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

Report on Uncertainty Testing Framework V.2
D 3. 2

Project Acronym	U-TEST	**Grant Agreement Number**		H2020-ICT-2014-1. 645463	
Document Version	1.0	**Date**	2017-05-05	**Deliverable No.**	3.2
Contact Person	Martin Schneider	**Organisation**		Fraunhofer FOKUS	
Phone	+49 30 3463 7383	**E-Mail**		martin.schneider@fokus.fraunhofer.de	

Document Version History

Version No.	Date	Change	Author(s)
0.1	2017-01-24	Initial document outline	FF
0.2	2017-04-24	Contribution from SRL	SRL
0.3	2017-04-25	Contributions from TUW, FF	TUW, FF
0.9	2017-04-26	Integrating all contributions for review	FF
0.9.1	2017-05-02	Revision based on review comments	FF
1.0	2017-05-05	Revision based on review comments	SRL, TUW

Contributors

Name	Partner	Part Affected	Date
Martin Schneider	FF	Sections 1, 2.1, 2.2, 3.1	2017-04-25
Man Zhang	SRL	Sections 2.3, 3.3, TR4.1.pdf, TR7.pdf	2017-04-24
Shaukat Ali	SRL	Sections 2.3, 3.3, TR4.1.pdf, TR7.pdf	2017-04-24
Tao Yue	SRL	Sections 2.3, 3.3, TR4.1.pdf, TR7.pdf	2017-04-24
Ivan Pavkovic	TUW	Sections 2.3,3.2, TR1	2017-04-25
Luca Berardinelli	TUW	Sections 2.3,3.2, TR1	2017-04-25
Hong-Linh Truong	TUW	Sections 2.3,3.2, TR1	2017-04-25

Reviewers

Name	Partner	Part Affected	Date
Robert Magnusson	NMT	All	2017-05-02
Karmele Intxausti	IKL	All	2017-04-28
Fabien Peureux	EGM	All	2017-05-01

Table of Contents

Executive Summary

This deliverable presents the Uncertainty Testing Framework (UTF) V.2 with model evolution algorithms and test strategies. It extends the works from V.1 of the Uncertainty Testing Framework and extends its focus to discovery of unknown uncertainties in addition to coverage of known uncertainties. Model evolution has been developed for uncertainty testing of Cyber-Physical Systems (CPS) at the three levels (application, infrastructure, and integration) of CPS. More specifically, the UTF takes the test-ready models specified with the Uncertainty Modelling Framework (UMF) as input, and (automatically) produces abstract test cases and executable test cases as output. This deliverable shows that we have successfully achieved Milestone 4 with the UTF V.2 for uncertainty testing at the three levels of CPS. The UTF V.2 provides a concrete foundation for achieving Milestone 5, in which we finalize the testing framework and apply it exhaustively to the pilot cases.

1 Introduction

This report describes the second version of the UTF. We are still making possible improvement on the UTF. The further iteration and refinements of UTF will be included in the next U-Test reports.

1.1 Objectives of the Deliverable

The goal of this deliverable is to present improvements of the UTF that we have developed since its version developed at M3. Our UTF supports for testing uncertainties and uncertain behaviours of Cyber-Physical Systems (CPS) at three levels: application, infrastructure, and integration. We developed and integrated different model evolution algorithms and testing strategies in the UTF. These model evolution algorithms cover different part of the problem to efficiently test cyber-physical systems for known and unknown uncertainties. All the model evolution algorithms and test 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) [1] and Uncertainty Modelling Framework (UMF) [2]. We used U-Taxonomy and UMF for specifying and modelling different uncertainties of CPS, at three levels, i.e., application, infrastructure, and integration. In this deliverable, we show how our UTF (V.2) is based 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 behaviours. The searching is based on known uncertainty behaviours at the three levels of CPS. Technical reports presenting details of approaches are completing this deliverable.

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),
- the U-Taxonomy (D1.2), and
- the UMF (D2.2)

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) and 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 modelled 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 (D4.3), Report on test case executions (D5.2), Dissemination (D6.4), and Exploitation (D7.3).

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.

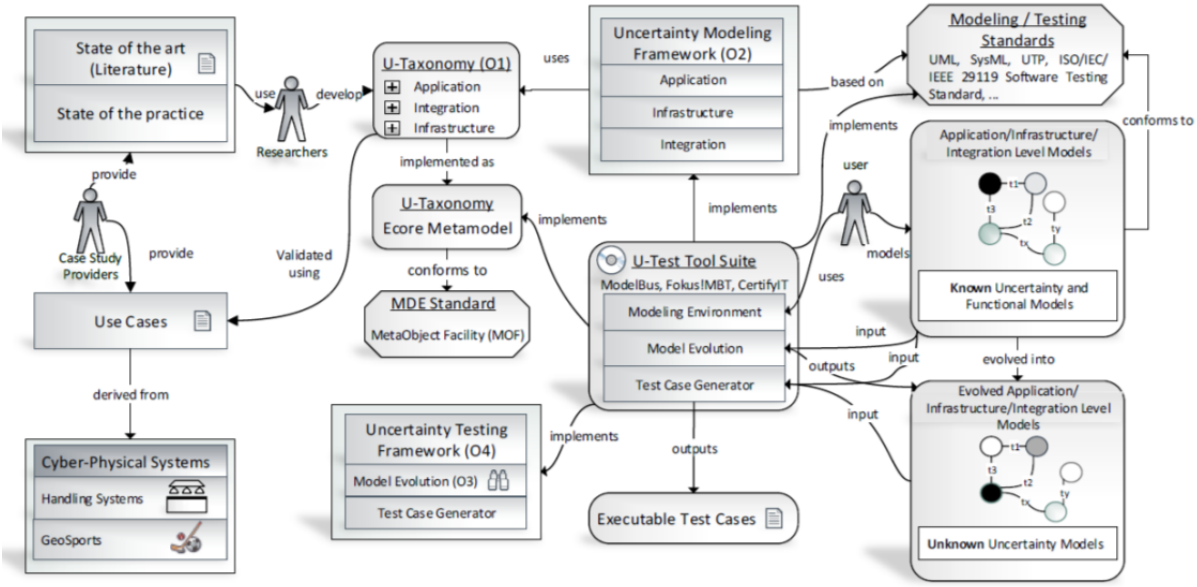

Figure 1. U-Test Workflow

Input:

- All previous Deliverables

Consumers of D3.2 (that are currently active)

- D4.3 (EGM, FF): Tool(s) Demonstrator
- D5.2 (FPX and ULMA): Report on test case executions
- D5.3 Validation with or without U-Test
- D6.3 Dissemination
- D7.2 (For Exploitation): Value Opportunities

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.

