figure 1.11 Model-based mutation process with the µUTA tool

The general approach implemented by the tool is described in Figure 1.11. A model of the system is created using UPPAAL Timed Automata (TA) and used for online conformance testing using the UPPAAL TRON test generator [22] against the implementation under test. If any errors are detected, they are fixed either in the implementation or in the model, and the process is repeated until all tests pass. This ensures that the IUT conforms to its specification, in this case, the *original UPPAAL model*.

The resulting UPPAAL model is used as input to the µUTA tool, which performs the following steps:

- **generates mutants** by applying a number of predefined *mutation operators* on the given UPPAAL model. These mutation operators have been discussed and evaluated in [23], [24]. Examples of mutation operators for UPPAAL TA are shown in Figure 1.12. For each application of a mutant to a suitable model fragment. The resulting mutants are also UPPAAL TA models that may exhibit different observable behavior than the original model.



(a) The original model (b) RG: Remove Guard($n > 5$) (c) CGL: Change Guard Logic ($n > 5$)

(d) CGV: Change Guard Variable ($n > 5$) (e) NG: Negate Guard ($n > 5$) (f) CN: Change the name of input action (a?)

(g) CT: Change Target (a?) (h) CS: Change Source (a?) (i) RA: Remove Action (d!)

(j) DA: Duplicate Action (c?) (k) CI: Change Invariant ($cl <= 10$) (l) IR: Invert Reset ($cl = 0$)

Figure 1.12 Example of mutation operators implemented in µUTA tool [25]

- **eliminates invalid mutants** (including those that are syntactically incorrect) by applying a set of reachability rules for the mutated model fragments.
- **eliminates equivalent mutants** by checking if a mutant is weakly bisimilar (wrt externally observable behavior) with the original model.
- **generates and executes online tests from non-equivalent mutants** against the IUT using UPPAAL TRON. The same test adapter as in the preliminary conformance testing phase is used to intermediate between the abstract and the physical communication (see Figure 1.13). Each test session is executed for a predefined period of time or until an error is discovered. If no error was discovered in the specified period, the mutant is considered as alive.

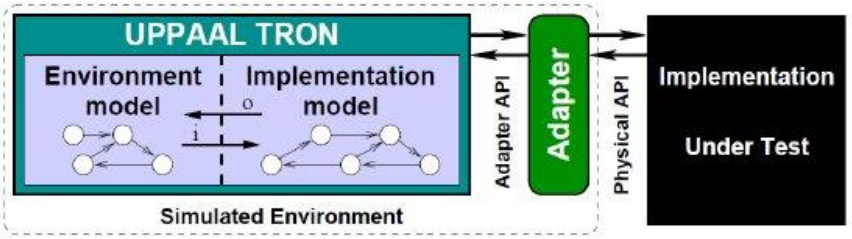


Figure 1.13 Test setup for the UPPAAL TRON framework [22]

- **returns the list of alive mutants** and the corresponding test traces.

At the moment, the **analysis of the alive mutants** is performed manually by the test engineer.

The approach has been applied in the past for testing the robustness [23] and security [26] of web services. The results showed that the model-based mutation testing approach was able to detect additional errors that were not detected by the traditional online model-based conformance testing approach.

Application to the case studies in the VeriDevOps project: We plan to apply the µUTA tool to both case studies for security testing of PLC components and, respectively, for testing RESTful Web APIs.

5. Conclusions

This initial deliverable reports on initial technologies for test generation used in VeriDevOps. The main focus in this deliverable will be on technologies for verifying that the implementation satisfies certain security properties specified in the design phase. In addition, it reports on the tools for generating and measuring the quality of the tests at the code level. The generic test generation process and the presented taxonomy helps to clarify the main characteristics of the automated test generation area and show the possible alternatives and directions of active tools for prevention developed in VeriDevOps towards software security. This information can classify test generation tools and help

testers or users understand which approaches fit their specific needs most closely. Automated test case generation has matured and large-scale deployments of this technology are underway in many industries. Given the myriad of approaches available, the tools and the taxonomy presented in this deliverable are of high value for both researchers and practitioners working in software security testing. Its application in research and practice will result in continuous validation and refinement of the taxonomy. In this initial version of this deliverable, we outline three tools for active prevention: CompleteTest (random testing and fuzzing based on model checking for detecting software vulnerabilities), SEAFOX (combinatorial security testing) and µUTA (model-based mutation testing for robustness and security testing).

