Testing Cyber-Physical Systems under Uncertainty: Systematic, Extensible, and Configurable Model-based and Search-based Testing Methodologies

D 3.1 - Report on Uncertainty Testing Framework V.1

Project Acronym	U-TEST	**Grant Agreement Number**		H2020-ICT-2014-1. 645463	
Document Version	1.0	**Date**	2016-10-26	**Deliverable No.**	3.1
Contact Person	Phu Hong Nguyen	**Organization**		Simula Research Laboratory	
Phone	+47 900 255 81	**E-Mail**		phu@simula.no	

Document Version History

Version No.	Date	Change	Author(s)
0.1	2016-04-14	Initial document outline	Phu Hong Nguyen
0.2	2016-07-06	Updated initial document outline	Phu Hong Nguyen
0.3	2016-08-12	Sections 1-3	Phu Hong Nguyen
0.4	2016-08-30	Complete outline of D3.1	Phu Hong Nguyen
0.5	2016-10-04	Integrating contribution from SRL	Man Zhang, Shaukat Ali, Tao Yue, Phu Hong Nguyen
0.6	2016-10-04	Integrating contribution from TUW	Daniel Moldovan, Hong-Linh Truong, Phu Hong Nguyen
0.7	2016-10-06	Integrating contribution from FF	Martin Schneider, Phu Hong Nguyen
0.8	2016-10-10	Complete version for reviews	Phu Hong Nguyen
0.9	2016-10-24	Some minor revisions	All
1.0	2016-10-26	Complete version for Project Coordinator	All

Contributors

Name	Partner	Part Affected	Date
Man Zhang	SRL	Sections 4.4, 5.3, TR4, TR5	2016-10-04
Shaukat Ali	SRL	Sections 4.4, 5.3, TR4, TR5	2016-10-04
Tao Yue	SRL	Sections 4.4, 5.3, TR4, TR5	2016-10-04
Phu Hong Nguyen	SRL	All	
Martin Schneider	FF	Sections 4.2, 5.1	2016-10-06
Daniel Moldovan	TUW	Sections 4.3, 5.2, TR1, TR2, TR3	2016-10-04
Hong-Linh Truong	TUW	Sections 4.3, 5.2, TR1, TR2, TR3	2016-10-04

Reviewers

Name	Partner	Part Affected	Date
Robert Magnusson	NMT	All	2016-10-13
Karmele Intxausti	IKL	All	2016-10-14
Fabien Peureux	EGM	All	2016-10-19
Malin Hedman	NMT	All	2016-10-19

Table of Contents

Executive Summary

This deliverable presents the Uncertainty Testing Framework (UTF) V.1 with various test strategies developed for uncertainty testing of Cyber-Physical Systems (CPS). The test strategies that we have developed and integrated into the UTF can support for uncertainty testing at the three levels (application, infrastructure, and integration) of CPS. More specifically, the UTF takes the test ready models specified with the Uncertainty Modeling Framework (UMF) as input, and (automatically) produces abstract test cases and executable test cases as output. Furthermore, this deliverable also reports on our search-based approaches for searching unknown uncertainty behaviors based on known uncertainty behaviors. The results of test case generation from known and unknown uncertainty models for the pilot cases are presented and discussed. This deliverable shows that we have successfully achieved Milestone 3 with the UTF V.1 for uncertainty testing at the three levels of CPS. Our UTF V.1 has provided a concrete foundation for achieving Milestone 4, in which we continue to improve the testing framework and apply it more extensively for the pilot cases.

Keywords: Cyber-Physical Systems, Model-Based Testing, Uncertainty Testing, Testing Framework

Abbreviations

AMQP	Advanced Message Queuing Protocol
APML	All Paths with Maximum Length
ASP	All Simple Path
CPS	Cyber-Physical Systems
Dx	Deliverable number x
EGM	Easy Global Market
FPX	Future Position X
FF	Fraunhofer FOKUS
IEC	International Electro-technical Commission
IEEE	Institute of Electrical and Electronics Engineers
IKL	Ikerlan
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MBT	Model-Based Testing
MQTT	Message Queuing Telemetry Transport (protocol)
NMT	Nordic Medtest
OCL	Object Constraint Language
OMG	Object Management Group
SBSE	Search-Based Software Engineering
SRL	Simula Research Laboratory
SUT	System Under Test
TR	Technical Report
TUW	Technische Universität Wien
U-Taxonomy	Uncertainty Taxonomy
ULMA	ULMA Handling Systems
UMF	Uncertainty Modeling Framework
UTF	Uncertainty Testing Framework
UTP	UML Testing Profile
WP	Work Package

1 Introduction

This report is the first official deliverable of the work package "Developing Uncertainty Testing Framework" (WP3). The Uncertainty Testing Framework (UTF) is one of the key outcomes of U-Test. The main objectives of this deliverable are given in Section 1.1. We show in Section 1.2 the relationship of this deliverable to the other deliverables of U-Test project. The structure of this deliverable is presented in Section 1.3.

1.1 Objectives of the Deliverable

The goal of this deliverable is to present the UTF that we have developed. Our UTF supports for testing uncertainties and uncertain behaviors of Cyber-Physical Systems (CPS) at three levels: application, infrastructure, and integration. We developed and integrated different testing strategies in the UTF. These testing strategies take the test ready models specified in the UMF as inputs for uncertainty testing.

As reported in the previous deliverables, we developed the Uncertainty Taxonomy (U-Taxonomy) [2] and Uncertainty Modeling Framework (UMF) [3]. We used U-Taxonomy and UMF for specifying and modeling different uncertainties of CPS, at three levels, i.e., application, infrastructure, and integration. In this deliverable, we show how our UTF (V.1) bases on the U-Taxonomy and the UMF. We developed UTF on the state of the art of Model-Based Testing (MBT) techniques, and especially customized for uncertainty testing at the three levels (application, infrastructure, and integration) of CPS. Moreover, this deliverable reports the definition of search-based approaches for searching unknown uncertainty behaviors. The searching is based on known uncertainty behaviors at the three levels of CPS. Finally, the initial results of test case generation from known and unknown uncertainty models for the pilot cases are presented and discussed.

This report provides the first version of our UTF. We are still making possible improvement on the UTF. The further iteration and refinements of UTF will be available in the next U-Test reports.

1.2 Relationship to other U-TEST Deliverables

This deliverable presents the results of U-Test's Work Package 3 that has relationships with other U-Test deliverables and work packages. In particular, the specification of the uncertainty requirements from two U-Test use cases (D1.1) [1], the U-Taxonomy (D1.2) [2], and the UMF (D2.1) [3] are the prerequisites of the UTF. In addition to that, UTF is also built on the state of the art of MBT techniques and standards, e.g., UML Testing Profile (UTP), ISO/IEC/IEEE 29119 Software Testing Standards. With the test ready models specified with the UMF as inputs, UTF has implemented different test strategies and MBT techniques for uncertainty testing at the three levels (application, infrastructure, and integration) of CPS. In other words, the output of the UMF is the main input of UTF. We modeled the test-ready models of the use cases by using UMF. These test-ready models are used in the UTF for test case generation.

The results of UTF will be used for U-Test's next active work packages such as Tool(s) Demonstrator (D4.2), Report on test case executions (D5.1), Dissemination (D6.3), and Exploitation (D7.2).

Figure 1 shows again the overall workflow of the methodology in our U-Test project and more specifically where the UTF is located in the workflow of U-Test project.

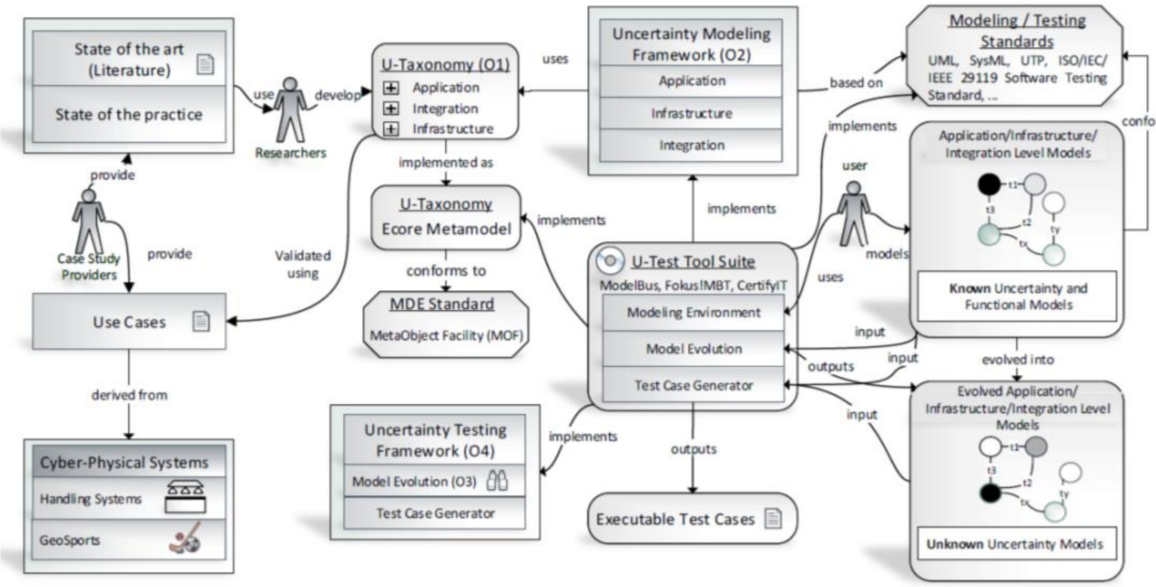


Figure 1 – U-Test workflow

Input:

- All previous Deliverables

Extension

- N/A

Consumers of D3.1 (that are currently active)

- D4.2 (EGM, FF): Tool(s) Demonstrator
- D5.1 (FPX and ULMA): Report on test case executions
- D7.2 (For Exploitation): Value Opportunities
- D6.3: Dissemination

Consumers of D3.1 (not active yet)

- D5.2 Empirical Evaluation of the Cost-Effectiveness of Test Strategies
- D5.3 Validation With or Without U-Test

1.3 Structure of the Deliverable

The deliverable consists of this main document and its appendix (as technical reports). The main content of this document gives the condensed presentation of the UTF. More details of some specific key results of the UTF can be found in the technical reports. The technical reports provide more detailed technical aspects of the UTF such as U-Evolve for evolving test ready models to discover unknown uncertainty [30].

Figure 2 depicts the relationships and the structure of this deliverable. Section 1 gives an overview of the deliverable.

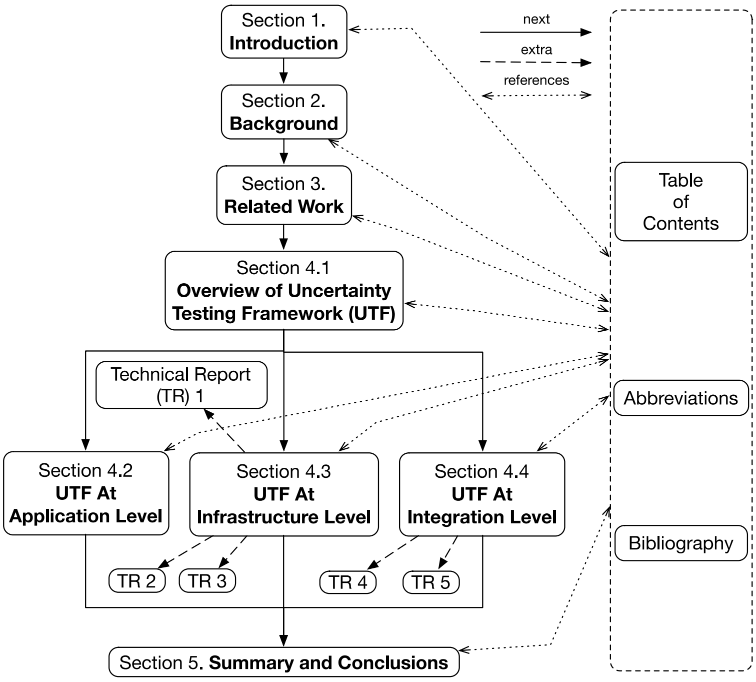

Section 1. Introduction
Section 2. Background
Section 3. Related Work
Section 4.1 Overview of Uncertainty Testing Framework (UTF)
Technical Report (TR) 1
Section 4.2 UTF At Application Level
Section 4.3 UTF At Infrastructure Level
Section 4.4 UTF At Integration Level
TR 2
TR 3
TR 4
TR 5
Section 5. Summary and Conclusions
next
extra
references
Table of Contents
Abbreviations
Bibliography

Figure 2 – The structure of this deliverable

The remainder of this deliverable is organized as follows. Section 2 provides briefly some background concepts that are used in this deliverable, e.g., MBT, UTP. In Section 3, we discuss some related MBT approaches to better position the contribution of our UTF in particular and U-Test work in general. An overview of the UTF is given in Section 4.1. How our UTF, which supports uncertainty testing at the application level, infrastructure level, and integration level of CPS, are provided in Sections 4.2, 1.1, and 4.4 correspondingly. Aiming at the comprehensiveness of this document, for presenting technical details on some specific topics, we organized them into technical reports (TRs). TR1, TR2, and TR3, which provide more technical details for Section 1.1, are included in the Appendix. TR4 and TR5, which provide more technical details for Section 4.4, are in forms of two separate PDF files attached with this document. We summarize the whole deliverable and give our conclusions in Section 5.

2 Background

In this section, we briefly provide some background concepts and techniques that are used in the deliverable. To be more specific, our UTF has been (partly) developed on these key concepts and techniques. First, Model-Based Testing (MBT) is shortly introduced in Section 2.1. In Section 2.2, we present the first standardized language for supporting MBT, which is called the UML Testing Profile (UTP). UTP is leveraged in our UTF. Section 2.3 gives a short introduction to Search-based Software Engineering methodology that is employed in UTF for discovering unknown uncertain behaviors.

2.1 Model-Based Testing

Testing is currently the most widely used technique in industry to gain some confidence in the quality of a system, normally in a cost-efficient way. MBT is a variant of testing that mainly encompasses the insight of using models, e.g. UML models that are extended for the purpose of testing of real-time embedded systems such as the OMG UML Testing Profile [7] for testing, and OMG MARTE [11] for Modeling and Analysis of Real-Time and Embedded Systems. More specifically, MBT relies on the behavior models of a system under test (SUT) and/or its environment to (automatically) derive test cases for the system [28]. Therefore, MBT allows the tests derivation process to be structured, reproducible, programmable, and documented. The results of the 2014 MBT User Survey suggest that MBT has positive effects on efficiency and effectiveness [27]. The study presented in [25] shows that MBT approach is more systematic, and more effective in detecting issues compared to the manual testing approach in certain areas. Under U-Test project, the UTF is developed based on the state of the art of MBT research.

2.2 UML Testing Profile (UTP)

The UML Testing Profile (UTP)[1] was the first industry-driven, standardized language to support MBT. Before that, proprietary attempts have been undertaken to use UML for doing MBT. The UTP [7] augments UML with test-specific concepts that are lacking in UML itself for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts commonly used in and required for various testing approaches, especially model-based testing. MBT specifications expressed with the UTP are independent of methodology, domain, tool, or system type. UTP was efficiently applied to foster early testing and to establish test automation by generation of executable test scripts. UTP has been applied to various industrial and research case studies to increase automation in test execution and test design [21] in various domains such as telecommunications, enterprise services choreographies, and e-Health [6, 7, 26]. Under U-Test project, the state of the art development of UTP is leveraged into the UMF as well as UTF.

2.3 Search-Based Software Engineering

Search-based Software Engineering (SBSE) attempts to solve a variety of software engineering problems by reformulating them as search problems [12]. Search algorithms are a set of generic algorithms. These generic algorithms are used to find optimal or near optimal solutions to problems that have large complex search spaces. SBSE in recent years has shown very promising results for various problems including requirements, architecture, design, and testing [13]. We employed SBSE in our context to evolve functional and known uncertainty models towards unknown uncertainty models capturing unknown uncertain behaviors.

3 Related Work

Most recent advances on MBT research area have been summarized in [27], [28]. There are some studies, e.g., [16, 17], [4], [10], and [24] that are more or less close to our work on the UTF in particular, and U-Test in general. However, none of them has dealt with the uncertainty of CPS.

Uppaal /Uppaal TRON presented in [17], [16], and [14] are modeling and testing languages, and tools that allow testing real-time behavior of embedded systems using timed automata that is essential for CPS. These were already employed for conformance testing of CPS. In [16], the authors present a tool for online testing of real-time systems called T-UPPAAL. The experiences in applying this tool and technique on an industrial case study are reported in [17]. Uppaal approach has been extended in [14] to support both offline and online testing of real-time systems. In other words, test cases can be generated and executed online, or they can be generated offline and executed later. However, Uppaal approach has not (directly) addressed the uncertainty of CPS.

Risk-based testing [4], [10] leverages risk assessments to support for all phases of the test process in optimizing testing efforts and limiting risks for the SUT. A risk-based testing approach and its application in a large project show how this approach can lead to more efficient testing with improved quality by focusing more efforts on critical functions [4]. Different risk-based testing approaches have been analyzed in [10] based on a taxonomy of risk-based testing. The taxonomy provides a framework to categorize, assess, and compare risk-based testing approaches. Risk-based

[1] http://www.omg.org/spec/UTP/

testing approaches are relevant for uncertainty testing because risk is an uncertainty state from where the outcome has undesired effect, which may result in unfortunate and negative impact on various concerns such as cost.

Fuzzing or fuzz testing [24], [15] is a testing methodology that stresses the interface of the SUT with invalid input data. If the implementation of the SUT processes invalid input data instead of rejecting them, these weaknesses might lead to intentionally or unintentionally crash the SUT or to modify its behavior in an unintended way. Therefore, invalid input data could be one of the main sources of the uncertainty of CPS. In [24], the authors present a new approach in fuzzing, i.e., behavioral fuzzing. This approach allows generating test cases by employing model-based behavioral fuzzing. The authors of [15] propose to integrate fuzzing into conformance testing and show how the robustness testing procedure for telecommunication products can be improved in that way.