The remainder of this deliverable is organized as follows. An overview of the UTF is given in Section 2.1. UTF, which supports uncertainty testing at the application level, infrastructure level, and integration level of CPS, is presented in Sections 2.2, 2.3, and 2.4 correspondingly. Aiming at the comprehensiveness of this document, for presenting technical details on some specific topics, we organized them into technical reports (TRs). TR1, which provides more technical details for Section 2.3, are included in the Appendix. TR4.1 and TR7, which provide more technical details for Section 2.4, are in forms of two separate PDF files attached with this document. We summarize the whole deliverable and give our conclusions in Section 3.

2 Uncertainty Testing Framework

Section 2.1 gives an overview of the UTF. Next, Section 2.2 presents the details of UTF for supporting uncertainty testing at the application level of CPS. Similarly, Sections 2.3 and 2.4 present the details of UTF for supporting uncertainty testing at the infrastructure and integration levels of CPS correspondingly.

2.1 Overview of Uncertainty Testing Framework

Figure 2 shows a high-level overview of the UTF with its input and output. The main input of UTF are the test-ready models that we have created by using the UMF.

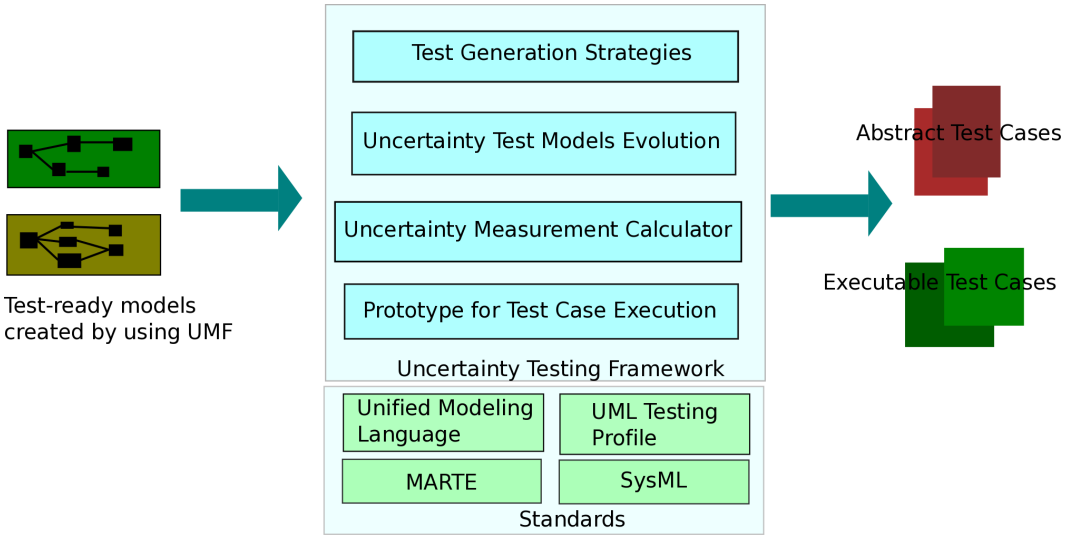

Figure 2. An Overview of Uncertainty Testing Framework

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 cases 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. Eventually, executable test cases are generated by the Uncertainty Testing Framework. The details of UTF for supporting uncertainty testing of CPS at the three different levels are presented in the following sections.

2.2 Uncertainty Testing at Application Level

This section describes the second version of the Uncertainty Testing Framework aiming at discovering uncertainties at the application level.

2.2.1 Uncertainty Model Evolution at Application Level

As indicated in D3.1, mutations to state machines are the atomic piece of information we deal with. Therefore, the evolution of state machines is focused on introducing mutations into state machines. These state machines describe valid interaction with the system under test. The problem we would like to solve is to find those sets of mutations to a state machine that reveal (the most unknown) uncertain behaviours.

Therefore, we consider the set of mutations associated with a state machine as an individual. One or more mutations are representing uncertainty in the environment of the CPS that may lead to an uncertain behaviour. Hence, as described in D3.1, the starting point for model evolution are state machines that describe the expected interaction of the environment of the CPS. Evolution is done by introducing mutations to this state machine. We implement the evolution of state machines by employing a genetic algorithm. The basic idea is to use uncertainties, modelled in test-ready models, to guide the mutation of state machines. Thus, to reduce the size of the search space whilst enabling to cover the different scenarios in which an uncertainty may occur.

In D3.1, an initial set of mutation operators were described that cover mainly mutations on transitions itself. Based on the literature, we extended this set of mutations as described in the following table:

Table 1. Mutation Operators (adapted from [3])

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. Equivalent to 'Change Guard: replace expression with false'.
Remove Transition (with State Merge)	Completely removes the transition. Merges the source and target state if the removed transition is the only one connecting them (*optional:* with the same direction). This avoid mutilated state machines which inhibit generating test cases.	Transitions having an initial state as source or a final node as target must not be removed. Equivalent to 'Change Guard: replace expression with false'.
Reverse Transition	Swaps source and target of the transition.	Transitions having an initial state as source or a final node as target must not be reversed. *Optional:* Transitions being the only one that connect source and target state must not be removed (*optional:* with the same direction). This avoid mutilated state machines which inhibit generating test cases.
Change Source/Target	Move the source or the 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.

Mutation Operator	Description	Constraints/Comments
		In case the source state of the transition is changed, the source must not be the final node.
Remove Trigger	Transforms the transition to a completion transition.	
Remove Guard	Removes the guard of a transition completely.	Equivalent to 'Change Guard: replace expression with true'
Remove Effect	Removes the effect of a transition completely.	
Change Trigger Operation	Changes the operation to another one of the same interface of the original operation.	
Change Guard/ Change Effect	- replace expression with true/false - negate expression - replace subexpression with true/false - negate subexpression - change logical operator - change relational operator - change arithmetic operator - change set operator - change quantifier - replace operand guard/effect mutation operators - remove statement - move statement - fix parameter/property of a called method or sent signal - change called method or sent signal - change operator - fix operand (replace it with a literal) - change operand (replace with variable, call parameter or signal property of the same type) - replace result: replace right-hand-side (RHS) expression with default value of left-hand-side (LHS)	Guards and effects are written in C#.

These mutation operators are employed to introduce mutations based on modelled uncertainties. Depending on the values of the properties of a modelled uncertainty, different mutation operators are applied to elements of the UML state machine. The following paragraphs discuss which mutation

operators and elements of a state machine are selected based on different properties of an uncertainty. Eventually, a set of mutation operator is aggregated based on the different properties to apply different kind of mutations to a state machine.

Concept of Uncertainty Taxonomy	UncertaintyNature::**Epistemic**
Mutation Operators	Depends on the other properties of the uncertainty
Selection Criterion	Depends on the other properties of the uncertainty

We perform a systematic mutation of selected elements supported according to the other properties of the uncertainty.