References

[1] S. Anand *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8. pp. 1978–2001, 2013 [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.02.061

[2] A. Rashid, H. Chivers, G. Danezis, E. Lupu, and A. Martin, *CyBOK: Cyber Security Body of Knowledge*. 2019 [Online]. Available: https://books.google.com/books/about/CyBOK.html?hl=&id=AUL6zQEACAAJ

[3] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, no. 4. pp. 1959–1981, 2018 [Online]. Available: http://dx.doi.org/10.1007/s10664-017-9570-9

[4] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, "Model-based security testing: a taxonomy and systematic classification," *Software Testing, Verification and Reliability*, vol. 26, no. 2. pp. 119–148, 2016 [Online]. Available: http://dx.doi.org/10.1002/stvr.1580

[5] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5. pp. 297–312, 2012 [Online]. Available: http://dx.doi.org/10.1002/stvr.456

[6] R.-H. Pfeiffer, "What constitutes Software?," *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020 [Online]. Available: http://dx.doi.org/10.1145/3379597.3387442

[7] T. H. Tse, "An Examination of Requirements Specification Languages," *The Computer Journal*, vol. 34, no. 2. pp. 143–152, 1991 [Online]. Available: http://dx.doi.org/10.1093/comjnl/34.2.143

[8] G. Fraser and J. M. Rojas, "Software Testing," *Handbook of Software Engineering*. pp. 123–192, 2019 [Online]. Available: http://dx.doi.org/10.1007/978-3-030-00262-6_4

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5. pp. 507–525, 2015 [Online]. Available: http://dx.doi.org/10.1109/tse.2014.2372785

[10] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortes, "A Survey on Metamorphic Testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9. pp. 805–824, 2016 [Online]. Available: http://dx.doi.org/10.1109/tse.2016.2532875

[11] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson, "Automated test generation using model checking: an industrial evaluation," *International Journal on Software*

Tools for Technology Transfer, vol. 18, no. 3. pp. 335–353, 2016 [Online]. Available: http://dx.doi.org/10.1007/s10009-014-0355-9

[12] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3. pp. 215–261, 2009 [Online]. Available: http://dx.doi.org/10.1002/stvr.402

[13] E. Enoiu, D. Sundmark, A. Causevic, and P. Pettersson, "A Comparative Study of Manual and Automated Testing for Industrial Control Software," *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2017 [Online]. Available: http://dx.doi.org/10.1109/icst.2017.44

[14] E. P. Enoiu and C. Seceleanu, "Model Testing of Complex Embedded Systems Using EAST-ADL and Energy-Aware Mutations," *Designs*, vol. 4, no. 1. p. 5, 2020 [Online]. Available: http://dx.doi.org/10.3390/designs4010005

[15] M. N. Zafar, W. Afzal, E. Enoiu, A. Stratis, A. Arrieta, and G. Sagardui, "Model-Based Testing in Practice: An Industrial Case Study using GraphWalker," *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*. 2021 [Online]. Available: http://dx.doi.org/10.1145/3452383.3452388

[16] E. Enoiu and R. Feldt, "Towards Human-Like Automated Test Generation: Perspectives from Cognition and Problem Solving," *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 2021 [Online]. Available: http://dx.doi.org/10.1109/chase52884.2021.00026

[17] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12. pp. 32–44, 1990 [Online]. Available: http://dx.doi.org/10.1145/96267.96279

[18] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018 [Online]. Available: http://dx.doi.org/10.1145/3243734.3243804

[19] T. A. Budd and A. S. Gopal, "Program testing by specification mutation," *Computer Languages*, vol. 10, no. 1. pp. 63–73, 1985 [Online]. Available: http://dx.doi.org/10.1016/0096-0551(85)90011-6

[20] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. Eric Wong, "Model-based mutation testing—Approach and case studies," *Science of Computer Programming*, vol. 120. pp. 25–48, 2016 [Online]. Available: http://dx.doi.org/10.1016/j.scico.2016.01.003

[21] A. Hessel and Others, "Testing Real-Time Systems Using UPPAAL," in *Formal Methods and Testing*, Springer-Verlag, 2008, pp. 77–117 [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78917-8_3

[22] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing Real-time Embedded Software Using UPPAAL-TRON: An Industrial Case Study," in *Proceedings of the 5th ACM International Conference on Embedded Software*, Jersey City, NJ, USA, 2005, pp. 299–306, doi: 10.1145/1086228.1086283 [Online]. Available: http://doi.acm.org/10.1145/1086228.1086283

[23] F. Siavashi, J. Iqbal, D. Truscan, and J. Vain, "Testing Web Services with Model-Based Mutation," in *Software Technologies*, vol. 743, E. Cabello, J. Cardoso, A. Ludwig, L. A. Maciaszek, and M. van Sinderen, Eds. Springer, 2017, pp. 45–67.

[24] F. Siavashi, D. Truscan, and J. Vain, "On Mutating UPPAAL Timed Automata to Assess Robustness of Web Services," in *ICSOFT-EA*, 2016, pp. 15–26, doi: 10.5220/0005970800150026 [Online]. Available: http://dx.doi.org/10.5220/0005970800150026

[25] F. Siavashi, "Model-based Verification and Testing of Web services: Functionality, Robustness and Vulnerability Analysis," Doctoral Dissertation, Åbo Akademi University, 2020 [Online]. Available: https://www.doria.fi/bitstream/handle/10024/176843/siavashi_faezeh.pdf?sequence=5&isAllowe

d=y

[26] F. Siavashi, D. Truscan, and J. Vain, “Vulnerability Assessment of Web Services with Model-Based Mutation Testing,” in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018, pp. 301–312, doi: 10.1109/QRS.2018.00043 [Online]. Available: http://dx.doi.org/10.1109/QRS.2018.00043