4 Uncertainty Testing Framework

Section 4.1 gives an overview of the UTF. Next, Section 4.2 presents the details of UTF for supporting uncertainty testing at the application level of CPS. Similarly, Sections 1.1 and 4.4 present the details of UTF for supporting uncertainty testing at the infrastructure and integration levels of CPS correspondingly.

4.1 Overview of Uncertainty Testing Framework

Figure 3 shows a high-level overview of the UTF with its input and output. The main input of UTF is the test-ready models that we have created by using the UMF.

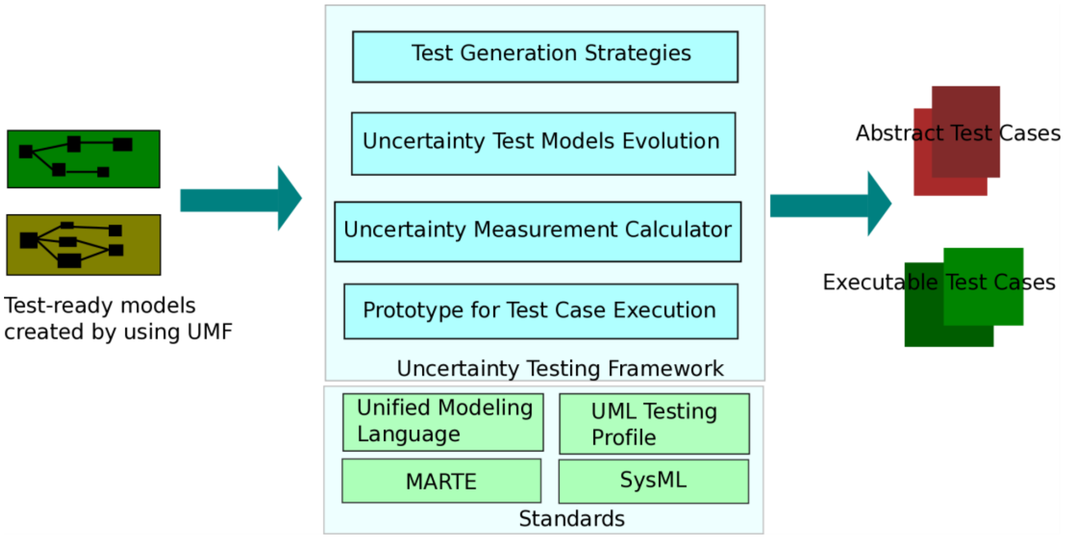

Test-ready models
created by using UMF
Test Generation Strategies
Uncertainty Test Models Evolution
Uncertainty Measurement Calculator
Prototype for Test Case Execution
Uncertainty Testing Framework
Unified Modeling Language
UML Testing Profile
MARTE
SysML
Standards
Abstract Test Cases
Executable Test Cases

Figure 3. An overview of Uncertainty Testing Framework

Our UTF is composed of the model-based test generation strategies that take as input the test-ready models above. These test-ready models cover the use cases for generating test for known uncertainties at the application level, infrastructure level, and integration level of CPS. On the other hand, UTF also integrates the uncertainty model evolution strategies aiming at discovering unknown uncertainties. The uncertainty measurement process can drive the test generation strategies with the support from the Uncertainty Measurement Calculator. We have demonstrated this approach at the integration level presented in Section 4.4. Because executing the large number of generated abstract test cases is impractical, we proposed a search-based approach to minimize the number of generated executable test cases while maximizing the coverage of transition. Last but not least, we developed a prototype as a proof-of-concept, in which the generated test cases can be executed.

The details of UTF for supporting uncertainty testing of CPS at three different levels are presented in the following sections.

4.2 Uncertainty Testing Framework At Application Level

Uncertainties at the application level comprise all the stimuli from the environment of the application level of the SUT. The project's pilots provide examples and described them in uncertainty use cases in D1.1, characterized using the uncertainty taxonomy described in D1.2 and semi-formally described in UML models according to the UMF developed in WP2. The purpose of uncertainty testing is (i) discovering *known uncertain behaviors* resulting from uncertainties that may be known at design time, and (ii) discovering *unknown uncertain behaviors* that may occur in the presence of yet unknown uncertainties.

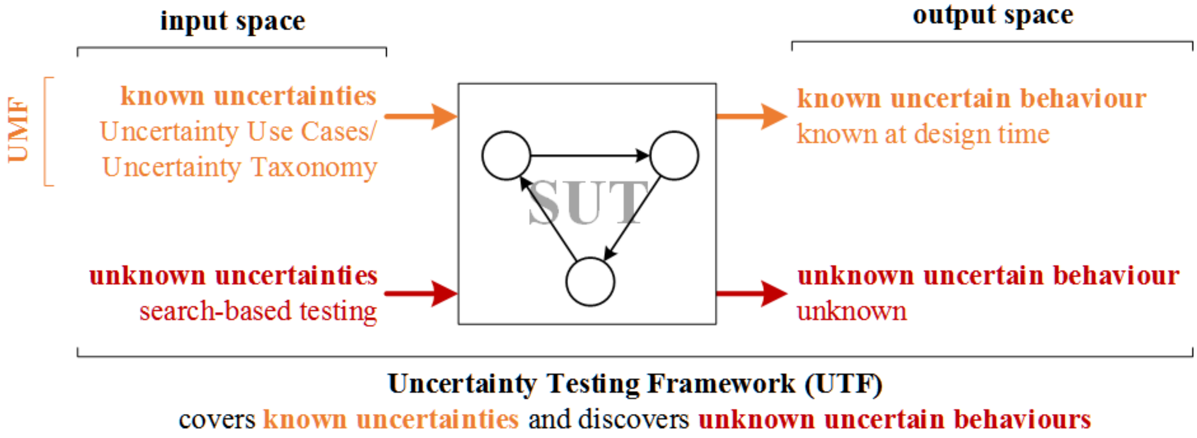


Figure 4: Relationship between Uncertainty Testing Framework (UTF) and other U-Test artifacts.

Uncertainties are representing stimuli to the system under test (SUT) that may cause uncertain behaviors located in the output space of the SUT. We distinguish between uncertainties known at design time ('known uncertainties') causing possibly uncertain behavior and unknown uncertainties that may cause unknown uncertain behavior. The Uncertainty Taxonomy allows describing their relevant characteristics whilst the UMF provides a means to describe them in terms of UML models. The UTF at the application level (UTF-AL) employs these models to search for known and unknown certainties that reveal uncertain behavior of the SUT. U-Test's pilots from the healthcare and the logistics domain provide Uncertainty Use Cases that serves the validation of the approach developed in U-Test.

Since we do not know these unknown uncertainties, we employ search-based techniques to walk efficiently through the input space. State machines are the individuals constituting the population of each generation. State machines are evolved through mutation and crossover. We employ a set of mutators specific to UML state machines, where each mutator becomes a mutation when applied to a specific element of a state machine. Furthermore, we employ crossover to recombine and swap mutations between state machines. Aiming at measuring whether we are approaching an uncertainty that may expose known or unknown uncertain behavior or if we have already discovered one, we exploit different outputs of the SUT as inputs to a fitness function for a genetic algorithm tailored to uncertainty testing. The genetic algorithm is part of the UTF for the Application Level (UTF-AL). The different parts of the genetic algorithm we employ to evolve state machines are explained in more detail in Section 4.2.3.

The uncertainty use cases modeled using UMF provide the starting point for UTF-AL. State machines describe all the valid interactions of the environment with the SUT from the viewpoint of the SUT. More details can be found in Deliverable 2.2 (D2.2). Transitions capture all this information in terms of guards, triggers, and effects specifying under which circumstances some effect behavior may occur. Additionally, the UMF enables to describe characteristics of known uncertainties. These modeled uncertainties form the basis for evolving state machines. Thus, we exploit the coupling effect [8], [23]. The goal is to gain state machines of which at least one path reveals known or unknown uncertain behavior.

4.2.1 Test Coverage Strategies

We use the following coverage strategies to measure and guide the progress of evolving state machines.

Strategy	**Measurand**[2]	**Description & Specification**
Transition Coverage	State Machine	Traditional transition coverage. $\frac{\#transitions_{covered}}{\#transitions_{all}}$
Uncertainty Coverage	UMF Model	How many modeled uncertainties are covered by evolved state machines and corresponding test cases of an evolved UMF model. $\frac{\#uncertainties_{covered}}{\#uncertainties_{modelled}}$
Mutation Coverage	State Machine	This is the ratio of mutations covered by all generated test cases of a single evolved state machine and all the mutations applied to the state machine. $\frac{\#mutations_{covered}}{\#mutations_{all}}$
Known Uncertainty Space Coverage	All generations of evolved state machines related to a single uncertainty	The Uncertainty Space in the context of a known uncertainty is spanned by all the possible inputs and all the mutations of the corresponding state machine. For this milestone, possible inputs are operations as trigger of the transitions and removing a trigger and an effect (the factor 2 stems from this). $\frac{\#mutations}{\#states \times (\#states - 1) \times \#operations \times 2}$

4.2.2 Test Data Generation

With respect to UTF-AL v1, we rely on test data generation facilities provided by Microsoft's Spec Explorer [19]. Additionally, we can include test data from earlier test runs. Such information can be provided together with the UMF model that serves as initial input for the UTF-AL.

4.2.3 Uncertainty Model Evolution at Application Level

In this section, we describe how we would evolve state machines with the help of modeled uncertainties aiming at two goals: finding instances of known uncertainties including their circumstances of them and the related uncertain behaviors the SUT reveals, and finding unknown uncertainties through evolving state machines with information provided by modeled uncertainties, use cases in terms of state machines and the SUT's interfaces.

[2] An entity quantified by a measurement, that is, the element we would like to measure through the test coverage strategy.

4.2.3.1 Overview

Figure 5 gives an overview of the process of state machine evolvement following the scheme of a genetic algorithm.

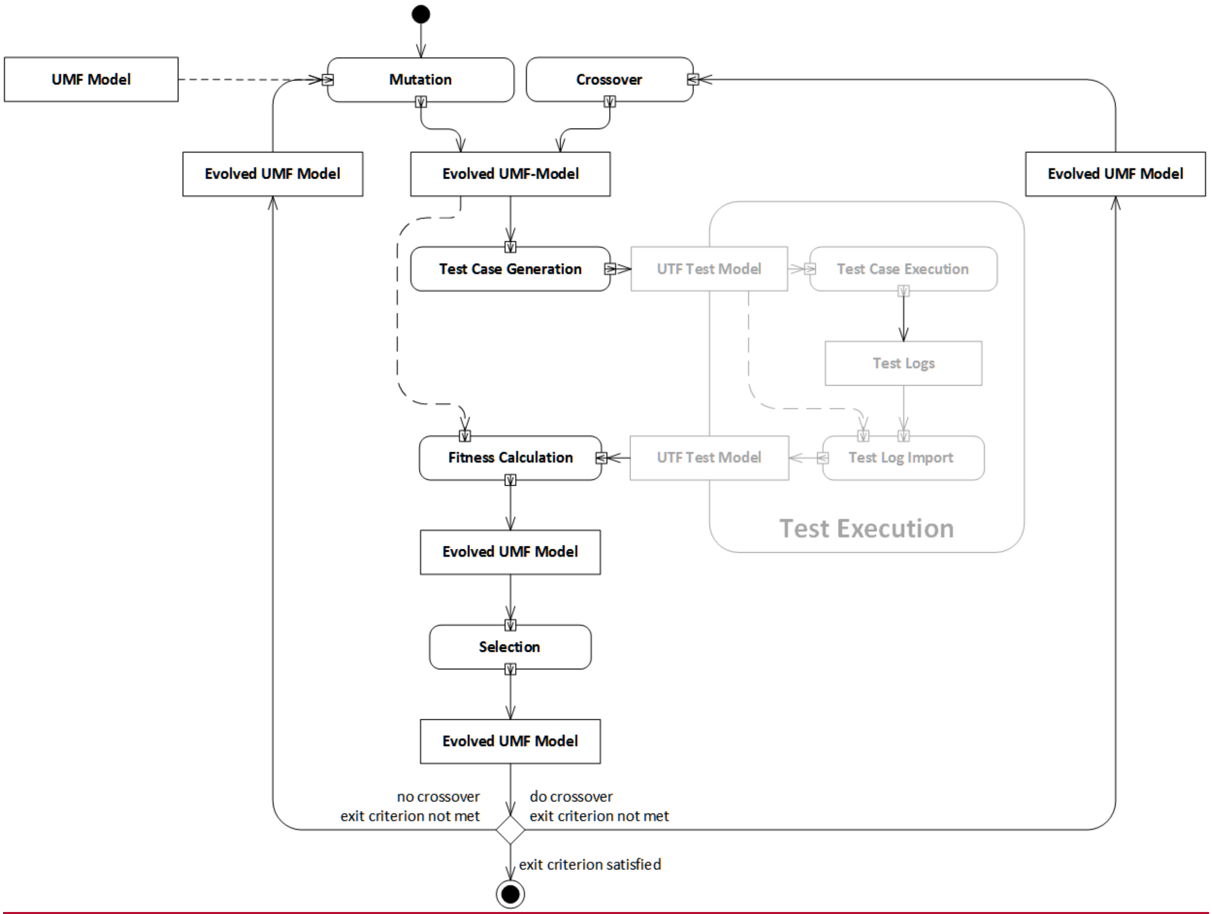


Figure 5: Overview of Uncertainty Testing for the Application Level

Step		Input *contents*	Actions *uses*	Output *added contents*
1.	**MUTATION**	**UMF-Model (WP2)** - *functional state machines* - *modeled uncertainties* - *use case-specific fitness factors* - *execution invariants*	**Mutation** - *uncertainties* - *state machines* See Section 4.2.3.3.	**Evolved UMF-Model** - *mutated state machines*
2.		**Evolved UMF-Model** - *mutated state machines*	**Test Case Generation** employs Microsoft Spec Explorer - *mutated state machines* - *execution invariants* See Section 4.2.4.	**UTF Test Model** - *test cases*
3.		**UTF Test Model** - *test cases*	**Test Case Execution (WP4/WP5)**	**UTF Test Model** - *mutated state*

Step		**Input** *contents*	**Actions** *uses*	**Output** *added contents*
			includes test code generation - *test cases* - *test adapter (EGM)*	*machines* - *fitness function* - *generated test cases* - *traces*
4.		**UTF Test Model** - *mutated state machines* - *fitness function* - *generated test cases* - *traces*	**Fitness Calculation** - *mutated state machines* - *traces* - *use case-specific fitness factors* - *fitness factors* See Section 4.2.3.5.	**Evolved UMF Model** - *mutated state machines* - *fitness values*
5.		**Evolved UMF Model** - *mutated state machines* - *fitness values*	**Selection** tournament selection - *mutated state machines* - *fitness values* See Section 4.2.3.2.	**Evolved UMF Model** - *selected, mutated state machines* - *fitness values*
6.	**CROSSOVER**	**Evolved UMF Model** - *selected, mutated state machines* - *fitness values*	**Crossover** - *selected, mutated state machines* - *fitness values* See Section 4.2.3.4.	**UMF Test Model** - *crossbred state machines*
7.		**Evolved UMF-Model** *mutated state machines*	**Test Case Generation** employs Microsoft Spec Explorer - *mutated state machines* - *execution invariants* See Section 4.2.4.	**UTF Test Model** - *test cases*
8.		**UTF Test Model** *test cases*	**Test Case Execution (WP4/WP5)** includes test code generation - *test cases* - *SUT-specific test adapter*	**UTF Test Model** - *mutated state machines* - *fitness function* - *generated test cases* - *traces*
9.		**UTF Test Model** - *mutated state machines* - *fitness function* - *generated test cases*	**Fitness Calculation** - *mutated state machines* - *traces* - *use case-specific fitness*	**Evolved UMF Model** - *mutated state machines* - *fitness values*

Step		Input *contents*	Actions *uses*	Output *added contents*
		traces	*factors* - *fitness factors* See Section 4.2.3.5.	
10.		**Evolved UMF Model** - *mutated state machines* - *fitness values*	**Selection** - *mutated state machines* - *fitness values* See Section 4.2.3.2.	**Evolved UMF Model** - *selected, mutated state machines* - *fitness values*

4.2.3.1.1 Individual

In genetic algorithms, an individual is a candidate solution. For the described approach, an individual could be the whole state machine, a path through the state machine, or a set of one or more mutations. In a first step, we consider whole state machines as individuals aiming at eventually having a state machine that is able to generate as many test cases that trigger as many uncertain behaviors as possible.

4.2.3.1.2 Fitness

The fitness of an individual is the quality of the candidate solution in terms of the optimization goal(s). Fitness calculation is essential to reveal successfully and efficiently uncertain behaviors. To do so, we distinguish the following types of fitness constituents:

- Use case-specific factors: e.g. difference between measured and actual positions for the GeoSports case study.
- System-specific factors: these are system-specific goals that are not necessarily related to a certain use case, e.g. response time or revealing unknown behavior.

Section 4.2.3.5 briefly describes how fitness values are calculated and mapped.

4.2.3.1.3 Building the first generation

Building the first generation of individuals aims at covering the known uncertainties modeled using the UMF developed in WP2. Further generations use information beyond this to find also unknown uncertainties.

This approach has two advantages: on one hand, we exploit the coupling effect [8], [23] stating that test cases that detects simple faults may help finding more complex faults . On the other hand, we use the search-based testing approach developed for finding unknown uncertainties in order to cover known uncertainties as well. Thus, it is only required to model use cases with intended interaction of the environment with the SUT and the corresponding known uncertainties. The complexity of creating models for uncertainty testing is also reduced. Additionally, a single approach for uncertainty testing is required and thus, reduces the complexity of the UTF-AL as well. We describe how we perform mutations, including those mutations based on modeled uncertainties, in a later section.

4.2.3.2 Selection

We use tournament selection for selecting state machines for the next generation and for crossover to select paths (see Section 4.2.3.4). This allows us to escape local optima.