Concept of Uncertainty Taxonomy	UncertaintyNature::**Aleatoric**
Mutation Operators	any
Selection Criterion	Depends on the other properties of the uncertainty

Aleatoric uncertainties are those where we are not aware of any systematics. Therefore, mutation is completely random while the elements to be mutated depend on the other properties of uncertainty. This will override the selection of mutation operators based on other properties of an uncertainty.

Concept of Uncertainty Taxonomy	Location::**InputParameters**
Mutation Operators	Change Guard
Selection Criterion	Guards of transitions whose trigger operation has an *in* parameter referred by **InputParameters**

Uncertainty w.r.t. the *in* parameters of an operation called by trigger. Changing a guard will eventually result in changed input parameters, i.e. stimuli, when test cases are generated from the mutated state machine.

Concept of Uncertainty Taxonomy	TechnologicalProcess::**TimingIssues**
Mutation Operators	Change Trigger
Selection Criterion	Triggers that have a TimeEvent and a corresponding TimeExpression

By changing the time expression, uncertain behaviour that may result due to unexpected timing may be observed. The change of the trigger leads to generation of test cases reflecting the changed time expression.

Concept of Uncertainty Taxonomy	TechnologicalProcess::ProtocolIssues::**InteroperabilityIssue**
Mutation Operators	Change Guard Change Target of Transition
Selection Criteria	Guards of transitions whose trigger operation has an *in* parameter referred by **InteroperabilityIssue**
	Transitions referred by this **InteroperabilityIssue**

Interoperability issues may arise from ambiguous or misinterpretation of requirements and specifications. By changing guards and input parameters as well as transitions, corresponding mutations resulting in different (behaviour originating from different interpretations) is achieved.

Concept of Uncertainty Taxonomy	TechnologicalProcess::ProtocolIssues::**FaultyProtocolImplementation**
Mutation Operators	Any

Selection Criterion	Elements referred by this **FaultyProtocolImplementation** that are related to a computer interface (i.e. the cyber environment)

Mutations will result in any modifications related to the digital interfaces, i.e. would reflect uncertainty from the cyber environment related to protocols.

Concept of Uncertainty Taxonomy	TechnologicalProcess::**ResourceIssues**
Mutation Operators	Change Effect Change Transition Source Change Transition Target
Selection Criteria	Transitions whose effect has an expression in which a resource is referred by **ResourceCompetition**
	Elements that store resources based on the assignments in an effect to a property of the state machine that is referred by **ResourceLocation**
	Resources accessed by SUT that were assigned to a resource location within an effect
Mutation Operators	Change Guard
Selection Criterion	Transitions whose effect has an expression in which a resource is referred by **ResourceCompetition**, which is indirectly influenced by a guard, e.g. where the control flow that leads to reading or writing a resource is influenced by a variable that is read by the guard

This is basically a union of all possible mutations based on the sub concepts of ResourceIssues.

Concept of Uncertainty Taxonomy	TechnologicalProcess::ResourceIssues::**ResourceCompetition**
Mutation Operators	Change Effect Change Transition Source Change Transition Target
Selection Criterion	Transitions whose effect has an expression in which a resource is referred by **ResourceCompetition**
Mutation Operators	Change Guard
Selection Criterion	Transitions whose effect has an expression in which a resource is referred by **ResourceCompetition**, which is indirectly influenced by a guard, e.g. where the control flow that leads to reading or writing a resource is influenced by a variable that is read by the guard

Resource competition can be distinguished in direct and indirect resource competition.

Direct resource competition occurs if the very same resource is accessed by one instance while being accessed by a second one. This is known as resource contention in computer science.

Indirect resource competition occurs if one instance would like to access resource A but requires accessing a resource B first, and second instance is trying to access B. This may be the case for resources that aren't directly accessible, such as stacks, either in the physical or the cyber world.

Concept of Uncertainty Taxonomy	TechnologicalProcess::ResourceIssues::**ResourceLocation**
Mutation Operators	Change Effect
Selection Criteria	Elements that store resources based on the assignments in an effect to a property of the state machine that is referred by **ResourceLocation**
	Resources accessed by the system under test that were assigned to a resource location within an effect

The aim of applying the Change Effect mutation operator is to change the assignment of a resource to a location in a wider sense, that is with respect to its physical location or its virtual location (in case of virtual resources, such as data representing physical entities).

Concept of Uncertainty Taxonomy	TechnologicalProcess:: ApplicationIssues::**InsufficientResources**
Mutation Operators	Change Effect
Selection Criterion	Elements that store resources based on the assignments in an effect to a property of the state machine that is referred by **InsufficientResources**

Mutations to effects introduced by applying the Change Effect mutation operator are applied such that the cyber resources, e.g., memory, or physical resources, e.g., goods, aren't available anymore, for instance by removing the resources or assigning them to other entities.

Concept of Uncertainty Taxonomy	TechnologicalProcess:: ApplicationIssues::**FunctionalFault**
Mutation Operators	n/a
Selection Criterion	n/a

Functional faults are an outcome when the system under test is stimulated with correct values but did not respond to it with the expected behaviour. Hence, this will not be addressed in the Uncertainty Testing Framework because no uncertainty in the environment is involved.

Concept of Uncertainty Taxonomy	TechnologicalProcess::ResourceIssues::**HumanBehavior**
Mutation Operators	Any
Selection Criterion	Any transitions with operations or any operation of an interface referred by **HumanBehavior** as a trigger

This reflects any mutation that may result due to human behaviour.

Concept of Uncertainty Taxonomy	TechnologicalProcess::ResourceIssues::**NaturalProcess**
Mutation Operators	Any
Selection Criterion	Elements referred by this **NaturalProcess** that are related to a physical (i.e. the physical environment)

Mutations will result in any modifications related to the digital interfaces, i.e. would reflect uncertainty from the cyber environment related to protocols.

2.2.2 Fitness Evaluation

In addition to the fitness factor framework described D3.1, we extended this framework to enable detection of uncertain behaviour that would manifest in non-continuous values in a data row. Non-continuous values are below an acceptable deviation. However, the difference between two data points is higher than the acceptable threshold, leading to "jumps" between data points point. Such uncertain behaviours, e.g., non-continuous data values, are depicted in Figure 3.