4.2.3.3 *Mutation*

Mutation is performed on one hand using information from modeled uncertainties, and on the other hand independent from that by using information of the system's application level interfaces described by the UMF model. We apply mutations to transitions based on this information and select paths for further mutation based on their fitness values after the first generation has been created and corresponding test cases have been generated and executed. Several mutations of the same element are allowed. If such a combination does not respect execution invariants (see Section 4.2.4), no test cases will be generated, the fitness of the resulting state machine will be 0, and thus, eventually eliminated by the genetic algorithm.

For the first generations, we apply mutations solely based on modeled uncertainties. Thus, we are covering known uncertainties and the modeled use cases in the earlier generations of evolved state machines. As coverage of the known uncertainty space (see Section 4.2.1) increases, mutations based on system's interfaces instead of using information from modeled uncertainties are introduced that may represent unknown uncertainties and thus, reveal unknown uncertain behaviors.

Based on the literature [9] [18], we use the mutation operators as described in Table 1 and adapted them to UML state machines.

Mutation Operator	**Description**	**Constraints/Comments**
Add Transition	Adds a new transition by duplicating an existing one and setting a new source and target state.	
Remove Transition	Completely removes the transition.	Transitions having an initial state as source or a final node as target must not be removed.
Remove Transition (with State Merge)	Completely remove the transition. Merge the source and target state if the removed transition is the only one connecting them.	Transitions having an initial state as source or a final node as target must not be removed.
Reverse Transition	Swap source and target of the transition.	Transitions having an initial state as source or a final node as target must not be reversed.
Change Source/Target of Transition	Move the source/target of the transition to any other state.	In case the target state of the transition is changed, the target must not be the initial state. In case the source state of the transition is changed, the source must not be the final node.
Remove Trigger of Transition	Transforms the transition to a completion transition.	
Change Trigger of Transition	Change Operation to another one of the same interface.	

Table 1 – Mutation Operators for UML State Machines (UTF-AL)

If we perform mutation based on modeled uncertainties, we select any transition with a CallEvent as a trigger that refers to an operation that is subject of an Application Level Uncertainty with a direct impact, impacted element or that refers to an operation of a Component that is subject of an Application Level Uncertainty with a direct impact, impacted element.

4.2.3.4 Crossover

Test cases and the mutations they cover are the atomic piece of information to perform uncertainty testing at the application level. Therefore, we solely consider mutations along paths to perform crossover of state machines. We use the following crossover methods:

- Combine all mutations of both parents. This yields one new child UML state machine.
- Combine the mutations of the fittest path(s): This yields n^2 new state machines where n is the number of fittest paths considered for crossover.

4.2.3.5 Fitness

4.2.3.5.1 Fitness Calculation

As briefly mentioned in Section 4.2.3.1.2, we consider two types of fitness factors: use case-specific factors and system-based factors.

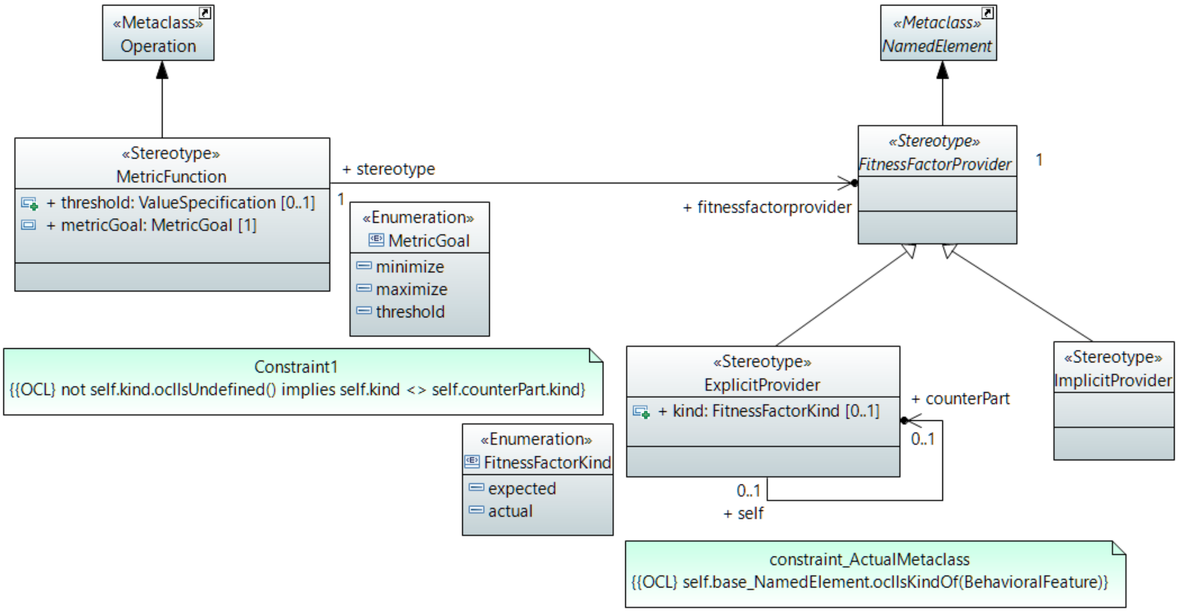


Figure 6. Model-based Means to Specify Fitness Factors

To specify use-case specific factors, UMF (see D2.2) provides stereotypes to identify elements that allow obtaining values from test runs («FitnessFactorProviders») that may be compared with an expected value by the corresponding counterpart («ExplicitProvider»). These can be used for instance, to measure the distance between a measured position and the actual position. For measured values without any comparative value, «ImplicitProviders» can refer to them. An optional threshold can be specified together with a 'metricGoal' that specifies whether the actual, measured value should be minimized, maximized or approach the threshold. In case of «ExplicitProviders», the difference between the actual and the expected value can be minimized or maximized. System-specific factors can be identified by the same means. Furthermore, we use generic measures such as response time, CPU load, and memory consumption if we can obtain these values from the SUT.

4.2.3.5.2 Fitness Mapping

Fitness values are calculated based on test logs obtained from test case executions. The mapping of calculated fitness values is done in two steps:

1. In the first step, we identify the path through the mutated state machine that represents the test case. We set the fitness value of the path to the fitness value of the test case.
2. In the second step, we aggregate the fitness values of all paths to the state machine.

4.2.4 Test Case Generation from Evolved State Machines

Test cases are generated from the evolved state machines by Microsoft's Spec Explorer [19]. To ensure that generated test cases are actually executable test cases, we maintain so-called execution invariants during test case generation. These execution invariants describe functional dependencies that have to be maintained under all circumstances, e.g., that a device has to be switched on before it can be configured. These invariants are expressed as behavioral descriptions with a stereotype «ExecutionInvariant» (see Figure 7). We employ UML Interactions to describe these dependencies. During test case generation, they are transformed such that they are maintained for all generated test cases.

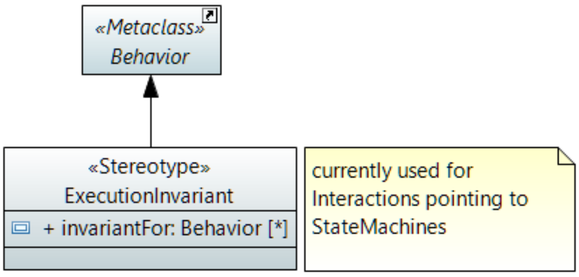

«Metaclass»
Behavior
«Stereotype»
ExecutionInvariant
+ invariantFor: Behavior [*]
currently used for Interactions pointing to StateMachines

Figure 7: Metamodel for ExecutionInvariants

We perform uncertainty testing and state machines at the application level. Describing valid interactions of the environment with the SUT, generating arbitrary paths through the evolved state machines may result in test cases that cover only original, non-mutated elements. In particularly, this may happen in the first generations of evolved state machines that contain only a few mutations. Therefore, we have to ensure that each path cover at least one mutation.

We use a profile to annotate which mutations are applied to state machines. This allow to identify the elements of an evolved state machine that have to be covered by all test cases and to configure Spec Explorer's test case generation engine accordingly.

Figure 8 depicts the profile for state machine mutation. A «UTestDirective» captures all the mutations in the form of «StateMachineMutation» for one individual.

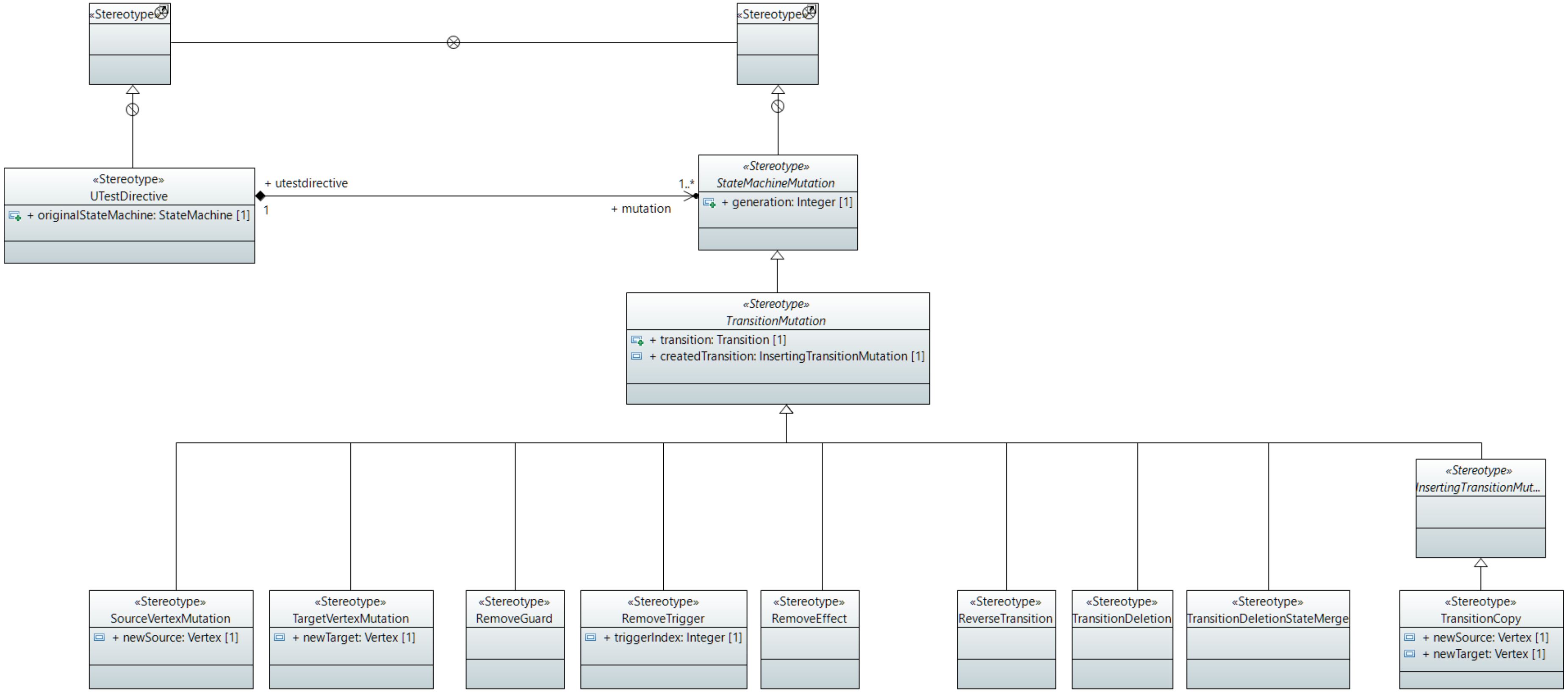


Figure 8 – Stereotypes for State Machine Mutations

4.3 Uncertainty Testing Framework At Infrastructure Level

In this section we describe our strategies for testing CPSs at the infrastructure level.

4.3.1 Test Coverage Strategies

Testing the infrastructure of CPSs brings its particular challenges due to the run-time uncertainty associated to the infrastructure. Infrastructure failures can appear due to incorrect infrastructure operation, such as unexpected infrastructure behavioral transitions. Failures can also appear at run-time in correctly operating infrastructures due to various causes, as captured in the infrastructure uncertainty taxonomy in Deliverable D1.2, Section 4.2 "Infrastructure level uncertainties properties classes". Thus, to correctly test and identify uncertainties in the infrastructure of CPSs, we have focused on:

- Testing the correctness of the infrastructure state transitions according to the CPS state transition belief model captured as state diagrams in D2.2.
- Testing at run-time if specific uncertainty-affected properties of CPSs still hold, indicating if an uncertain CPS behavior has occurred or not.

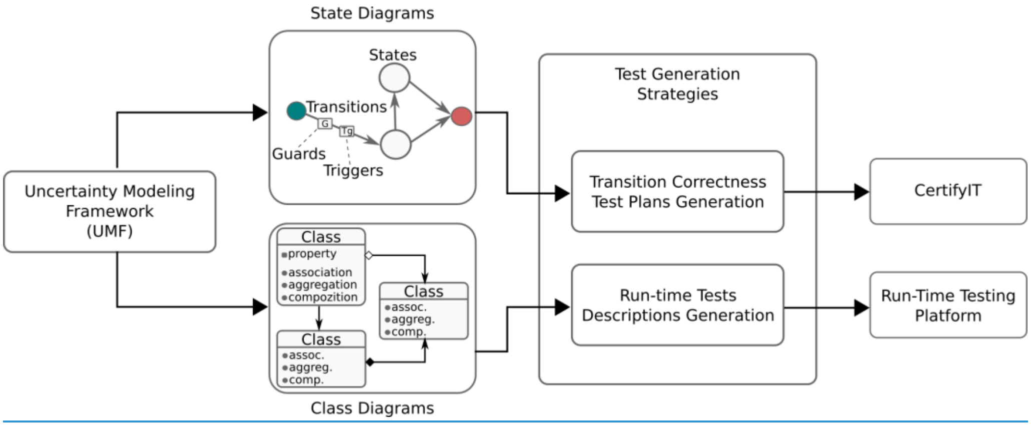

State Diagrams
States
Transitions
G
Tg
Guards
Triggers
Uncertainty Modeling Framework (UMF)
Class
property
association
aggregation
compozition
Class
assoc.
aggreg.
comp.
Class
assoc.
aggreg.
comp.
Class Diagrams
Test Generation Strategies
Transition Correctness Test Plans Generation
Run-time Tests Descriptions Generation
CertifyIT
Run-Time Testing Platform

Figure 9. Uncertainty Testing at Infrastructure Level Overview

Figure 9 show an overview of uncertainty testing at the infrastructure level of CPS. For testing uncertainty at infrastructure level we use as input UML Class Diagrams and State Diagrams described using the U-Test Uncertainty Modeling Framework. State Diagrams are used by our State Machine Transition Correctness Testing Strategies (Section 4.3.1.1). The transition correctness strategies generate abstract test plans, which after being implemented by a SUT expert, are executed by CertifyIT. In turn, the UML Class Diagrams describing the SUT components and uncertainty profile are used by our Run-Time Testing Strategies (Section 4.3.1.2) to generate run-time tests descriptions, which are used to implement and execute tests with TUW Platform for Run-Time Testing of Cyber-Physical Systems (Appendix, Technical Report 2).

4.3.1.1 State Machine Transition Correctness Testing Strategies

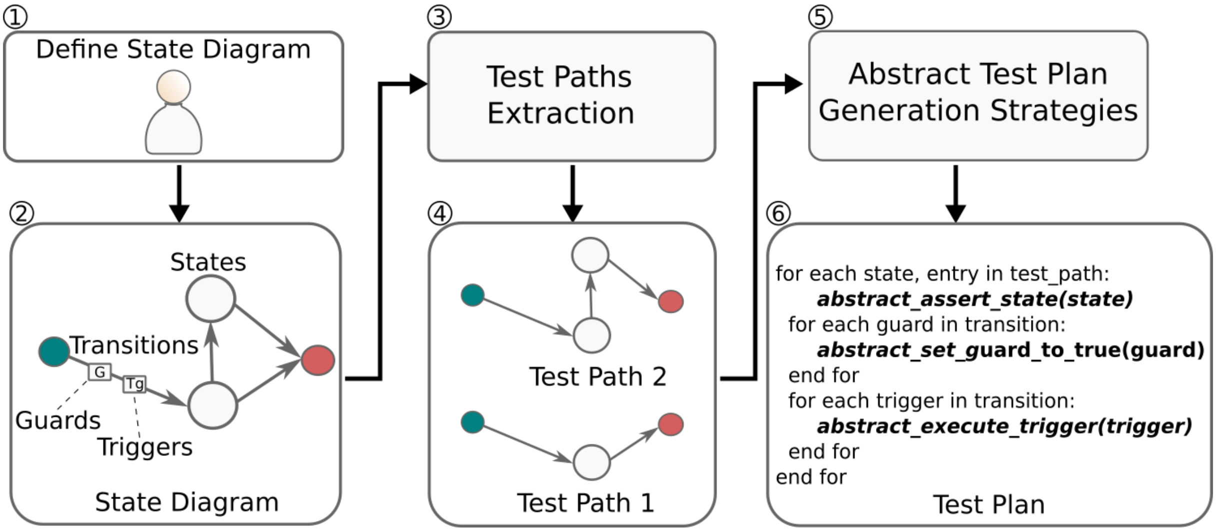
This first category of testing strategies focuses on testing if the SUT behaves as expected according to the expected behavior described as UML state diagrams. State diagrams describe the belief we have about what are the possible states of each system component, and the possible transitions between those states.

To test state transition correctness we define two test strategies for generating the test plan to be executed. The strategies consider the information in the state diagrams as state transition graphs, and focus on testing:

a) All state transitions paths starting with an initial state and ending in a final state.

b) Only state transitions paths, which pass at least one state that has at least one Infrastructure Uncertainty Stereotype (as defined in Uncertainty Core Profile in Deliverable D2.1, Section 4.2.1.3 Infrastructure Level) applied between an initial state and a final state.