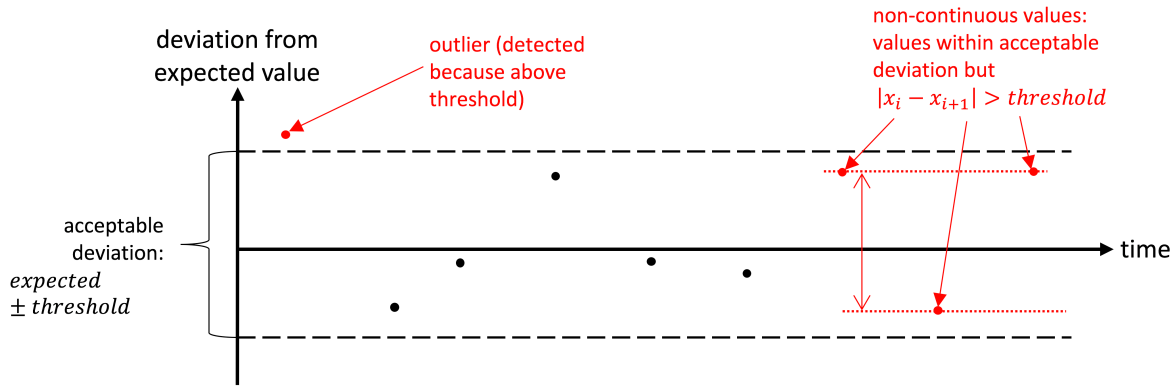

Figure 3. Outliers and Non-Continuous Values

2.2.3 Test Strategies

No update for this milestone. Initial versions of test strategies were proposed in Section 4.2.1 of D3.1.

2.2.4 Test Data Generation

Test data generation is realized by facilities of MS SpecExplorer.

2.3 Uncertainty Testing at Infrastructure Level

In this section, we describe our strategies for testing CPS at the Infrastructure level

Testing the infrastructure of CPS 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 behavioural 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. Uncertainties at infrastructure level may appear due to heterogeneity of CPS and data transmissions in it, i.e. physical units, sensors, actuators, networks, protocols, cloud services and other parts of the CPS. Additionally, means of discovery of possible unknown uncertainties from a potentially infinite domain of unknown uncertainties are necessary. Thus, to correctly test and identify uncertainties in the infrastructure of CPSs, we focus 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 behaviour has occurred or not.
- Testing for particular properties of a CPS, both at design-time and run-time, in order to test and discover unknown uncertainties in the system and provide possible recommendations for model evolution

Figure 4 shows an overview of uncertainty testing at the infrastructure level of CPS. For testing uncertainty at infrastructure level, we use as input the UML Model as obtained from the U-Test Uncertainty Modelling Framework. The UML Model contains Classes and StateMachines to model the overall system architecture and behaviours, depicted on Class and StateMachine Diagrams, respectively. UML StateMachines are the input to the State Machine Transition Correctness Testing Strategies (see D3.1). The Transition Correctness are transformed in 11 test plans and then in concrete test plans. In turn, the SUT architecture depicted on UML Class diagrams where SUT components and connectors types are defined. Different test configurations can be obtained by instantiating Classes (as InstanceSpecifications), their Properties (as Slots), and their Relationships (as Links). The SUT

architecture (at the type level) and test configurations (at the instance level) are used to generate test cases by U-CertifyIT (D4.2) and run-time tests descriptions by TUW Platform for Run Time Testing of Cyber-Physical Systems (D3.1).

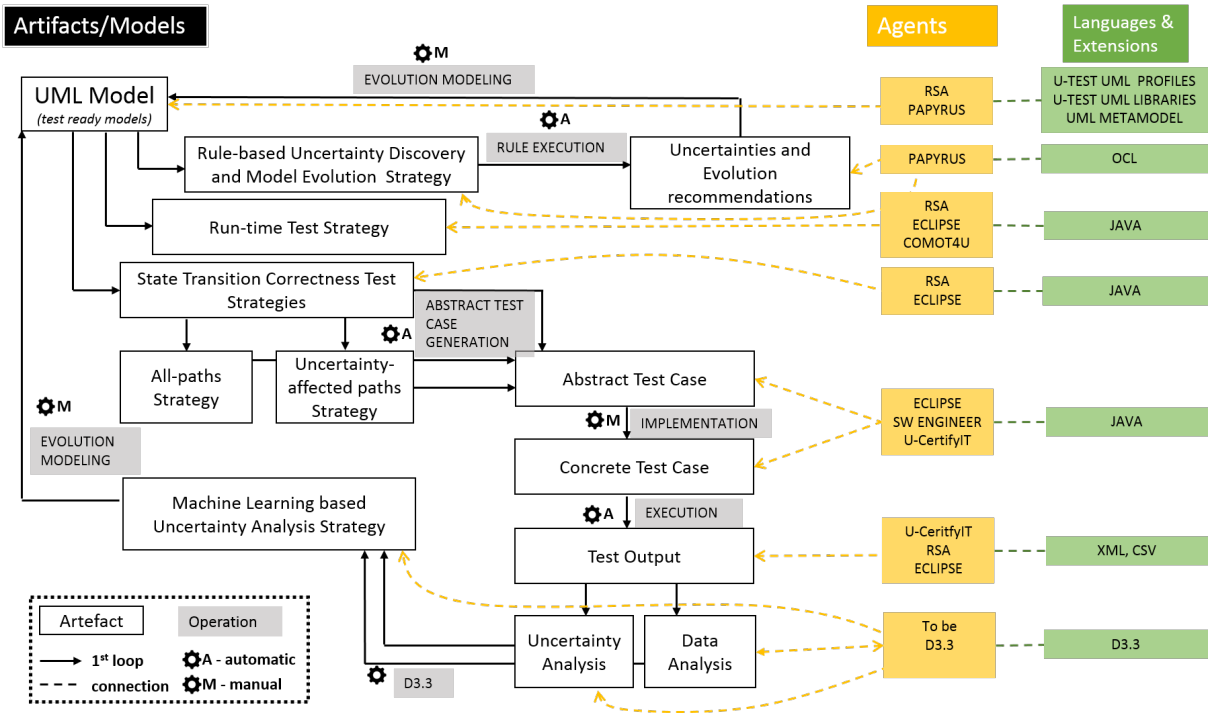

Figure 4. Uncertainty Testing at Infrastructure Level Overview

In this deliverable, we focus on Rule-based Uncertainty Discovery and Model Evolution Strategy, which takes the UML Model annotated with CPS Infrastructure Profile and applies a set of predefined rules (i) to discover potential unknown uncertainties and (ii) to suggest model evolution recommendations as an output.

2.3.1 Uncertainty Model Evolution at Infrastructure Level

The Rule-based Uncertainty Discovery and Model Evolution Strategy uses rules to detect the potential unknown uncertainties in CPS on UML Model and suggests model evolution actions intended as additions/deletions/updates of structural and/or behavioural model elements. Whereas previously developed test strategies (D3.1) aimed at testing the known uncertainties, the Rule-based strategy aims at detecting unknown uncertainties in proactive manner to prevent and then reduce the possible unknown uncertainties in the SUT in the later stages of the U-Test engineering process.

As depicted in Figure 5 the Rule-based Uncertainty Discovery and Model Evolution Strategy takes a UML Model as input, suitably annotated with stereotypes from the CPS Infrastructure profile.

Rules consist of queries on model elements of a test-ready UML Model, define conditions that determine the detection of potential unknown uncertainty over the collected model elements, generate Boolean result (true = detected, false = not detected), and suggest potential

changes/evolutions on the source UML model. If these recommendations are realized on the source UML Model (i.e. addition/deletion/update of model elements and/or stereotype applications), a new evolved version of the source UML Model is obtained.

The envisaged Rule-based Uncertainty Discovery and Model Evolution Strategy is iterative. At any iteration of the proposed strategy a new evolved version of a UML Model can be obtained by realizing the suggested recommendations from a non-empty set of rules applied to the whole model (e.g., all system components and all test configurations) or a subset of it (e.g., only to selected components and test configurations). Currently, it is up to the modeller deciding the needed iterations, i.e. which rules (out of a set of predefined ones) to apply, their scope (the whole model or only a subset of it), and the order of their application. It is worth noting that the modeller can decide to completely skip the Rule-based Uncertainty Discovery and Model Evolution steps or ignore suggested recommendations and proceed with the next U-Test engineering steps requiring the UML Model.

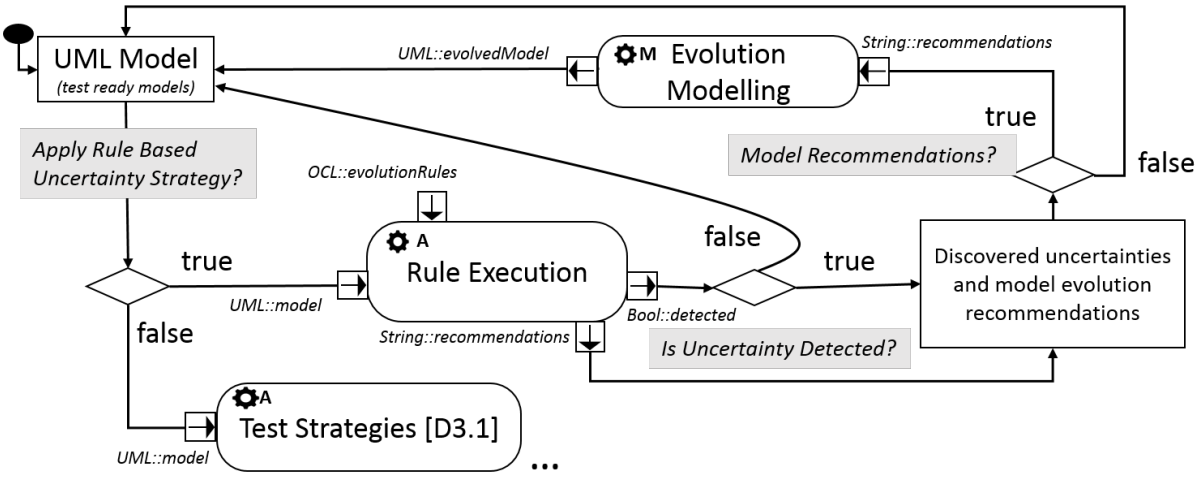

Figure 5. Rule-based Uncertainty Discovery and Model Evolution strategy overview

The evolution rules required to perform the Rule Execution step in Figure 5 have been first specified in a tabular with the following template.

Every defined rule has the following attributes:

- Rule level
- Rule aspect
- Rule element (from profile)
- Rule description
- Rule condition (algorithm)
- Model evolution recommendation
- Model evolution recommendation domain
- Non-functional property (quality) affected
- Source or justification
- Rule-specific attribute (optional)

- Rule-specific attribute execution period (optional)

Rules are categorized in different levels (application, infrastructure, integration) as per division in U-Test project. Rule aspects are categorized as per division proposed by NIST CPS Framework [4] - data, functional, business, human, trustworthiness, timing, boundaries, composition, lifecycle, plus security, as intrinsic to all aspects of CPS. Rules reference the profile, which they are using as a basis for uncertainty detection and a specific profile element. Each rule has a verbal description and the accompanying algorithm, with the conditions, which must be met. Each rule provides a possible recommendation for model evolution, while stating the possible domain for further research. Each rule affects one or more non-functional properties (qualities) of a CPS. Each rule has to be backed up by at least one source (paper, best practice recommendation, existing standards, etc.), or in experimental research domains, if there are no sources, it needs to have a justifiable clarification. If there is no attribute in referenced profile to apply the rule, we define them as rule-specific attributes. In such case, we define if the attribute value represents the design-time (expected) or run-time (measured) value.

Initial set of 20+ defined rules are located in an open source TUW U-Test GitHub[1] code repository, together with other test strategies implemented. Further definition of rules is an on-going work. Rules currently mostly cover aspects of data (since uncertainties at infrastructure level appear due to heterogeneity of CPS and data transmissions in it), security aspect, and current trends in CPS evolution (e.g. CPS to Fog/Edge evolution, CPS Elasticity, etc.) across multiple aspects. An example of data aspect rule, security aspect rule and CPS elasticity rule is shown in Table 2. Additionally, table shows FPX/ULMA use cases where particular unknown uncertainty can be discovered with the usage of the specific rule. However, the full effectiveness of the Rule-based strategies, as well as other previously implemented strategies, and full evaluation based on project-defined metrics (e.g., number of previously unknown uncertainties identified by the reporting system for use case x) will be provided as output of empirical evaluation task in D5.4.

The expected outputs of the rules are:

(i) Textual recommendation to modellers displayed as uncertainty warning messages (e.g., via Console view in Eclipse-based environment) with traceability links among applied rules, checked model elements, and model evolution recommendations.

(ii) Application(s) of stereotypes defined in the CPS *Uncertainty* profile to model element(s) with detected uncertainty.

One goal of the Rule-based Uncertainty Discovery Strategy is to decouple its realization from modelling guidelines and tools required by EGM and FF.

In order to decouple the proposed strategy from specific modelling guidelines proposed by EGM and FF, we decided to realize a CPS Infrastructure Model Library, which can be imported[2] in U-Test ready models, where the discovery rules are applied. Moreover, being decoupled from U-Test ready model,

[1] TUW U-Test GitHub, https://github.com/tuwiendsg/COMOT4U

[2] The import step can be realized “by reference” or “by value”. In the first case, the model library is read only while in the latter the imported library can be modified.

the model library can evolve independently from U-Test models to accommodate refinements of existing discovery rules or definition and implementation of new ones.

Concerning tool support, two Eclipse-based UML modelling tools are adopted to create U-Test ready models, i.e., RSA and Papyrus, both implemented on top of Eclipse UML. Both tools provide support to OCL, the OMG standard query language for MOF-based artefacts, that we choose to implement the rule specifications given in a tabular form in Table 2.

In order to produce the expected outputs, rules will be implemented in OCL [5] (Object Constraint Language) using the Eclipse OCL plugin. Since OCL is a side-effect free query language for artefacts serialized as OMG XMI documents, any model evolution (i.e., change on the test ready model like stereotype application) are planned to be implemented via external Java-based routines with support of the Eclipse UML API.