We use the steps captured in Figure 10 to generate abstract test plans for testing correctness of CPS state transitions. We start from defining the UML state diagrams of CPS infrastructure use cases (Step 1-2). Then, we analyze each defined state machine, and determine the transition paths to be captured in the generated abstract test plan (Step 3-4). The particular mechanism for determining the transition paths to test is presented further in this section (Listing 1, and Listing 2). Further, for each test path, we generate an abstract test plan (Steps 5-6). In general, an abstract test plan will contain methods to: (i) assert that the CPS is in a particular state, (ii) invoke transition triggers needed to move the system to another state, and (iii) ensure the guard conditions on the inspected transition is true, so that we can follow in testing particular paths, in case transitions are determined by particular conditions.



```
for each state, entry in test_path:
     abstract_assert_state(state)
  for each guard in transition:
     abstract_set_guard_to_true(guard)
  end for
  for each trigger in transition:
     abstract_execute_trigger(trigger)
  end for
end for
```

Figure 10: Generating abstract transition correctness test plans flow

To generate abstract test plans testing the correctness of state transitions starting from state diagrams we first process each state diagram and determine the sequence of states and transitions, which must be tested. We consider each state diagram as describing a state graph, in which states represent graph vertices and state transitions the graph edges.

To traverse the state graph we employ a recursive algorithm, which starts from the initial state of the CPS (Listing 1) and on each encountered transition it branches and inspects each next state. The main logic for generating testing paths containing the states and transitions to test in a single test plan is captured in Listing 2. The used algorithm ensures that we consider a transition only once for a given test-plan. This ensures we do not enter infinite cycles. However, a test plan might contain the same state multiple times, if the state can be reached through a transition originating in a different state each time. Lines 10-12 of the algorithm depicted in Listing 2 check if the transition currently inspected has been encountered before in the particular test path under construction, and if yes, it does not consider the transition again.

```
ALGORITHM generate_test_paths_for_state_diagram
INPUT: states_transition_graph
OUTPUT: test_paths

-- start by analyzing the CPS initial state
SET initial_state TO states_transition_graph.get_initialState

SET test_path TO [initial_state]
CALL explore_state(initial_state, test_path, test_paths)

RETURN test_paths
```

Listing 1: Navigating state diagram and finding test paths algorithm - Part 1

In the `explore_state` function depicted in Listing 2 we start from a given state, the current test path we are constructing, and the list of test paths constructed by now. If the state, which we are processing, is a Final State, then we add the test path under construction to the generated test paths list. Otherwise, we iterate through each state transition possible from the current state (Line 7), and clone the current test path (Line 17). Cloning the path is required to ensure that for each possible transition we generate a different test plan. We add to the test plan generated so far the transition we are investigating and its end state to the test plan generated so far (Lines 18-19). We recursively call the `explore_state` function with the next state as parameter. After all recursive calls terminate we have a list of states and transitions (`test_paths)`, which we should consider testing. If an explored state path does not lead to a Final State and has no other transitions to explore, then a notification is shown to the user (Lines 5-6).

```
ALGORITHM explore_state
INPUT: state, test_path, test_paths

1  -- only add a test path if it can reach a final state
2  IF state IS FinalState THEN
3      CALL test_paths.add_path(test_path)
4    RETURN
5  ELSE IF state.get_transitions IS EMPTY DO
6      CALL notify_user(state)
7  END IF
8
9  FOR EACH transition IN state.get_transitions DO
10   -- do not take the same transition twice in the same path to avoid cycles
11   -- states can be taken more times
12   IF  transition IN test_path THEN
13     CONTINUE
14   END IF
15
16   -- for each new transition we have a potentially new testing path
17   SET new_path TO test_path.clone()
18   SET next_state TO transition.get_end_state
19   CALL new_path.add_entry([transition,next_state])
20
21  -- explore the next state indicated by the current transition
22   CALL explore_state(next_state, new_path, test_paths)
```

Listing 2: Navigating state diagram and finding test paths algorithm - Part 2

We leave the deciding on how and if we should test a particular test path to individual testing strategies. Currently we have implemented two strategies for testing state transition correctness, captured in Table 2. Each strategy takes as input a list of test paths and generates for each path an abstract test plan. Each abstract test plan contains a set of abstract methods to be implemented for particular CPSs. The methods belong to three categories: (i) methods returning void, (ii) methods invocating particular CPS APIs, and (iii) methods asserting the CPS is in a certain state. The first category is used for creating methods, which initialize the CPS to its Initial State, and clean up after the test has terminated. Methods invocating particular CPS APIs are used to activate the CPS state transition triggers, and ensure for each tested transition that its guard conditions are fulfilled.

Based on the generated test paths, we now generate abstract test plans for each path, using one of our test plan generation strategies. Each strategy performs the following steps:

- Starts from UML state diagrams capturing the CPS states and transitions between states
- Processes the "Triggers" and "Guards" placed on the state transitions
- Processes the Uncertainty Stereotypes associated to CPS states
- Produces an "abstract" test plan, which must be implemented and made concrete before executing them on particular CPS.

Table 2: Supported test plan generation strategies for testing state transition correctness

Test Plan Generation Strategies		Description
1	Test correctness of state transitions in *all* test paths	Tests all state transition paths starting with an initial state and ending in a final state.
2	Test correctness of state transitions in *uncertainty affected* test paths.	Generates test plans only for test paths, which have at least one state with at least one Infrastructure Uncertainty Stereotype, as defined in Uncertainty Core Profile in Deliverable D2.1, Section 4.2.1.3 Infrastructure Level.

4.3.1.1.1 *Strategy for Testing Correctness of State Transitions in All Test Paths*

Listing 3 depicts the algorithm of test transition correctness strategy 1. We iterate through each test path entry containing a state and the transition to the next path state (Lines 3-5). For each state we encounter we add an additional set of abstract methods to the test plan generated. Depending on the type of state we encounter, we generate different methods in the abstract test plan. If the state is the *Initial State*, we generate and add to the test plan an abstract method (Lines 10-12), which must be implemented to initialize the CPS under test. Similar, when we encounter a *Final State* we generate a method for cleaning up after the test execution, and add the generated test plan to a list of test plans returned by our approach. If navigating the state diagram we encounter a state different than Initial and Final state, then we first generate a method to be used in asserting that the system has reached the expected state (Lines 21-22). Next, we look at the transition captured in the investigated test path, taking the system from this state to another. If the transition has guard conditions, they indicate the transition takes place only IF certain conditions are met. Thus, we add to the generated test plan, for each guard condition, an abstract method to be implemented to force the system (if necessary) to fulfill the transition conditions, to ensure the system follows the transition path we are investigating and testing (Lines 26-36). Transitions can also have associated triggers, i.e., actions that must be taken on the system for the transition to take place. Thus, for each transition trigger, we add to the test plan an abstract method used to invoke the trigger and force the transition to take place (Lines 38-40).

```
ALGORITHM generate_transition_correctness_tests
INPUT: test_paths
OUTPUT: test_plans

FOR EACH test_path IN test_paths DO
  SET test_plan TO EMPTY
  FOR EACH entry IN test_path:
    SET state TO entry.state
    SET transition TO entry.transition

    IF state is InitialState THEN
       -- for the initial state we create an abstract method which must be
       -- implemented to do all required initial setup of the system
       SET abstract_method TO generate_abstract_method(return="void",
                                              name="SetupInInitialState")
       test_plan.add(abstract_method)
    ELSE IF state is FinalState THEN
       -- for final state the method implementation should perform necessary
       -- cleanup after testing
       SET abstract_method TO generate_abstract_method(return="void",
                                              name="CleanupInFinalState")
       test_plan.add(abstract_method)
    ELSE
       -- create method to assert that we have reached the current state
       SET abstract_assert_method TO
                    generate_abstract_assert_method(return="boolean",
                                              state_name=state.name)
       test_plan.add(abstract_assert_method)

       FOR EACH guard in transition.guards:
            -- for each transition, force if needed the CPS to a state in
            -- which each Guard returns OK
            -- CPS might have multiple transitions from a state, depending on
            -- the guard value
            SET abstract_invocation_method TO generate_invocation_method(
                                              target=guard.target,
                                              parameter=guard.parameter,
                                              value=trigger.value)
            test_plan.add(abstract_invocation_method)
       END FOR

       FOR EACH trigger in transition.triggers:
          -- for each transition, invoke all triggers to perform the
          -- transition, and check all guard conditions,
          -- and assert that the state we reach after the transition is the
          -- expected state
          SET abstract_invocation_method TO generate_invocation_method(
                                              target=trigger.target,
                                              method=trigger.method,
                                              args=trigger.parameters)
          test_plan.add(abstract_invocation_method)
       END FOR
    END IF
  END FOR
  -- add generated abstract test plan
  test_plans.add(test_plan)
END FOR
RETURN test_plans
```

Listing 3: Test strategy 1 - Generating abstract transition correctness test plans from test paths

4.3.1.1.2 Strategy for Testing Correctness of State Transitions in Uncertainty-affected Test Paths.

Listing 4 depicts the algorithm of test transition correctness strategy 2, in which we consider for testing the generated test paths, which pass through at least one state, which has at least, one Infrastructure Uncertainty stereotype applied to it, as defined in Uncertainty Core Profile in Deliverable D2.1, Section 4.2.1.3 Infrastructure Level. The only difference between test transition correctness strategy 1 and 2, is that in the uncertainty-based strategy we add an additional verification (at Line 3) to check if any state in the test path has *InfrastructureUncertainty*.

```
ALGORITHM generate_transition_correctness_tests_with_uncertainty
INPUT: test_paths
OUTPUT: test_plans

1 FOR EACH test_path IN test_paths DO
2
3    IF ANY state in test_path.entries HAS InfrastructureUncertainty
4      SET test_plan TO EMPTY
```

Listing 4: Test strategy 2 - Generating abstract transition correctness test plans only for test paths under uncertainty

4.3.1.2 Run-Time Testing Strategies

This second category of infrastructure testing strategies focuses on testing if the SUT behaves as expected at run-time. Testing only transition correctness cannot determine all health problems that can occur at run-time for complex systems due to uncertain behaviors. Thus, testing CPSs at run-time is very important for determining if uncertain behaviors have taken place causing failures or behavioral errors. Further, complex CPSs can contain components acquired from third parties, which are “black box” to the tester, i.e., they allow no access to their inner workings. In may cases, testing third party components might only be possible at run-time. This is because not enough knowledge about their behavior is available for performing transition correctness testing. Examples of such third-party components are the X4 units in the GeoSports use case, or the Scanner in the ULMA Handling Systems Demonstrator use case, described in U-Test deliverable D1.1.

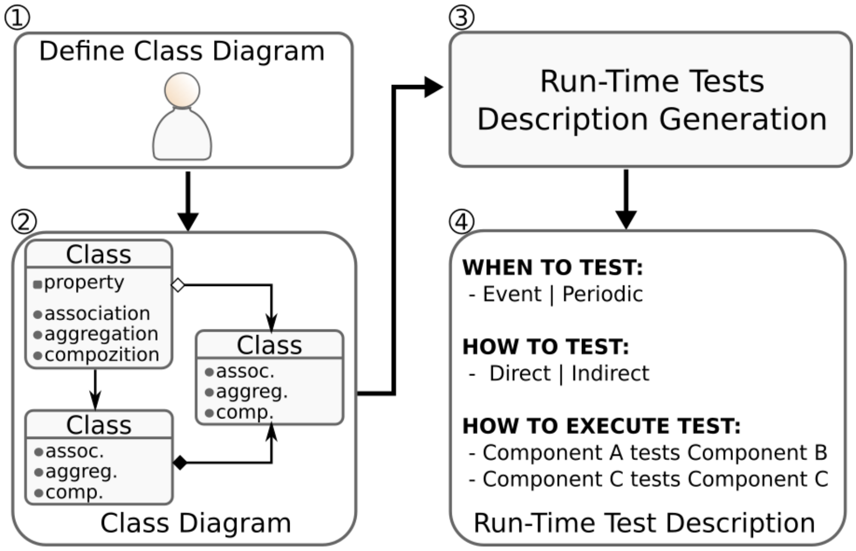


Figure 11: Generating run-time tests descriptions flow

Our approach for run-time testing on CPS is captured in Figure 11, and relies on UML Class Diagrams defined for each CPS, which must be tested (Steps 1-2). In the approach we use a set of strategies, which process the class diagrams (Step 3), and generate descriptions of what, how, and when to test at run-time (Step 4). The result of our approach is a collection of descriptions, which should be used at run-time to execute concrete tests implemented for particular systems. In the following we first discuss the content of the generated test descriptions, after which we introduce our strategies for processing UML Class Diagrams and generating the test descriptions.

4.3.1.2.1 Run-time Tests Description

Given that uncertain behaviors can appear anytime during the run-time of a system, we have first identified when to execute run-time tests. We have identified two testing times: (i) periodic testing, and (ii) event-based testing.

4.3.1.2.1.1 When to test infrastructure at run-time

a. Periodic testing

In *Periodic* testing, tests must be executed at certain time intervals, depending on the tested component and the nature of the uncertainties the component can exhibit. One periodic test can cover one or more uncertainties.

Example of periodic test: GeoSports use case, UC1_INFR_1_QUUPPA

- Test every minute that heartbeat detected by body-sensor is > 0

b. Event-based testing

In Event-based testing, tests must be executed after a certain event has occurred in the CPS. An event can be associated to a particular CPS component, e.g., if component started/stopped.

An event can also be associated to particular CPS component states. Example:

- ULMA Handling Systems Demonstrator, UC2_INFR_1.1:
    - IF “Packet received by the wearehouse” event then TEST Scanner State != 0 (scanner is operating)
- FPX/NMT GeoSports use case UC1_INFR_1_QUUPPA:
    - IF “Trainer activates Quuppa sensor” then test “body sensor measures athlete heartbeat”

An event could also be associated with the result of another test execution, to enable uncertainty identification through more fine-grained tests executed on demand in case of event. Example:

- FPX/NMT GeoSports use case UC1_INFR_1_QUUPPA:
    - IF "heartbeat detected by body-sensor is > 0" fails, then test "Quuppa is active"

4.3.1.2.1.2 How to test infrastructure at run-time

Testing infrastructure at run-time implies considering the particular capabilities of the CPSs components, and their testing capabilities. While one might be able to directly access and test a component over which one has full control, third party components might be more difficult to test directly. Thus, to test at run-time CPSs we focus on two testing mechanisms:

a. Direct testing:

- A direct test is executed against the API/capability of the tested component.
- Example: ULMA Handling Systems Demonstrator, UC2_INFR_1.1
    - Testing that Scanner status is behaving normally by leveraging Scanner API to retrieve Scanner Status value
    - "assert scanner.Status == 0"

b. Indirect testing:

- An indirect test is executed against the API/capability of another system component.
- Example: FPX/NMT GeoSports use case, UC1_INFR_1_QUUPPA
    - Assert Quuppa body sensor measures athlete heartbeat by collecting measurement data through invoking API of the Quuppa management service.
    - "assert managementService.CollectedHeartbeat != empty and values (managementService.CollectedHeartbeat) != null"

4.3.1.2.2 Strategies for generating run-time tests descriptions

For generating the descriptions on how/what/when to test at run-time we employ a conceptual strategy described as pseudo-code in Listing 5. In our approach we first take as input an UML Class. For each of the class attributes, we generate a description for executing a "direct" test to verify if the value of the attribute is within expected values (Lines 4-7). In this conceptual representation we leave out the particular implementation of the test description generation (Line 5), as this must be done according to particular run-time tests execution mechanisms. For a particular implementation please check Technical Report 1 in Appendices. Further, we iterate through each association relationship found on the UML class. An association between two UML Classes implies communication between CPS components of the type represented by the classes. Thus, for any association we generate a test description for an "indirect" test (Lines 12-16). The generated indirect test descriptions are meant to state that the components belonging to the currently investigated UML Class must check that the components, which they are accessing, are accessible. As in the case of the tests generated for class attributes, we leave out the implementation of the particular test description generation (Lines 13-14). Our approach iterates through all classes captured in a CPS class diagram and for each class returns the descriptions of the direct and indirect tests, which should be executed, on the CPS at run-time.

```
ALGORITHM generate_run_time_tests_description
INPUT:  uml_class
OUTPUT: [direct_tests_descriptions, indirect_tests_descriptions]

1  -- generate description for testing the value of each attribute
2  -- we assume each attribute can be tested directly on the CPS component
3  -- represented by the UML class
4  FOR EACH attribute in uml_class.get_attributes DO
5       SET test_description TO generate_direct_test(uml_class, attribute)
6       CALL direct_tests_descriptions.add(test_description)
```

```
END FOR

-- generate description for testing connectivity between CPS components
-- for each UML association relationship present on the uml_class
-- connectivity test descriptions are generated as indirect tests
FOR EACH association in uml_class.get_relationships DO
    SET test_description TO generate_indirect_test(uml_class,
                                                   association.associated_class)
    CALL indirect_tests_descriptions.add(test_description)
END FOR

RETURN [direct_tests_descriptions, indirect_tests_descriptions]
```

Listing 5: Run-time testing - Generating test from UML class diagrams

4.3.2 Test Data Generation

Our approach relies on two data sources used in infrastructure testing:

a. Information captured as UML Profiles, Class Diagrams, and State Diagrams during the CPS modeling phase, as described in D2.1 and D2.2. This information is used in generating the abstract transition correctness tests and run-time test descriptions.
b. Expert knowledge brought by CPS owner/user used in the implementation of the concrete tests according to particularities of the tested CPS.

4.3.3 Uncertainty Model Evolution at Infrastructure Level

The focus of our work so far was to generate tests and test CPSs at the infrastructure level. We have not defined so far approaches for model evolution at infrastructure level.

4.4 Uncertainty Testing Framework at Integration Level

This section presents the overview of the work related to UTF at the integration level from the following fours perspectives as shown in Figure 12: 1) Test Model Evolution (C1), 2) Test Case Generation (C2), 3) Uncertainty-Based Test Case Minimization (C3), 4) Test Case Execution (C4). This subsection only provides an overview of each of these activities and all the technical details are provided in the form of two technical reports ([30] and [29]) attached with this deliverable as two separate documents (**TR4.pdf** and **TR5.pdf**). Test Model Evolution (C1) is described in Section 1.1.1, Test Case Generation (C2) in Section 4.4.2, Uncertainty-based Test Case Minimization (C3) in Section 4.4.3), and Test Case Execution (C4) in Section 4.4.4.

As shown in Figure 12, the initial input of the UTF (C0) at Integration Level (UTF at Integration Level), i.e., test ready model is the output of the UMF as presented in the deliverables D2.1/D2.2.

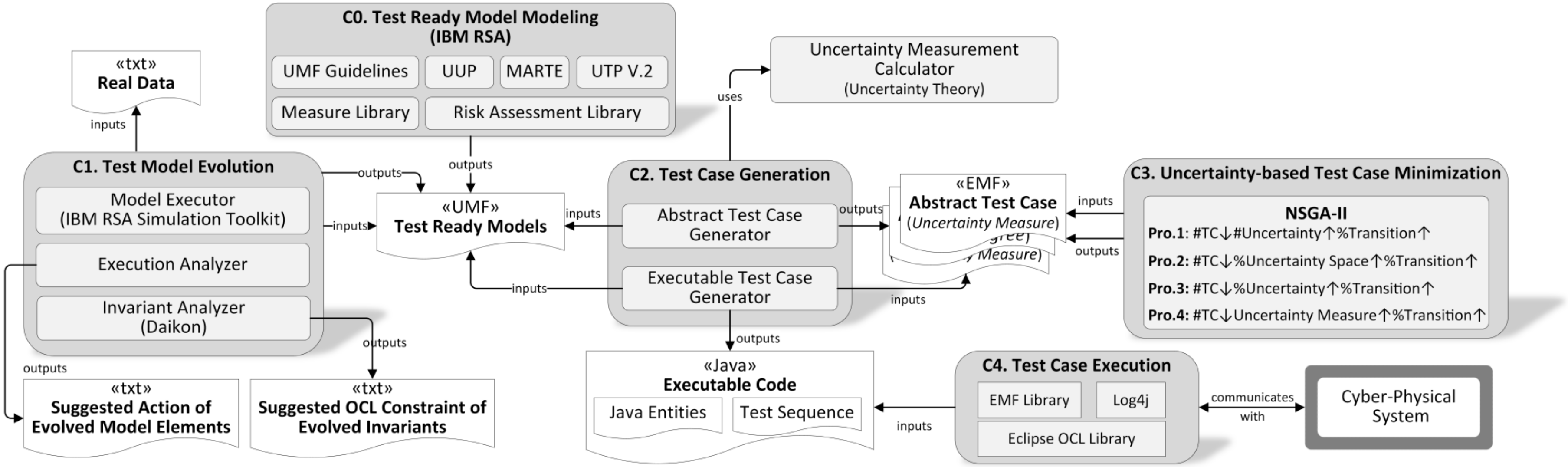


Figure 12. The Overall of Uncertainty Testing Framework at Integration Level

4.4.1 Uncertainty Model Evolution at Integration Level

As shown in Figure 12, the key inputs of Uncertainty Test Model Evolution are initial test ready models (outputs of UMF) of CPS with specified subjective belief information (belief agent, uncertainty and related measurement) and real data collected in the GeoSport case study (X4 device). The process of evolving test ready model contains three main steps 1) verifying the initial test ready model applied UMF (D2.1/D2.2) against real data via model execution by using IBM RSA Simulation Toolkit, then outputting the suggested actions of evolved model elements to User to update the test ready model 2) Evolving subjective uncertainty measurements based on the evolved objective uncertainty measurement via model execution and 3) Evolving state invariants (test oracle) and guards of transitions by using dynamic invariant detector- Daikon, then outputs the suggested OCL constraints of evolved invariants to User to update the test ready model accordingly. More details including the evaluation are presented in the **TR4.pdf** attached with this deliverable (see the Appendix of this deliverable). The TR is also available online at [30].

4.4.2 Test Case Generation at Integration Level

As shown in Figure 12, the input of Test Case generation (C2) is test ready model, which could be 1) initial version from C0, or 2) evolved version from C1. The process of test case generation contains three main steps:

1) Generating abstract test cases by using JGraph [22] according to two developed test case generation strategies: *All Simple Path* (ASP) and *All Paths with Maximum Length* (APML),

2) Generating the uncertainty measurement based on the developed Uncertainty Measurement Calculator, which is implementation of Uncertainty Theory, and

3) Generating executable test cases based on generated/minimized abstract test cases.

More details including evaluations are presented in the **TR5.pdf** attached with this deliverable (also available online at [29]) (see the Appendix of this deliverable).

4.4.3 Test Case Minimization at Integration Level

The test case minimization (C3) at integration level is proposed, because the number of abstract test cases generated by C1 is typically very large for any non-trivial CPS and it is impossible to execute all of them. As shown in Figure 12, the problems developed in the test case minimization is a multi-objective search problem and thus we opted for the commonly used multi-objective search algorithm, i.e. NSGA-II, based on the guideline [5]. Further we develop four concrete search problems, aiming to minimize the number of test cases and maximize coverage of transition, according to these four objectives respectively 1) The average number of uncertainties covered by the subset of the test cases after minimization; 2) The average percentage of uncertainty space covered by the subset of the test cases after minimization; 3) The average *uncertainty measure* (UM, defined in appendix) of the subset of test cases after minimization; 4) The average number of unique uncertainties covered by the subset of test cases after minimization. More details including evaluations are presented in the **TR5.pdf** attached with this deliverable as a separate document and is also available online at [29] (see the Appendix of this deliverable).

4.4.4 Test Case Execution at Integration Level

During the test execution (C4) as shown in Figure 12, 1) the test oracles (constraints) are checked at runtime by integrated EMF Library and Eclipse OCL, 2) the test data are generated by EsOCL before the test execution or generated randomly at runtime, 3) the indeterminacy sources are introduced into test execution based on specified constraints applied «IndeterminacySource», and 3) the test log are handled by the integrated Log4j. More details including evaluations are presented in the **TR5.pdf** attached with this deliverable as a separate document and is also available online at [29] (see the Appendix of this deliverable).

5 Summary and Conclusion

In this section, we summarize the overall achievements of our work according to Milestone 3 (M3) for developing the UTF to support uncertainty testing at application level (Section 5.1), infrastructure level (Section 5.2), and integration level (Section 5.3). For each level, we also present our plan to achieve Milestone 4 (M4).

5.1 UTF at Application Level

Achievement of M3

For the UTF V.1 at the Application level, we developed a search-based approach aiming at discovering both, known and unknown uncertain behaviors. We adapted the building blocks of the genetic algorithm to uncertainty testing, employing information from modeled uncertainties and the corresponding use cases in form of UML state machines, and provided model-based means for describing fitness factors and mutations to UML state machine transitions. Through test coverage strategies, we are able to guide the search-based testing process.

Plan for achieving M4

The main focus of the developments for the next milestone is improving the genetic algorithm with mutation operators for guards and effects in form of UML activities, investigate options for exploiting more information from uncertainties and optimizing the fitness function in order to increase the chances for discovering unknown uncertain behavior.

5.2 UTF at Infrastructure Level

Achievement of M3

The second milestone was achieved through the introduction of two classes of test strategies for infrastructure level: (i) state machine transition correctness strategies, and (ii) run-time testing strategies. For each strategy class we have introduced two test-plan generation strategies. The strategies where implemented as IBM RSA Plug-Ins, ensuring their applicability on the UML Models developed in the context of WP2.

The strategies from the first category generate abstract test plans. These abstract test plans when implemented as Junit tests can be executed by CertifyIt. The run-time testing strategies generate descriptions of run-time tests, which are usable in conjunction with a Run-Time Testing Platform provided by TUW (described in this deliverable in Technical Report 2).

We have applied the test strategies to generate test plans for UC1_INFR_1_QUUPPA and UC2_INFR_1.1 use cases modeled in D2.1.

Plan for achieving M4

For M4 we will focus on providing the first version of algorithms for model evolution. We plan to start from the Uncertainty Taxonomy at the Infrastructure Level, and analyze the State and Class diagrams of each U-Test use case. Based on the taxonomy we will be able to detect where uncertainties can appear in each use case, uncertainties not captured during the modeling phase, and enrich the U-Test use cases models with them.

5.3 UTF at Integration Level

Achievement of M3

We have successfully reached the M3 regarding the UTF V.1 for uncertainty testing of the Integration level of CPS. More specifically, we developed the evolution approach for Uncertainty Test Model at the Integration level, which aims to improve the quality of test ready models, evolve captured uncertainty, and potentially discover unknown uncertainty and model elements. Secondly, test case generation strategies have been developed to generate abstract test cases as well as executable test cases. The test case generation process can be driven by uncertainty measurement based on the developed Uncertainty Measurement Calculator. Thirdly, because executing the large number of generated test cases is impractical, we proposed a search-based approach to minimize the number of test cases while maximizing the coverage of transition. Last but not least, we developed a prototype as a proof-of-concept for our approaches above, in which the generated test cases can be executed. The test cases were generated with the input is the test ready models at the integration level (covering 66% of use cases at the integration level for M3).

Plan for achieving M4

As we have made a concrete foundation in the UTF V.1 for uncertainty testing of the Integration level of CPS, we will continue improving our framework along the way to M4. In achieving M4, we will use the test ready models, which cover 100% the use cases at the integration level, for test generation.

Appendix

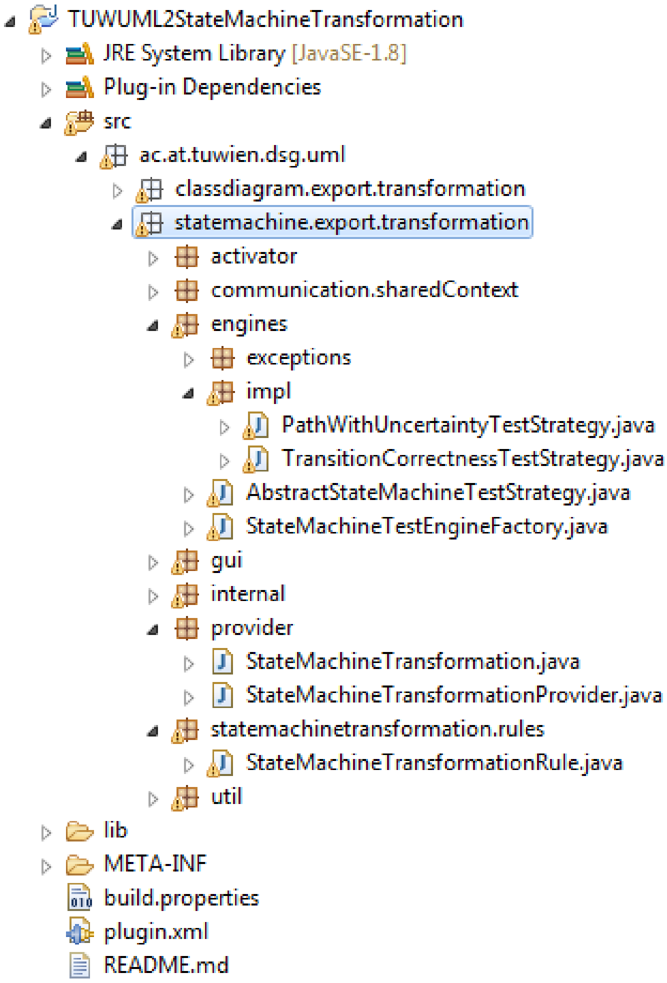
The Appendix section summarizes the technical reports that provide more detailed technical aspects of the UTF.

Technical Report 1 - Integrating Transition Correctness Test Strategies in IBM RSA 9.5

In this report we discuss how we apply and integrate our strategies for generating transition correctness tests described in Section 4.3.1.1 “State Machine Transition Correctness Testing Strategies” in IBM Rational Software Architect (IBM RSA). We use for the infrastructure level modeling of CPSs as primary modeling tool version IBM Rational Software Architect (IBM RSA) 9.5. IBM RSA is one of the most mature and widely used UML modeling tools. By employing it to create our models we increase the adoption chances of our approach, as our models can be imported and used in other IBM RSA instances.

Thus, we implement a series of IBM RSA Plug-Ins for applying the transition correctness test plan generation strategies described in Section 4.3.1.1 “State Machine Transition Correctness Testing Strategies” over the CPS infrastructure models described in D2.1 and D2.2. IBM RSA is built on top of Eclipse Integrated Development Environment, which provides and easy to use extension mechanism relying on Plug-Ins.

Transition Correctness Tests Generation IBM RSA Plug-In



```
◢ TUWUML2StateMachineTransformation
    ▷ JRE System Library [JavaSE-1.8]
    ▷ Plug-in Dependencies
    ◢ src
        ◢ ac.at.tuwien.dsg.uml
            ▷ classdiagram.export.transformation
            ◢ statemachine.export.transformation
                ▷ activator
                ▷ communication.sharedContext
                ◢ engines
                    ▷ exceptions
                    ◢ impl
                        ▷ PathWithUncertaintyTestStrategy.java
                        ▷ TransitionCorrectnessTestStrategy.java
                    ▷ AbstractStateMachineTestStrategy.java
                    ▷ StateMachineTestEngineFactory.java
                ▷ gui
                ▷ internal
                ◢ provider
                    ▷ StateMachineTransformation.java
                    ▷ StateMachineTransformationProvider.java
                ◢ statemachinetransformation.rules
                    ▷ StateMachineTransformationRule.java
                ▷ util
    ▷ lib
    ▷ META-INF
      build.properties
      plugin.xml
      README.md
```

Figure 13: Transition Correctness Strategy IBM RSA Plug-In

We implement our strategies in Java as a specific type of plug-ins called “Transformation Extensions” (code depicted as Eclipse project in Figure 13).

We have named our Transformation plug-in “StateMachineTransformation”, belonging to a plug-ins group called “ac.at.tuwien.dsg.uml.statemachine.export.transformation”.

The Plug-In code is located in an open-source GitHub code repository at https://github.com/tuwiendsg/COMOT4U/tree/master/T4U/UncertaintyTestsGeneration/TUWUML2StateMachineTransformation. The particular classes implementing the plug-in are located in the “src/ac/at/tuwien/dsg/uml/statemachine/export/transformation/” subdirectory.

The StateMachineTransformation is applicable only to UML State Diagrams. The plug-in extends the “com.ibm.xtools.transform.core.RootTransform”, enabling it to access state diagram information relying on Eclipse UML2 API from package “org.eclipse.uml2.uml.” We define an internal representation model for the states graph in the StateMachineStateGraph class. Based on the StateMachineStateGraph class we implement the two test plan generation strategies described in Section 4.3.1.1: class TransitionCorrectnessTestStrategy implements test plan generation strategy 1, while class PathWithUncertaintyTestingStrategy implements the second strategy considering uncertain test paths.

The glue between the test plan generation strategies and the UML State Diagram processing is implemented in StateMachineTransformationRule class. StateMachineTransformationRule takes as input an UML State Diagram represented as a org.eclipse.uml2.uml.StateMachine object, parses the object, and constructs a StateMachineStateGraph. Further, it calls the test plan generation strategy in use to generate based on the StateMachineStateGraph a test plan. The test plan is generated relying on Eclipse AST/DOM API to generate Java class files. Thus, an abstract test plan is generated as an abstract Java class. In order to execute the test plan on particular CPSs, a concrete class implementing all test plan abstract methods must extend the abstract class.

Using the Transition Correctness Tests Generation IBM RSA Plug-In

To use the plug-in please follow the following steps:

Install:

- Import the plugin project in your RSA run-time and deploy it to local host (or install plug-in Jar from deployment).
- Or just run as Eclipse Application the plug-in project (no install required, local run)

Use:

- Create a new Transformation Configuration: "File -> New -> Other -> Transformations -> Transformation Configuration".
- Choose transformation type: Check "show all transformations".
- Choose "ac.at.tuwien.dsg.uml.statemachine.export.transformation"
  -> "StateMachineTransformation" (Figure 14).
- From transformation "Source and Target" tab select source state machine diagram and target location where transformation file should be generated (Figure 15).
- From "Select Test Generation Strategy" tab select the test generation strategy to use when generating the test plan (Figure 16).
- Execute the transformation (Figure 17).
- Right click on Target project and Refresh
- Test abstract plan generated file will be named as
  [TestStrategySimpleName]_TestPlanForStateMachine_[StateMachineName] (Figure 18).

Extend:

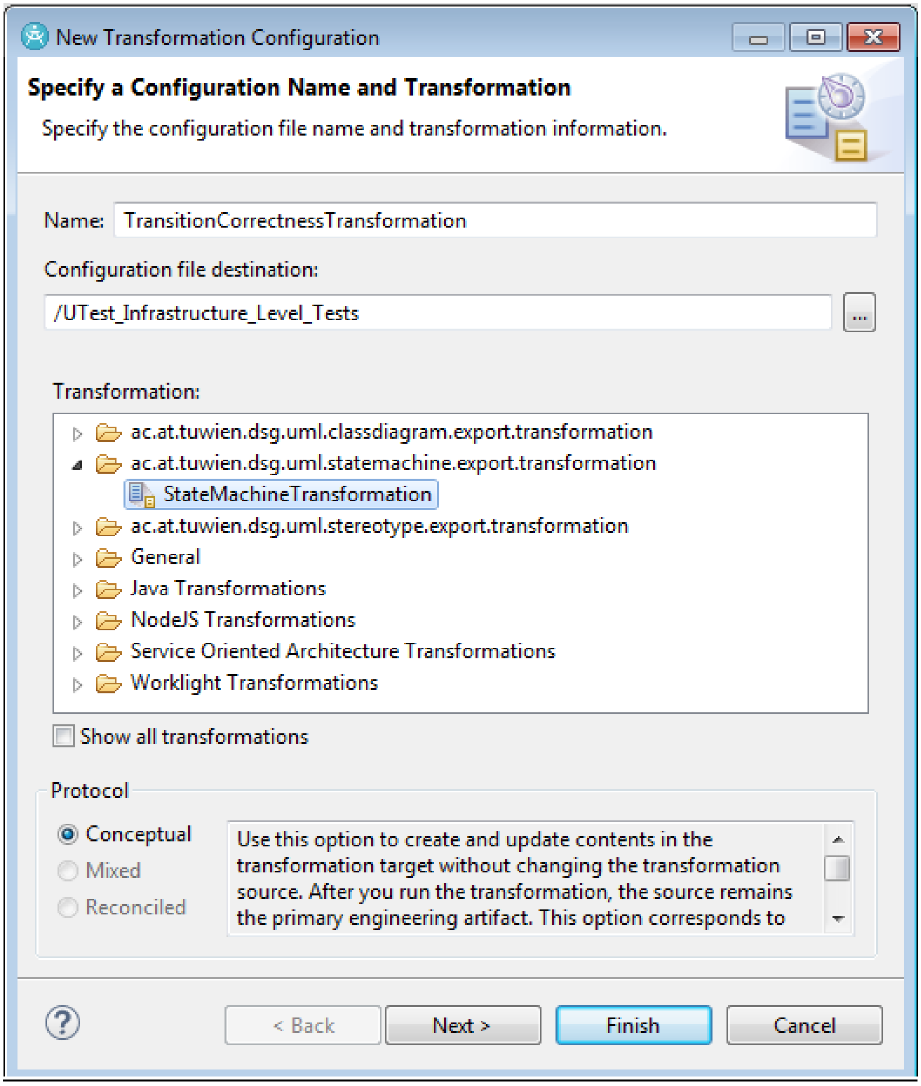
To extend our plug-in with other test generation strategies one must consider the most important Java classes we provide in our implementation:

1. Most important classes for extending the plug-in are:

- AbstractStateMachineTestStrategy: class which has the abstract "generateTestPlan" method. The class must be sub-classed and the method implemented for any custom strategy.
- StateMachineTestEngineFactory: factory class holding all supported strategies. Any new subclass of AbstractStateMachineTestStrategy must be added in "supportedStrategies". After this they will appear on the selection tab and be called when required.
- StateMachineStateGraph: class holding state machine parsed information. The class has methods ("toString", and "getStatesWithUncertainties()") which give good examples on how to go deeper and extract state information such as uncertainties, and the uncertainties' parameters.
- StateMachineTransformationRule: class which contains the "createTarget" method which creates the StateMachineStateGraph from a state machine diagram. If user wants to add more information to StateMachineStateGraph then the rule can be modified to extract more information from the machine diagram.