2.3.2 Test Strategies

Infrastructure level test strategies are currently divided in four categories, as previously shown in Figure 4:

- State Machine Transition Correctness Testing Strategies (Test Correctness of State Transitions in All Test Paths Strategy and Test Correctness of State Transitions in Uncertainty-affected Test Paths Strategy),
- Run-Time Testing Strategies (periodic testing, event-based testing, direct testing, indirect testing)
- Rule-based Uncertainty Discovery and Model Evolution Strategy and
- Machine Learning based Uncertainty Analysis Strategy, aimed at analysis of uncertainty patterns related to particular infrastructure elements (ongoing work, to be reported in D3.3).

For the Rule-based Uncertainty Discovery and Model Evolution Strategy, Figure 6 shows an example where we start from the initial model with applied referenced profiles and run the strategy over the model elements. The strategy discovers three possible uncertainties with model recommendations, which modeller implements. Over the three evolved models, strategy is run again. Rule-based strategy finds no uncertainties in first evolved model. In the second, one new unknown uncertainty is discovered, model is again evolved, and test strategy ran again, and no new unknown uncertainties are found. Additionally, rule-based testing strategy finds one more unknown uncertainty in the third evolve model, however, the stakeholders of that particular CPS under test are unable to evolve the model. The reasons for inability for model evolution are specific for each particular CPS (e.g., lack of API, change requires too much effort, change does not fit with business plan, etc.). However, this newly discovered unknown uncertainty transitioned from an unknown uncertainty domain to a known uncertainty domain can be further tested and observed with previously developed testing strategies, e.g., Run-time Test Strategy or State Machine Transition Correctness Testing Strategies.

Table 2. Example of data, security and CPS elasticity rule

		Data rule	Security rule	CPS elasticity rule
	Rule Name	Check Timestamp Mechanism Availability	Check If Safety Critical Actuator	Check Data Management Mechanism Availability
Rule attributes	Rule level	CPS Infrastructure	CPS Infrastructure	CPS Infrastructure
	Rule aspect	Data	Security	Data, Lifecycle
	Rule element	CPSProfile::Unit	CPSProfile::Actuator	CPSProfile::CPS
	Rule description	Sensor data should be timestamped, to monitor the latency between the following event occurrences, i.e. the measurement event occurrence and the data availability to Unit event occurrence.	If an actuator is safety-critical (e.g., centrifuge in chemical plant, that may cause harm to a CPS), consider adding new physical controls over the CPS (e.g., manual valves) to reduce possible harm in case of misuse.	Sensors may produce too much data (e.g., if sensors are activated by certain events) which the CPS cannot handle due to its limitations. Please test the system with both maximum and minimum workload of sensors to find out its limitations. Additionally, please ensure the elasticity of the CPS in such occasions.
	Rule algorithm	`IF Unit. timestampMechanism== notImplemented`	`IF Actuator. safetyCritical== TRUE`	`IF SensorDataManagementMechanism== FALSE`
	Model evolution recommendation	Please implement the timestamp mechanism.	Please consider adding new physical controls over the CPS.	Please test and evolve the system as instructed.
	Model evolution domain	Data timestamping	Physical safety	Elasticity
	Non-functional property	Latency	Safety	Elasticity
	Source or justification	NIST CPS Framework [4], page 4	NIST CPS Framework [4], page 79	DSG TU Wien, SYBL [6]
	Rule-specific attribute	CPS Infrastructure stereotype: Unit attribute: hasTimestampMechanism values: true, false	CPS Infrastructure stereotype: Actuator attribute: isSafetyCritical values: true, false	CPS Infrastructure stereotype: CPS attribute:hasSensorDataMngmtMechanism values: true, false
	Attribute execution period	Design-time	Design-time	Design-time
	Uncertainty found:	ULMA UC2_INFR_2.1	ULMA UC2_INFR_1.2	FPX UC1_INFR_8

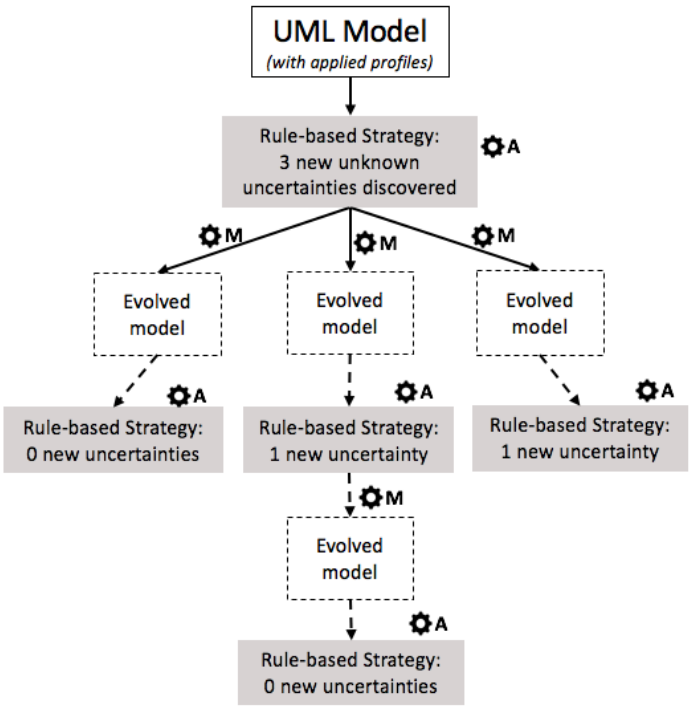

Figure 6. Example of an Iterative Application of Rule-based Strategy

2.3.3 Test Data Generation

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

a. Information captured as UML Profiles, Class Diagrams, and State Diagrams during the CPS modelling phase, as described in D2.1 and D2.2. This information is used in generating the abstract transition correctness tests, run-time test descriptions, discovery of new unknown uncertainties and model evolution.
b. Expert knowledge brought by CPS owner/user used in the implementation of the concrete tests according to particularities of the tested CPS.
c. Test data generation as implemented in U-CertifyIT together with test strategies.

2.4 Uncertainty Testing at Integration Level

This section presents the overview of the work related to UTF at the integration level from the following four perspectives, as shown in Figure 7: 1) Uncertainty-wise Model Evolution, 2) Uncertainty-wise Test Case Generation, 3) Uncertainty-wise Test Case Minimization, and 4) Uncertainty-wise Test Case Prioritization. In this section, we only provide an overview of each of these activities and all the technical details are provided in the form of two technical reports, i.e. Technical Report 2: Uncertainty-Wise Evolution of Test Ready Models [7] and Technical Report 3: Uncertainty-Wise and Time-Aware Test Case Prioritization with Multi-Objective Search [8]. The short summaries of these two technical reports can be found in the Appendix. The full technical reports are attached with this deliverable as two separate documents (TR4.1.pdf and TR7.pdf).