Usage example screenshots:

In the following screenshots we exemplify the usage of our State Machine Transformation Plug-In in IBM RSA. We apply our plug-in to ULMA Infrastructure profile described in D2.1 Section 5.3.2.



New Transformation Configuration

Specify a Configuration Name and Transformation

Specify the configuration file name and transformation information.

Name: TransitionCorrectnessTransformation

Configuration file destination: /UTest_Infrastructure_Level_Tests

Transformation:

- ac.at.tuwien.dsg.uml.classdiagram.export.transformation
- ac.at.tuwien.dsg.uml.statemachine.export.transformation
  - StateMachineTransformation
- ac.at.tuwien.dsg.uml.stereotype.export.transformation
- General
- Java Transformations
- NodeJS Transformations
- Service Oriented Architecture Transformations
- Worklight Transformations

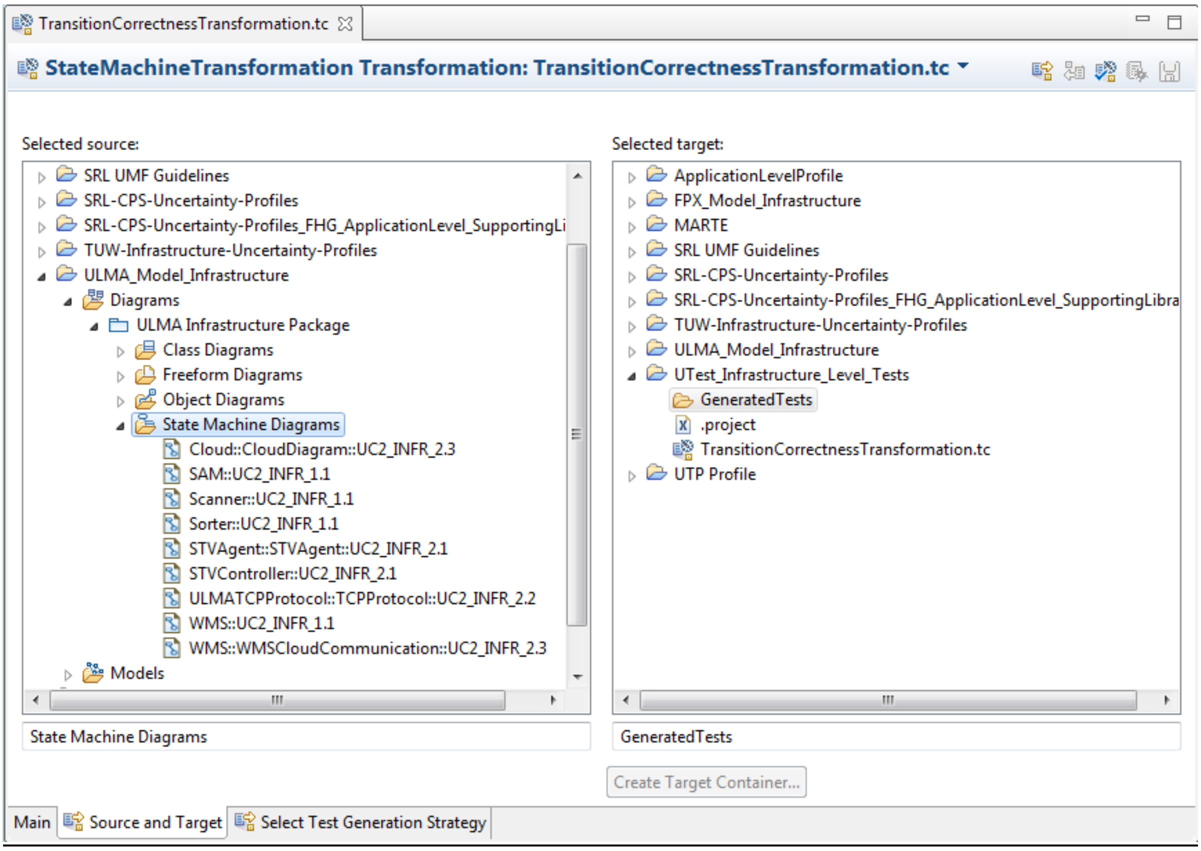
☐ Show all transformations

Protocol

● Conceptual
○ Mixed
○ Reconciled

Use this option to create and update contents in the transformation target without changing the transformation source. After you run the transformation, the source remains the primary engineering artifact. This option corresponds to

Figure 14: Creating a new State Machine Transformation Configuration



TransitionCorrectnessTransformation.tc

StateMachineTransformation Transformation: TransitionCorrectnessTransformation.tc

Selected source:

- SRL UMF Guidelines
- SRL-CPS-Uncertainty-Profiles
- SRL-CPS-Uncertainty-Profiles_FHG_ApplicationLevel_SupportingLi
- TUW-Infrastructure-Uncertainty-Profiles
- ULMA_Model_Infrastructure
  - Diagrams
    - ULMA Infrastructure Package
      - Class Diagrams
      - Freeform Diagrams
      - Object Diagrams
      - State Machine Diagrams
        - Cloud::CloudDiagram::UC2_INFR_2.3
        - SAM::UC2_INFR_1.1
        - Scanner::UC2_INFR_1.1
        - Sorter::UC2_INFR_1.1
        - STVAgent::STVAgent::UC2_INFR_2.1
        - STVController::UC2_INFR_2.1
        - ULMATCPProtocol::TCPProtocol::UC2_INFR_2.2
        - WMS::UC2_INFR_1.1
        - WMS::WMSCloudCommunication::UC2_INFR_2.3
  - Models

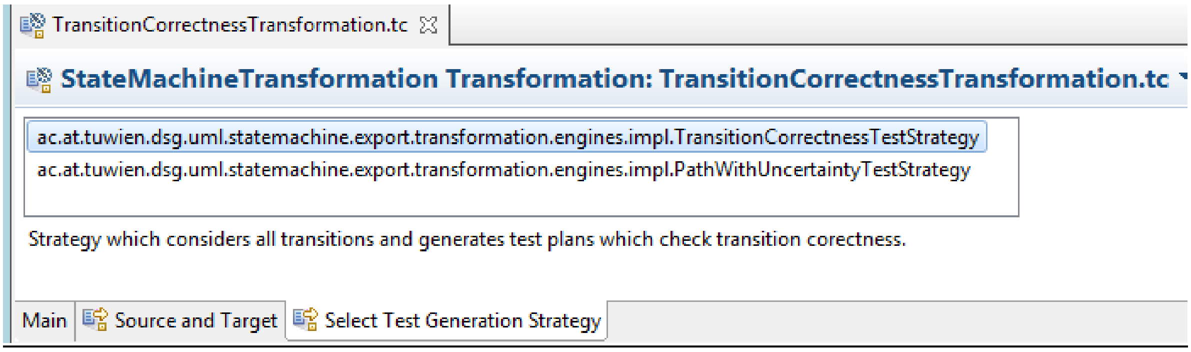
State Machine Diagrams

Selected target:

- ApplicationLevelProfile
- FPX_Model_Infrastructure
- MARTE
- SRL UMF Guidelines
- SRL-CPS-Uncertainty-Profiles
- SRL-CPS-Uncertainty-Profiles_FHG_ApplicationLevel_SupportingLibra
- TUW-Infrastructure-Uncertainty-Profiles
- ULMA_Model_Infrastructure
- UTest_Infrastructure_Level_Tests
  - GeneratedTests
  - .project
  - TransitionCorrectnessTransformation.tc
- UTP Profile

GeneratedTests

Figure 15: Choosing the State Diagram used in the Test Generation



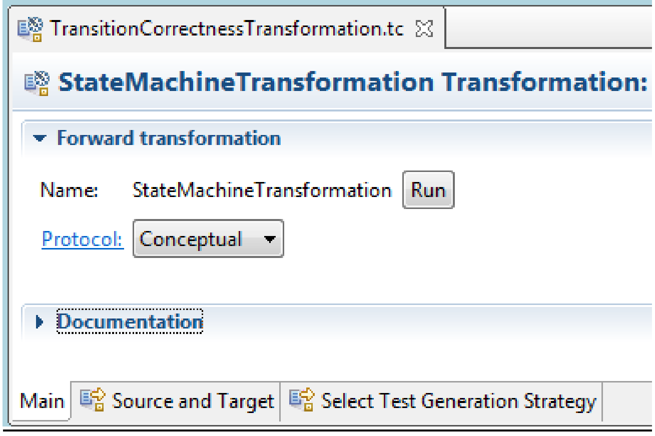
TransitionCorrectnessTransformation.tc

StateMachineTransformation Transformation: TransitionCorrectnessTransformation.tc

ac.at.tuwien.dsg.uml.statemachine.export.transformation.engines.impl.TransitionCorrectnessTestStrategy
ac.at.tuwien.dsg.uml.statemachine.export.transformation.engines.impl.PathWithUncertaintyTestStrategy

Strategy which considers all transitions and generates test plans which check transition corectness.

Figure 16: Choosing Test Plan Generation Strategy



TransitionCorrectnessTransformation.tc

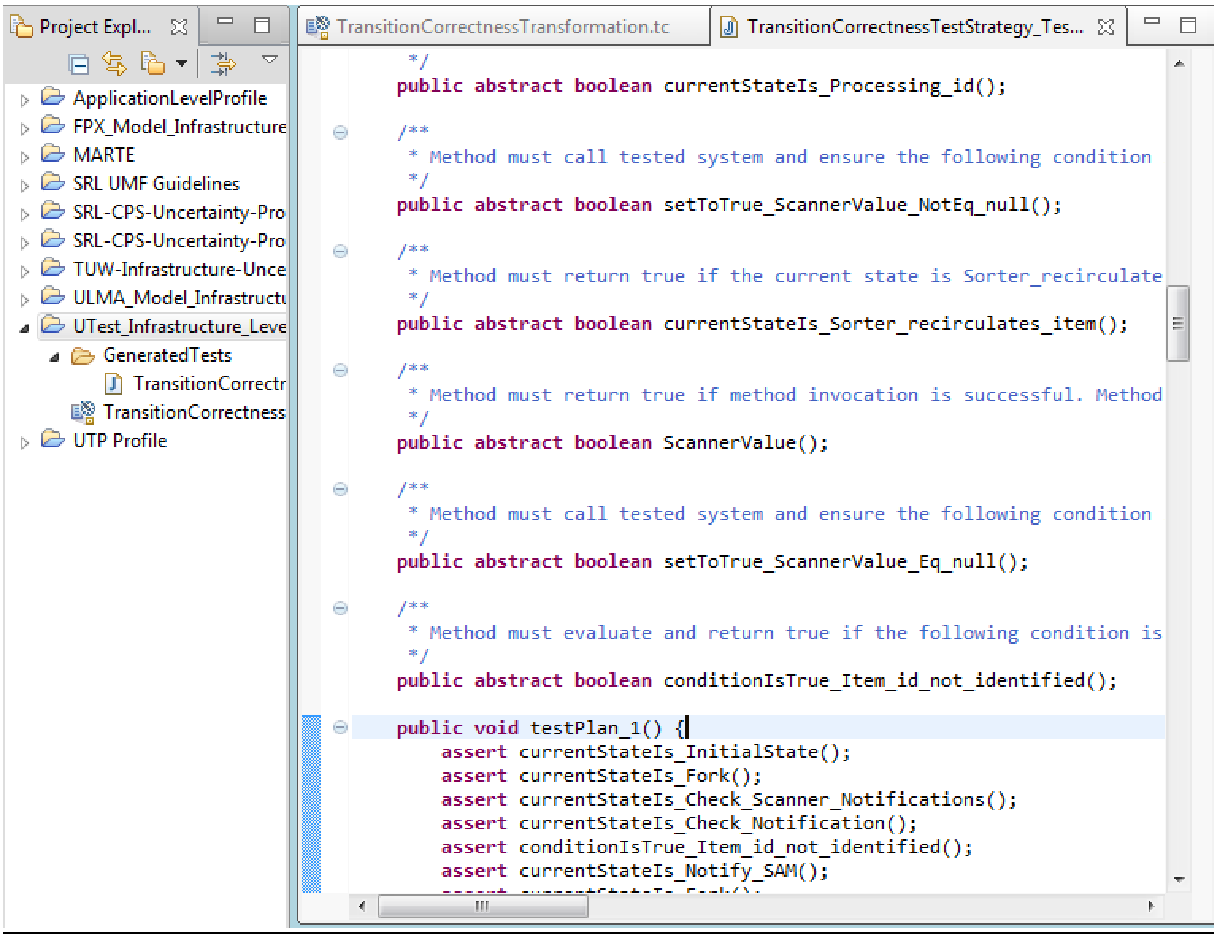
StateMachineTransformation Transformation:

Forward transformation

Name: StateMachineTransformation

Protocol: Conceptual

Figure 17: Executing the State Machine Transformation Plug-In



```
     */
    public abstract boolean currentStateIs_Processing_id();

    /**
     * Method must call tested system and ensure the following condition
     */
    public abstract boolean setToTrue_ScannerValue_NotEq_null();

    /**
     * Method must return true if the current state is Sorter_recirculate
     */
    public abstract boolean currentStateIs_Sorter_recirculates_item();

    /**
     * Method must return true if method invocation is successful. Method
     */
    public abstract boolean ScannerValue();

    /**
     * Method must call tested system and ensure the following condition
     */
    public abstract boolean setToTrue_ScannerValue_Eq_null();

    /**
     * Method must evaluate and return true if the following condition is
     */
    public abstract boolean conditionIsTrue_Item_id_not_identified();

    public void testPlan_1() {
        assert currentStateIs_InitialState();
        assert currentStateIs_Fork();
        assert currentStateIs_Check_Scanner_Notifications();
        assert currentStateIs_Check_Notification();
        assert conditionIsTrue_Item_id_not_identified();
        assert currentStateIs_Notify_SAM();
```

Figure 18: Abstract Test Plan Generated by the State Machine Transformation Plug-In

Technical Report 2 - Platform for Run-Time Testing of Cyber-Physical Systems

This technical report presents our platform for testing at run-time CPSs. All details of this TR can be found in [20].

Run-Time Testing Platform Functionality

In the following we also present here an overview over our platform, to ensure the deliverable is understandable standalone. Documentation and demos of our testing platform are available at http://tuwiendsg.github.io/RuntimeVerification/, and the code is available at https://github.com/tuwiendsg/RuntimeVerification .

Our run-time testing platform provides functionality for (Figure 19):

- Specifying the logical structure of CPSs through a JSON representation capturing their deployment stack and communication dependencies (Step 1).
- Managing the run-time structure of elastic CPSs, introducing a decentralized notification-based system for managing the addition/removal of system components (Step 2).
- Specifying run-time tests and tests execution description, introducing a domain-specific language for defining periodic and event-driven execution of direct and indirect verification tests on different system components (Step 3).
- Executing tests, introducing a distributed mechanism based on remote code execution for execution of tests and collection of test enforcement results (Step 4).
- Notifying interested parties about the test results, introducing mechanisms for notifying users about changes in the tests results (Step 5).

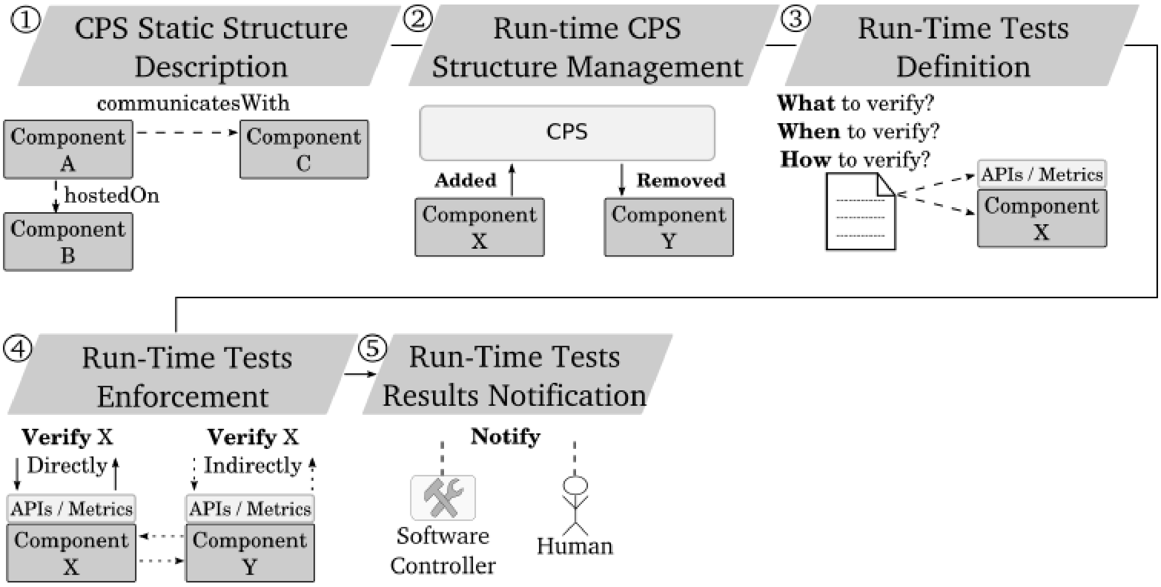


Figure 19: Functionality of Platform for Run-Time Testing of CPSs

Run-Time Testing Platform Implementation

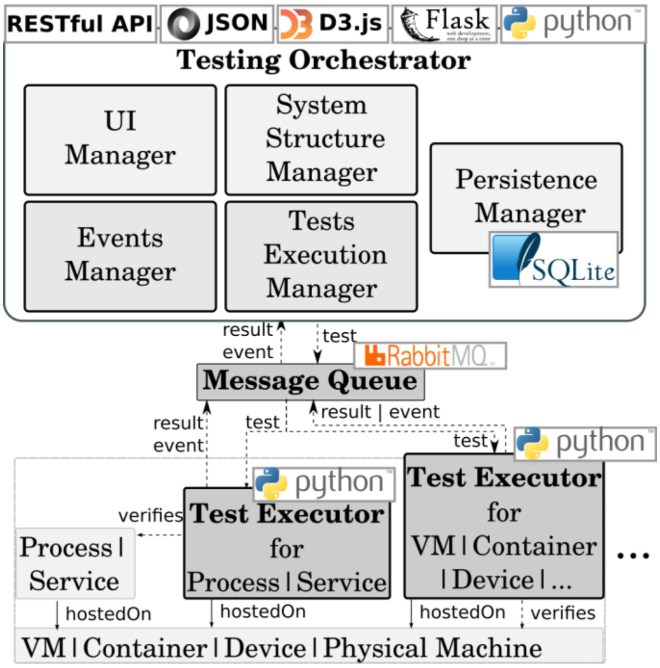


Figure 20: Platform for Run-Time Testing Prototype

We implement our run-time testing platform prototype in Python due to its low resource consumption and reduced complexity in deploying and operating the platform. Our platform has a centralized Testing Orchestrator providing most of the platform's functionality, and a Test Executor component deployed along system components to enforce tests. We expect custom test executors to be implemented for particular target systems, and provide a Messaging Queue. The queue acts as a communication broker between the Testing Orchestrator and Test Executors, hiding their particular implementation details from each other. We use RabbitMQ for the queuing middleware, as it supports both AMQP and MQTT protocols, providing a queuing solution applicable to a wide range of systems and components. The platform's functionality is divided between: (i) a System Structure Manager handling any structure-related operation; (ii) an Events Manager handling the processing of events received from the test executors due to test results or addition/removal of system component instances; (iii) a Tests Execution Manager dispatching run-time tests; (iv) a Persistence Manager using SQLite to persist system and tests information; and (v) a UI Manager handling interactions with the platform's web user interface. For ease of use and integration with third party software components, we implement the interactions with our run-time testing platform as RESTful services using Flask and JSON. We also implement a web-based interface relying on HTML, Javascript and D3.js enabling human users to interact with our platform. A run-time test is a self-contained sequence of Python code and we provide a library to report the results of particular test executions. We also provide contextualization mechanisms that injects in each python test variables denoting the ids and uuids of the test target and executor to be used in the test.

Run-Time Tests Description Language

```
#Description
#name: "TestName"
#description: "human readable description"
#timeout: 10

#Triggers
#every:  30 s
#event:  "E1" , "E2" on UnitType.VirtualMachine
#event:  "E1FFF" , "E2" on UnitType.Process

#Execution
#executor: UnitType.VirtualMachine for UnitType.VirtualMachine, UnitType.VirtualContainer, UnitType.Process
#executor: UnitType.VirtualContainer for UnitType.Process
#executor: UnitType.SoftwareContainer for UnitType.SoftwareContainer
#executor: UnitType.SoftwareContainer for UnitID."A-Za-z0-9_", UnitID."Process.ProcessNAME", UnitUUID."A-Za-z0-9_."
#executor: UnitID."A-Za-z0-9_" for UnitID."Process.ProcessNAME", UnitUUID."A-Za-z0-9_."
```

Listing 6: Run-Time Tests Description Language

We introduce a domain specific language and format for specifying what/when/how to test for a run-time test. The language literals are explained in Table 3, and its keywords in Table 4. The test description uses the UnitType indicator

Table 3: Run-Time Tests Description Language: Literals

Literal	Description
UnitType	Represents a component type according to CPS infrastructure profile captured in D2.1
ID	Represents a custom component ID used to identify a component (e.g., a system component) in the system's design-time structure
UUID	Represents the unique ID of a deployed system component instance. An elastic system component (e.g., web server) can have multiple instances running at one time.
Event	Represents a custom defined system event identified by its name or ID (e.g., scale-out)

Table 4: Run-Time Tests Description Language: Keywords

Keyword	Description
Description	Marks the test description section

name	Marks the name of the test to be executed
description	Human-readable description of the test to be executed
timeout	Defines a timeout in which test result must be received before considering the test failed
Triggers	Marks the test triggers section defining when the test is executed
event	Specifies that the test should be executed when certain events are encountered
on	Used to specify on which system component the event must be detected to trigger test execution
every	Used to specify periodical test execution
Execution	Marks the section describing what component executes the test
executor	Defines for which components the test is executed, and which components will execute it
for	Used to define what component executes a test defined for the same

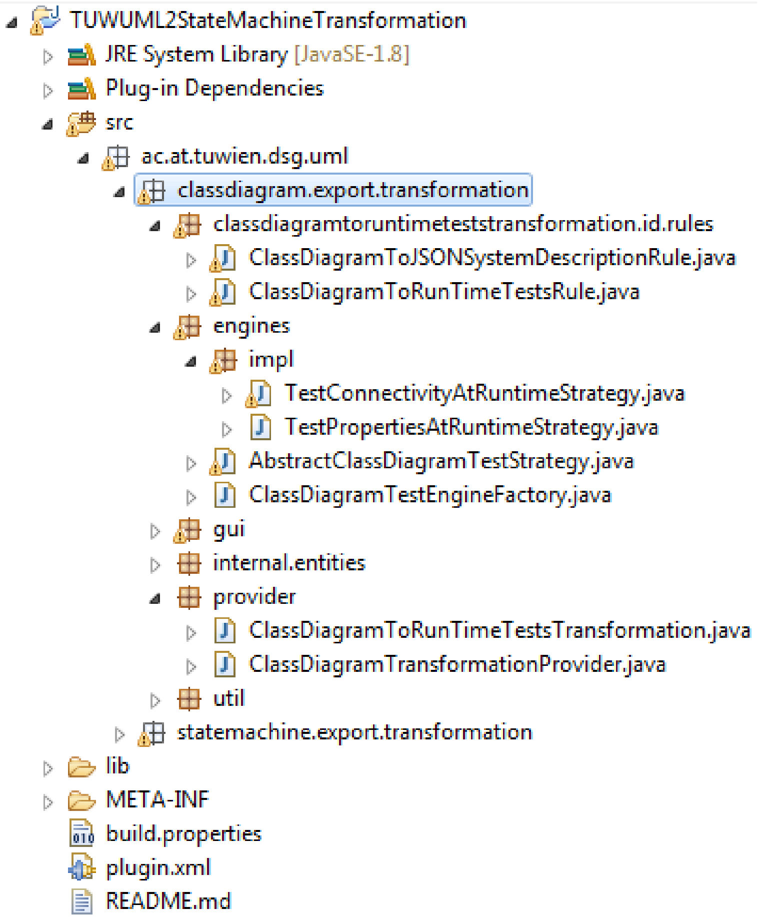
Technical Report 3 - Integrating Run-Time Testing Strategies with our Platform for Run-Time Testing of Cyber-Physical Systems

In this report we discuss how we apply and integrate our strategies for generating descriptions for run-time tests described in Section 4.3.1.2 "Run-Time Testing" with our Run-Time Testing Platform presented in the previous Technical Report 2 .

As for the previous testing strategies, we implement the run-time testing strategies as an IBM RSA plug-in. The plug-in takes as input this time a UML Class Diagram defined in IBM RSA in a UML Profile Based on the UML Class the plug-in first generates the CPS description in the format accepted by our Run-time Testing Platform. Further, for each class captured in the class diagram, the plug-in will generate corresponding run-time tests descriptions, according to the testing strategy employed (from the two described in Section 4.3.1.2).

The Plug-In code is located in an open-source GitHub code repository at:

https://github.com/tuwiendsg/COMOT4U/tree/master/T4U/UncertaintyTestsGeneration/TUWUML2StateMachineTransformation. The particular classes implementing the plug-in are located in the "src/ac/at/tuwien/dsg/uml/ classdiagram/export/transformation/" subdirectory.



```
◢ TUWUML2StateMachineTransformation
    ▷ JRE System Library [JavaSE-1.8]
    ▷ Plug-in Dependencies
    ◢ src
        ◢ ac.at.tuwien.dsg.uml
            ◢ classdiagram.export.transformation
                ◢ classdiagramtoruntimeteststransformation.id.rules
                    ▷ ClassDiagramToJSONSystemDescriptionRule.java
                    ▷ ClassDiagramToRunTimeTestsRule.java
                ◢ engines
                    ◢ impl
                        ▷ TestConnectivityAtRuntimeStrategy.java
                        ▷ TestPropertiesAtRuntimeStrategy.java
                    ▷ AbstractClassDiagramTestStrategy.java
                    ▷ ClassDiagramTestEngineFactory.java
                ▷ gui
                ▷ internal.entities
                ◢ provider
                    ▷ ClassDiagramToRunTimeTestsTransformation.java
                    ▷ ClassDiagramTransformationProvider.java
                ▷ util
            ▷ statemachine.export.transformation
    ▷ lib
    ▷ META-INF
      build.properties
      plugin.xml
      README.md
```

Figure 21: Run-Time Testing Strategy IBM RSA Plug-In

We implement our run-time testing strategies in Java as a specific type of plug-ins called "Transformation Extensions" (code depicted as Eclipse project in Figure 13). We have named our Transformation plug-in "ClassDiagramToRunTimeTestsTransformation", belonging to a plug-ins group called "ac.at.tuwien.dsg.uml.classdiagram.export.transformation".

The ClassDiagramToRunTimeTestsTransformation is applicable only to UML Class or UML Package types. The plug-in extends the "com.ibm.xtools.transform.core.RootTransform", enabling it to access UML class diagram information relying on Eclipse UML2 API from package "org.eclipse.uml2.uml." Based on the Eclipse UML2 API we implement the test plan generation algorithm from Section 4.3.1.2.2 in two classes: class TestPropertiesAtRuntimeStrategy implements run-time tests generation for each UML Class attributes, while class TestConnectivityAtRuntimeStrategy generates run-time connectivity tests for all UML Class associations.

The glue between the run-time tests description generation classes and the UML Class Diagram processing is implemented in ClassDiagramToRunTimeTestsRule class. ClassDiagramToRunTimeTestsRuletakes as input an UML Class Diagram represented as a org.eclipse.uml2.uml.ClassImpl object. Further, it calls the testing strategy in use to generate based on the UML Class run-time tests descriptions. The test plan is generated relying on particular configuration templates used by our Run-Time Testing Platform. By replacing the content of the used configuration files one can extend our approach to generate test descriptions for other testing platforms. Thus, our plug-in generates only the descriptions needed to test CPSs at run-time. In order to execute the tests, for each description a test implementation for a particular CPPS must be provided and submitted to the Run-Time Testing Platform.

Using the Run-Time Testing Correctness IBM RSA Plug-In

To use the plug-in please follow the following steps:

Install:

- Import the plugin project in your RSA run-time and deploy it to local host (or install plug-in Jar from deployment).
- Or just run as Eclipse Application the plug-in project (no install required, local run)

Use:

- Create a new Transformation Configuration: "File -> New -> Other -> Transformations -> Transformation Configuration"
- Choose transformation type: Check "show all transformations".
- Choose "ac.at.tuwien.dsg.uml.classdiagram.export.transformation" -> "ClassDiagramToRunTimeTestsTransformation" (Figure 22).
- From transformation "Source and Target" tab select source state machine diagram and target location where transformation file should be generated (Figure 23).
- From "Select Test Generation Strategy" tab select the test generation strategy to use when generating the test plan (Figure 24).
- Run the transformation.
- Right click on Target project and Refresh.
- The system description will be generated in a file having the name of the UML Package containing the UML Classes (Figure 25).
- The generated tests will be structured in one folder per UML Class, having names Test_[attribute_property_name] (Figure 26).
- Use the generated system description and tests description in conjunction with the Run-Time Testing platform described in Technical Report 2 .

Extend:

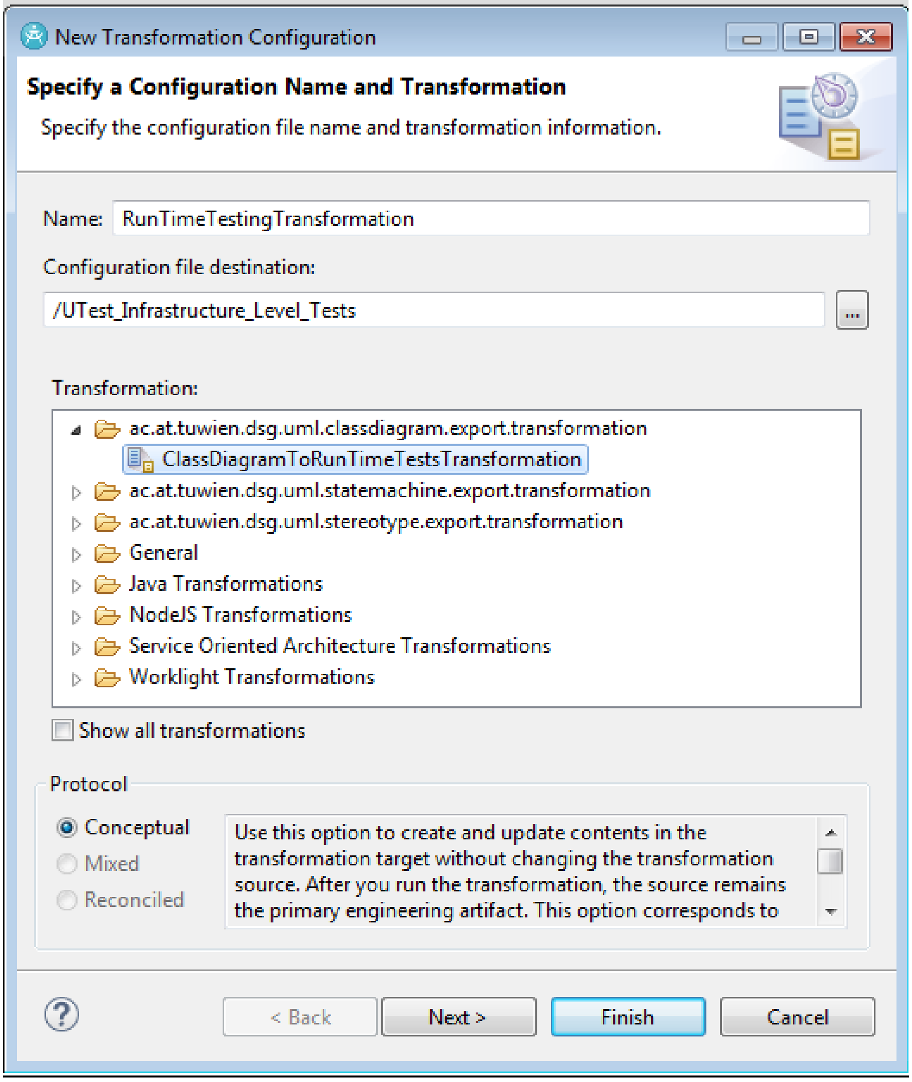
To extend our plug-in with other test generation strategies one must consider the most important Java classes we provide in our implementation:

1. Most important classes for extending the plug-in are:

- `AbstractClassDiagramTestStrategy`: class, which has the abstract "`generateTestConfig`" method. The class must be sub-classed and the method implemented for any custom strategy.
- `ClassDiagramTestEngineFactory`: factory class holding all supported strategies. Any new subclass of `AbstractClassDiagramTestStrategy` must be added in "supportedStrategies". After this they will appear on the selection tab and be called when required.

Usage example screenshots:

In the following screenshots we exemplify the usage of our Class Diagram to Run-Time Tests Transformation Plug-In in IBM RSA. We apply our plug-in to ULMA Infrastructure profile described in D2.1 Section 5.3.2.



New Transformation Configuration

Specify a Configuration Name and Transformation

Specify the configuration file name and transformation information.