Uncertainty-wise Model Evolution (C1) will be described in Section 2.4.1, Uncertainty-wise Test Case Generation (C2) in Section 2.4.2, Uncertainty-wise Test Case Minimization (C3) in Section 2.4.3, and Uncertainty-wise Test Case Prioritization (C4) in Section 2.4.4. Comparing with D3.1, UncerPlore in uncertainty-wise model evolution (C1.2) and uncertainty-wise test case prioritization (C4) are newly proposed in this deliverable, and the description for the rest (C1.2, C2, C3, and C4) only highlights the updates.

As shown in Figure 7, the initial input of the UTF at Integration Level, i.e. belief test ready model (BM), is the output of the UncerTum (C0) (presented in the deliverables of WP2, D2.2 and D2.3). The overall workflow is:

1. Belief test-ready models are evolved based on the uncertainty-wise model evolution component (C1);
2. The uncertainty-wise test case generation component (C2) takes (evolved) belief test-ready models as input to generate abstract test cases;
3. By taking generated abstract test cases as input, the uncertainty-wise test case minimization component can be optionally used to minimize the number of abstract test cases when needed (C3);
4. The uncertainty-wise prioritization component (C4) takes abstract test cases and test results as input and prioritizes the sequence to execute test cases in a cost-effective way;
5. The uncertainty-wise test case generation component (C2) takes the (minimized/prioritized) belief test-ready models as input to generate executable test cases;
6. The uncertainty-wise test case execution component (C5) takes the (minimized/prioritized) test cases to execute on test infrastructure.

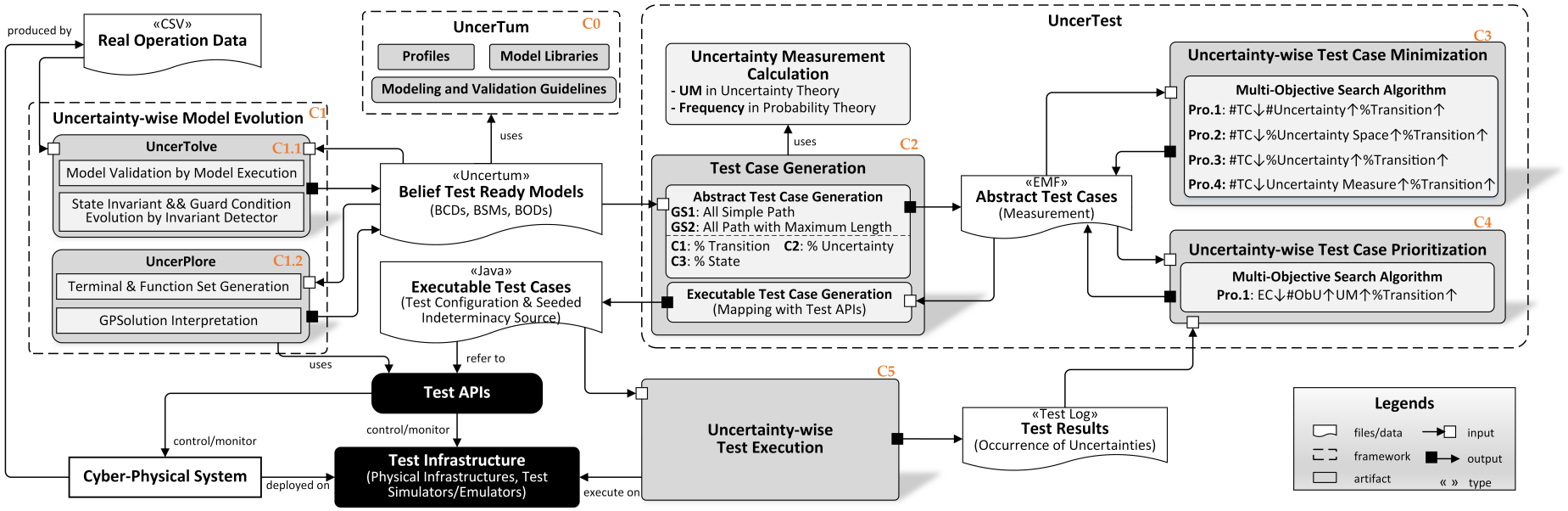

Figure 7. Overview of Uncertainty Testing Framework at Integration Level

The rest of this section describes more details about each component.

2.4.1 Uncertainty-wise Model Evolution

This section presents the uncertainty-wise model evolution (C1 in Figure 7), which contains two methodologies: UncerTolve (C1.1, Section 2.4.1.1) and UncerPlore (C1.2, Section 2.4.1.2)

2.4.1.1 UncerTolve

This section presents updates of UncerTolve (C1.1 in Figure 7), comparing with D3.1. A summary of our work on UncerTolve is provided in Technical Report 2: Uncertainty-Wise Evolution of Test Ready Models [7]. The corresponding technical report (**TR4.1.pdf**) is also attached.

In general, the main updates in the technical report (**TR4.1.pdf**) include:

1. Scientific challenges, objectives, context, scope and contribution are clearly described in Section 1, and Figure 1 is updated to clarify the context and scope of UncerTolve;
2. The presentation of the UncerTolve framework is restructured in Section 5.1. Figure 9 is newly added to describe the high-level components of UncerTolve and Table 2 discusses the rationale behind the selection of techniques/languages/tools for the implementation.

2.4.1.2 UncerPlore

This section presents the UncerPlore framework, which evolves test-ready models by using genetic programming (GP) [9] [10] and benefiting from runtime test ready model execution on the dedicated test infrastructure (physical infrastructure or Simulators/Emulators). The overview diagram is shown in Figure 8. The UncerPlore framework is implemented on the ECJ tool [11].

As shown in Figure 8, we formalized a belief state machine as the basic input of GP (Section 2.4.1.2.1): 1) a state is formalized as a terminal that is evaluated based on the runtime status of the system; 2) a transition is formalized as a function with two arguments that indicate the two statuses before/after executing this transition. Note that required data for executing events of transitions can be generated by the OCL Solver (EsOCL [12]). In addition, we define an algorithm to interpret tree structure results produced by GP, which will be described in Section 2.4.1.2.2.