Name: RunTimeTestingTransformation

Configuration file destination: /UTest_Infrastructure_Level_Tests

Transformation:

- ac.at.tuwien.dsg.uml.classdiagram.export.transformation
  - ClassDiagramToRunTimeTestsTransformation
- ac.at.tuwien.dsg.uml.statemachine.export.transformation
- ac.at.tuwien.dsg.uml.stereotype.export.transformation
- General
- Java Transformations
- NodeJS Transformations
- Service Oriented Architecture Transformations
- Worklight Transformations

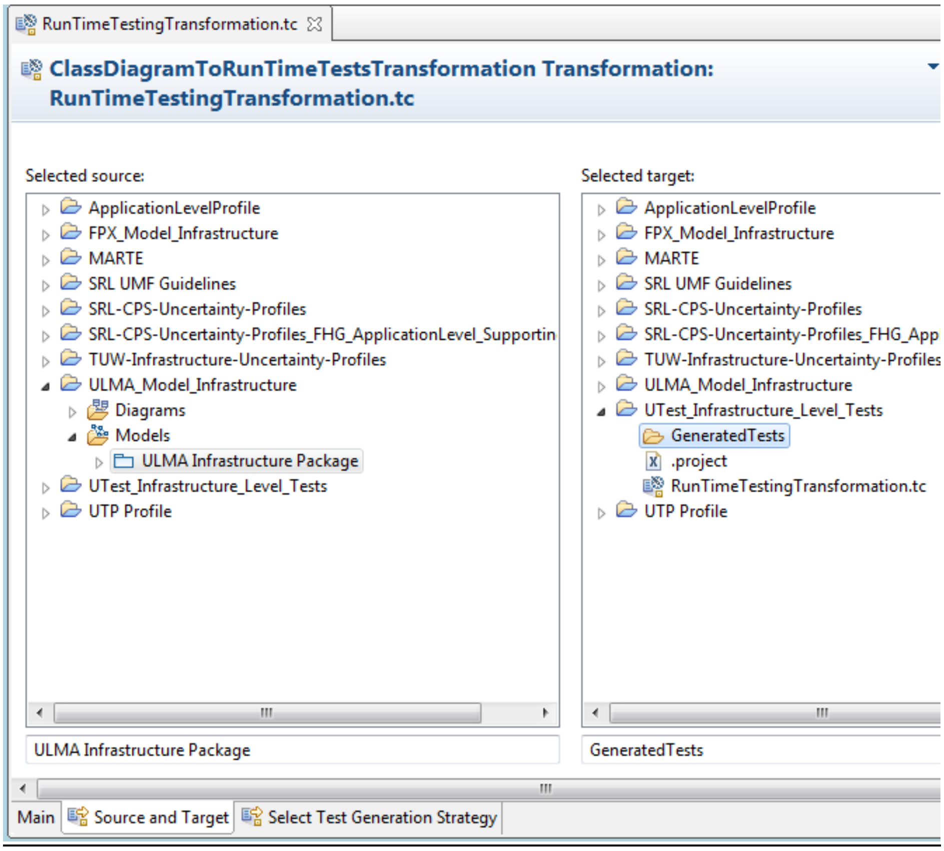
□ Show all transformations

Protocol

● Conceptual
○ Mixed
○ Reconciled

Use this option to create and update contents in the transformation target without changing the transformation source. After you run the transformation, the source remains the primary engineering artifact. This option corresponds to

Figure 22: Creating a new Class Diagram to Run-Time Tests Transformation Configuration



RunTimeTestingTransformation.tc

ClassDiagramToRunTimeTestsTransformation Transformation: RunTimeTestingTransformation.tc

Selected source:

- ApplicationLevelProfile
- FPX_Model_Infrastructure
- MARTE
- SRL UMF Guidelines
- SRL-CPS-Uncertainty-Profiles
- SRL-CPS-Uncertainty-Profiles_FHG_ApplicationLevel_Supportin
- TUW-Infrastructure-Uncertainty-Profiles
- ULMA_Model_Infrastructure
  - Diagrams
  - Models
    - ULMA Infrastructure Package
- UTest_Infrastructure_Level_Tests
- UTP Profile

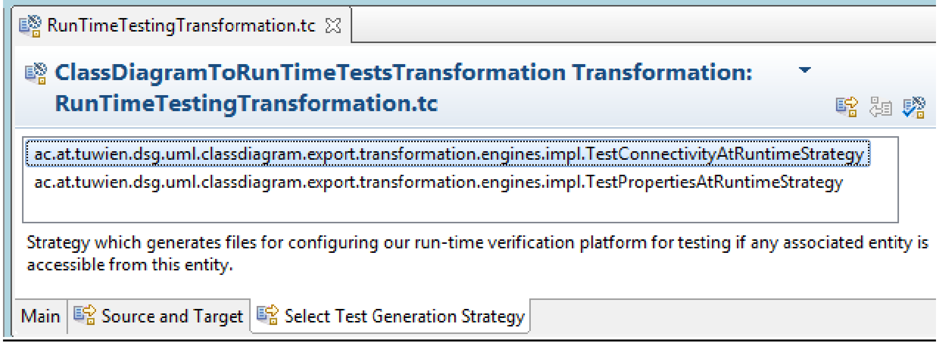
ULMA Infrastructure Package

Selected target:

- ApplicationLevelProfile
- FPX_Model_Infrastructure
- MARTE
- SRL UMF Guidelines
- SRL-CPS-Uncertainty-Profiles
- SRL-CPS-Uncertainty-Profiles_FHG_App
- TUW-Infrastructure-Uncertainty-Profiles
- ULMA_Model_Infrastructure
- UTest_Infrastructure_Level_Tests
  - GeneratedTests
  - .project
  - RunTimeTestingTransformation.tc
- UTP Profile

GeneratedTests

Figure 23: Choosing the UML Package containing the UML Classes used in Test Generation



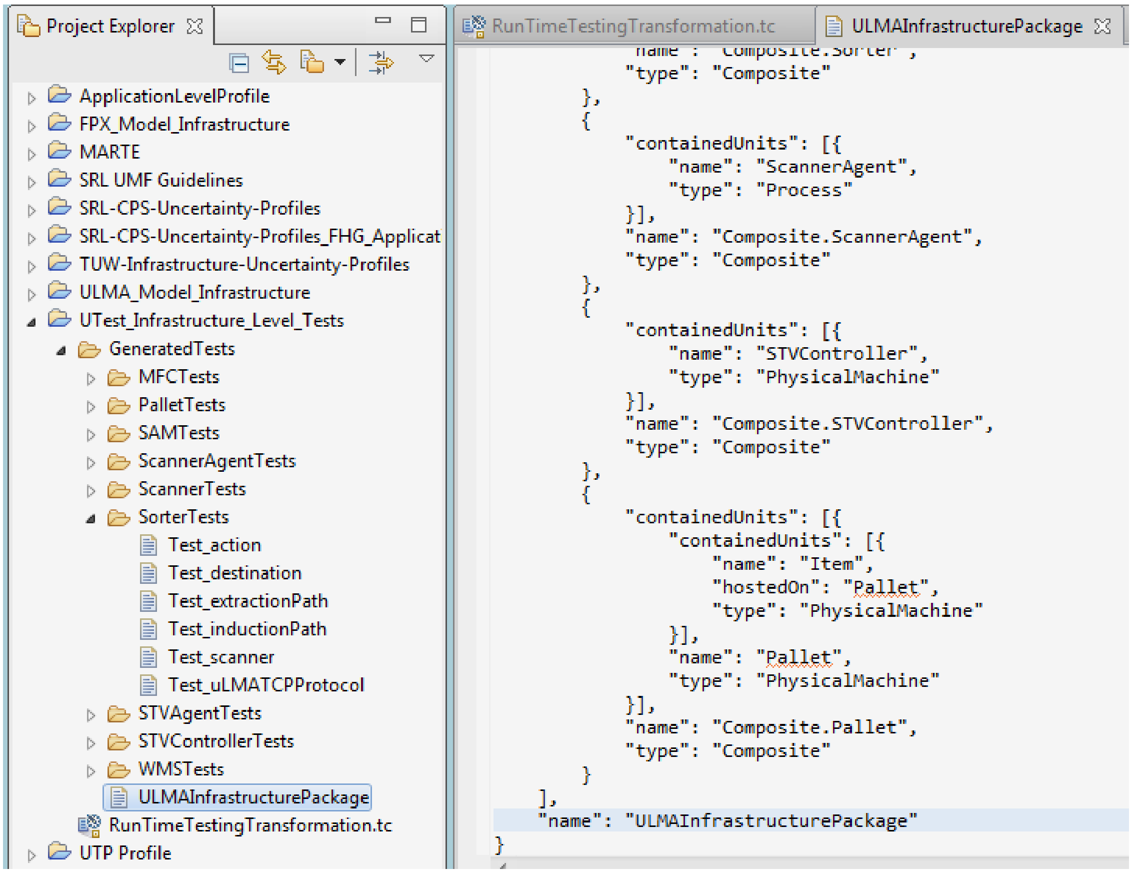
RunTimeTestingTransformation.tc

ClassDiagramToRunTimeTestsTransformation Transformation: RunTimeTestingTransformation.tc

ac.at.tuwien.dsg.uml.classdiagram.export.transformation.engines.impl.TestConnectivityAtRuntimeStrategy
ac.at.tuwien.dsg.uml.classdiagram.export.transformation.engines.impl.TestPropertiesAtRuntimeStrategy

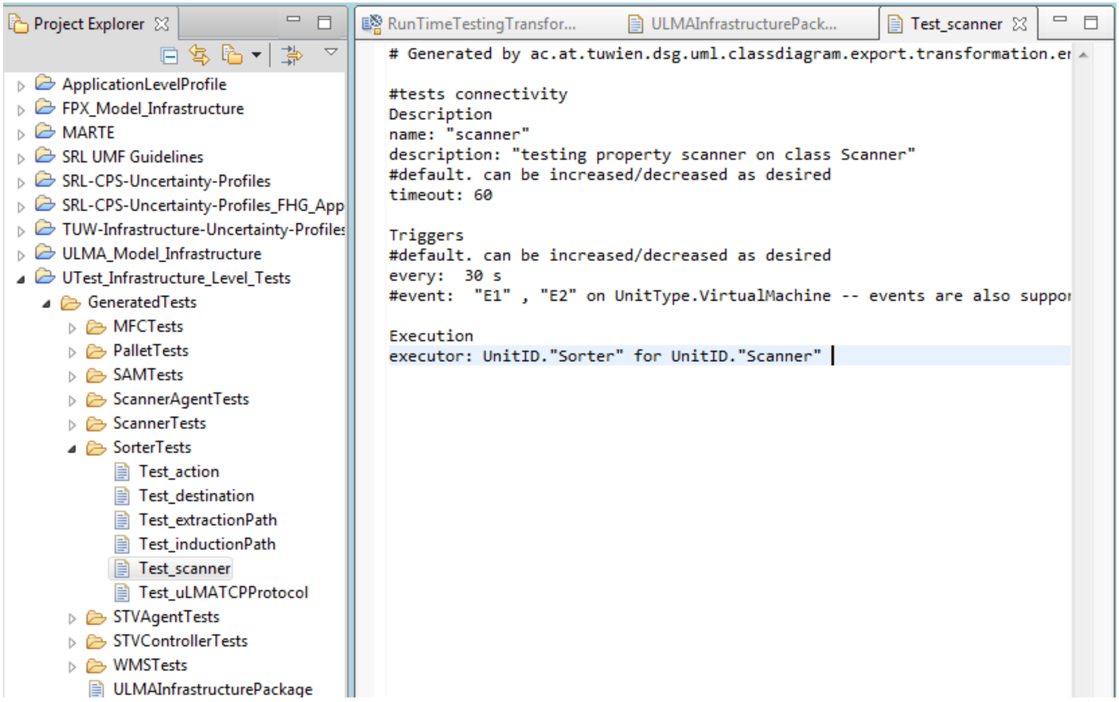
Strategy which generates files for configuring our run-time verification platform for testing if any associated entity is accessible from this entity.

Figure 24: Choosing the Run-Time Tests Generation Strategy



```
            "name": "Composite.Sorter",
            "type": "Composite"
        },
        {
            "containedUnits": [{
                "name": "ScannerAgent",
                "type": "Process"
            }],
            "name": "Composite.ScannerAgent",
            "type": "Composite"
        },
        {
            "containedUnits": [{
                "name": "STVController",
                "type": "PhysicalMachine"
            }],
            "name": "Composite.STVController",
            "type": "Composite"
        },
        {
            "containedUnits": [{
                "containedUnits": [{
                    "name": "Item",
                    "hostedOn": "Pallet",
                    "type": "PhysicalMachine"
                }],
                "name": "Pallet",
                "type": "PhysicalMachine"
            }],
            "name": "Composite.Pallet",
            "type": "Composite"
        }
    ],
    "name": "ULMAInfrastructurePackage"
}
```

Figure 25: Generated JSON CPS Description for Run-Time Testing Platform



```
# Generated by ac.at.tuwien.dsg.uml.classdiagram.export.transformation.er

#tests connectivity
Description
name: "scanner"
description: "testing property scanner on class Scanner"
#default. can be increased/decreased as desired
timeout: 60

Triggers
#default. can be increased/decreased as desired
every:  30 s
#event:  "E1" , "E2" on UnitType.VirtualMachine -- events are also suppor

Execution
executor: UnitID."Sorter" for UnitID."Scanner"
```

Figure 26: Generated Run-Time Test Description for Testing Connectivity between Sorter and Scanner

Technical Report 4 - Interactively Evolving Test Ready Models with Uncertainty Developed for Testing Cyber-Physical Systems

Please see TR4.pdf attached with this deliverable as a separate document. The TR is also available online at [30].

Technical Report 5 - Uncertainty-based Test Case Generation and Minimization for Cyber-Physical Systems: A Multi-Objective Search-based Approach

Please see TR5.pdf attached with this deliverable as a separate document. The TR is also available online at [29].

Bibliography

[1] *U-Test Deliverable Report on Requirements Collection.*

[2] *U-Test Deliverable Report on Taxonomy D 1. 2.*

[3] *U-Test Deliverable Report on Uncertainty Modelling Framework V1*

[4] Amland, S., *Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study.* Journal of Systems and Software, 2000. **53**(3): p. 287-295. Available from: http://www.sciencedirect.com/science/article/pii/S0164121200000194.

[5] Arcuri, A. and L. Briand. *A practical guide for using statistical tests to assess randomized algorithms in software engineering*. in *2011 33rd International Conference on Software Engineering (ICSE)*. 2011. IEEE.

[6] Bagnato, A., A. Sadovykh, E. Brosse, and T.E.J. Vos. *The OMG UML Testing Profile in Use--An Industrial Case Study for the Future Internet Testing*. 2013. IEEE.

[7] Baker, P., Z.R. Dai, J. Grabowski, I. Schieferdecker, and C. Williams, *Model-driven testing: Using the UML testing profile*. 2007: Springer Science & Business Media.

[8] DeMillo, R.A., R.J. Lipton, and F.G. Sayward, *Hints on Test Data Selection: Help for the Practicing Programmer.* Computer, 1978. **11**(4): p. 34-41. Available from: 10.1109/C-M.1978.218136.

[9] Fabbri, S.C.P.F., M.E. Delamaro, J.C. Maldonado, and P.C. Masiero. *Mutation analysis testing for finite state machines*. in *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*. 1994. IEEE.

[10] Felderer, M. and I. Schieferdecker, *A taxonomy of risk-based testing.* International Journal on Software Tools for Technology Transfer, 2014. **16**(5): p. 559-568.

[11] Gérard, S. and B. Selic, *The UML–MARTE Standardized Profile.* IFAC Proceedings Volumes, 2008. **41**(2): p. 6909-6913.

[12] Harman, M. and B.F. Jones, *Search-based software engineering.* Information and software Technology, 2001. **43**(14): p. 833-839.

[13] Harman, M., S.A. Mansouri, and Y. Zhang, *Search-based software engineering: Trends, techniques and applications.* ACM Computing Surveys (CSUR), 2012. **45**(1): p. 11.

[14] Hessel, A., K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, *Testing real-time systems using UPPAAL*, in *Formal methods and testing*. 2008, Springer. p. 77-117.

[15] Johansson, W., M. Svensson, U.E. Larson, M. Almgren, and V. Gulisano. *T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols*. in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 2014. IEEE.

[16] Larsen, K.G., M. Mikucionis, and B. Nielsen. *Online testing of real-time systems using uppaal*. in *International Workshop on Formal Approaches to Software Testing*. 2004. Springer.

[17] Larsen, K.G., 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*. 2005. ACM.

[18] Li, J.-h., G.-x. Dai, and H.-h. Li. *Mutation analysis for testing finite state machines*. in *Electronic Commerce and Security, 2009. ISECS'09. Second International Symposium on*. 2009. IEEE.

[19] Microsoft. *Spec Explorer*. [cited 2016 September 25th]; Available from: https://msdn.microsoft.com/en-us/library/ee620411.aspx.

[20] Moldovan, D. and H.-L. Truong. *A Platform for Run-time Health Verification of Elastic Cyber-physical Systems*. in *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. 2016. London, UK

[21] Mussa, M., S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj. *A survey of model-driven testing techniques*. 2009. IEEE.

[22] Naveh, B. *JGrapht*. 2016 [cited 2016; Available from: http://jgrapht.org/.

[23] Offutt, A. *The coupling effect: fact or fiction*. in *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*. 1989. ACM.

[24] Schneider, M., J. Großmann, I. Schieferdecker, and A. Pietschker. *Online model-based behavioral fuzzing*. in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. 2013. IEEE.

[25] Schulze, C., D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, *Assessing model-based testing: an empirical study conducted in industry*, in *Companion Proceedings of the 36th International Conference on Software Engineering*. 2014, ACM: Hyderabad, India. p. 135-144.

[26] Stefanescu, A., M.-F. Wendland, and S. Wieczorek. *Using the UML testing profile for enterprise service choreographies*. 2010. IEEE.

[27] Utting, M., B. Legeard, F. Bouquet, E. Fourneret, F. Peureux, and A. Vernotte, *Chapter Two-Recent Advances in Model-Based Testing.* Advances in Computers, 2016. **101**: p. 53-120.

[28] Utting, M., A. Pretschner, and B. Legeard, *A taxonomy of model-based testing approaches.* Software Testing, Verification and Reliability, 2012. **22**(5): p. 297-312.

[29] Zhang, M., S. Ali, T. Yue, and M. Hedman, *Uncertainty-based Test Case Generation and Minimization for Cyber-Physical Systems: A Multi-Objective Search-based Approach*. Technical Report, 2016. Simula Research Laboratory, Technical Report 2016-13. Available from: https://www.simula.no/file/utestingtrpdf/download.

[30] Zhang, M., S. Ali, T. Yue, and R. Norgren, *Interactively Evolving Test Ready Models with Uncertainty Developed for Testing Cyber-Physical Systems*. Technical Report, 2016. Simula Research Laboratory, Technical Report 2016-12. Available from: https://www.simula.no/file/ist-u-evolvesubmittedtrpdf/download.