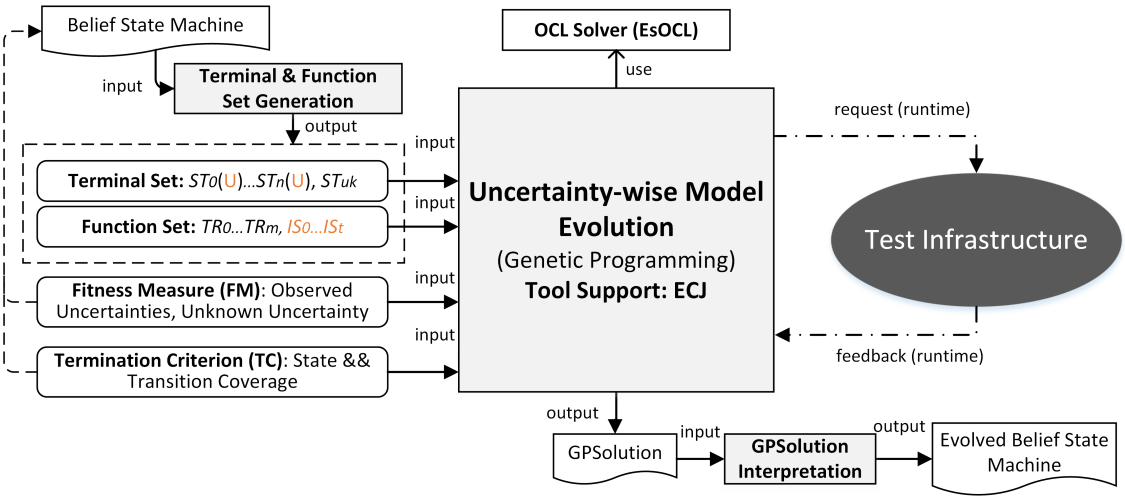

Figure 8. Overview of UncerPlore (C1.2)

2.4.1.2.1 Problem Representation

In [10], Koza defined five key steps to enable Tree-based GP evolution of programs. These steps include: 1) Specification of set of terminals, e.g., external inputs, 2) A set of functions, 3) A fitness measure, 4) parameter settings, 5) Termination criteria [13]. To evolve belief state machine in our context, we defined the formalization shown in Table 3. The terminal set and function set are the basic ingredients for GP to create programs [13], so we further defined the generation rules (Table 3) to automatically generate terminal set and function set from the belief state machine.

Table 3. Formalizing the uncertainty-wise evolution problem as a GP problem

GP	Definition	Description

Terminal Set	$TS = \{ST_0 \dots ST_n, ST_{uk}\}$	Each state (*ST*) is converted into a terminal, whose data type is Boolean. ST_{uk} is a state that all existing states are not satisfied.
		`ST extends Node{`
		` data:Boolean`
		` // return true when «BeliefElement»State contains uncertainty.`
		` isUncertain():Boolean`
		` // operation to evaluate the state invariant based on runtime status`
		` evaluateStateInvariant():Boolean`
		`// operation to evaluate occurrence of indeterminacy source if it has`
		` evaluateIndSpecification():Boolean`
		`}`

Function Set	$FS = \{TR_0 \dots TR_m\} \cup \{TRInS_0 \dots TRInS_t\}$	Each transition (*TR*) or each transition with specified indeterminacy input (*TRInS*) is converted into one function with two arguments and a Boolean return value, represented as $TR\ (a1, a2) = a1 \wedge a2$ under the execution a1→TR→a2 *true* result indicates the valid program.
		`TR extends Node{`
		` children[2]:Node`
		` //execute event of transition, OCL Solver is used when guard condition exits`
		` execute():void`
		` // optionally specify the precondition to execute this transition`
		` evaluatePrecondition():Boolean`
		`}`
		`TRInS extends TR{`
		` //execute the trigger to enable indeterminacy source`
		` enableInSInput():void`
		`}`

Fitness Measure	$POU = \frac{\#ObUn_{kn}}{\#Un_{kn}}$ $FM = 1 - \frac{1}{2}\left(POU + \frac{\#Un_{uk}}{\#Un_{uk}+1}\right)$	*POU* is the percentage of observed uncertainty comparing with specified uncertainty, where $\#ObUn_{kn}$ indicates the number of observed uncertainties, and $\#Un_{kn}$ indicates the number of specified uncertainties. *FM* is fitness measure to evaluate the solution, which is related to *POU* and observed unknown uncertainties.
Termination Criterion	(isValid) and ($coverage_{runtime}$ >= $coverage_{specified}$)	An evolution is terminated when the corresponding GPTree is valid (with the *true* result) and involved elements are more than specified ones.
Parameter	Crossover Operator: Subtree Crossover [9] [10] Mutation Operator: Point Mutation [9] [10] Population size: 50 Maximum Generations: 1000	Default parameter setting.

2.4.1.2.2 Interpretation of GP Solution

This section describes the interpretation of the Tree-based GP solution to generate evolved belief state machines. Based on the formalizations of the GP problem in Table 3, 1) each transition is presented as non-leaf node, 2) each state is presented as the leaf node in the tree, 3) the source and target of the transition are presented as the previous and next visiting node based on inorder traversal, and 4) the incomings and outgoings of the state are presented as the previous and next visiting node based on inorder traversal. The pseudocode of the algorithm to generate evolved belief state machines is shown in Figure 9, and a simple example describing the traversal process is presented in Figure 10.

```
Algorithm  GenerateBSM(node:Node, sm:BSM, list:list<Node>)
    Input  node is the root of GPTree
           list records the traversal sequence
   Output  sm is the evolved state machine
Begin
    1  if(node.children != null)
    2    generateBSM (node.children[0], sm, list)
    3  if(node is kind of ST)
    4    state = getState(node, sm)
    5    sm.update(state)
    6    if(list.last != null)
    7      state.incomings.add(getTransition(list.last,sm))
    8      getTransition(list.last,sm).target = state
    9  if(node is kind of TR)
   10    transition = getTransition(node, sm)
   11    sm.update(transition)
   12    transition.source = getState(list.last, sm)
   13    getState(list.last, sm).outgoings.add(transition)
   14  list.add(node)
   15  if(node.children != null)
   16    generateBSM (node.children[1], sm, list)
End
```

Figure 9. The algorithm to generate evolved belief state machines based on GP solutions

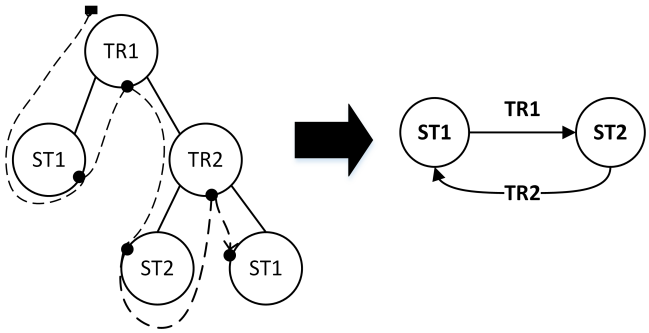

Figure 10. An example of generating belief state machines from GPTree

2.4.2 Uncertainty-wise Test Case Generation

No Specific Update. The final version will be provided in D3.3.

2.4.3 Uncertainty-wise Test Case Minimization

No Specific Update. The final version will be reported in D3.3.

2.4.4 Uncertainty-wise Test Case Prioritization

As shown in Figure 7, the key inputs of uncertainty-wise test case prioritization are abstract test cases that contain uncertainty information, e.g. Uncertainty Measure, the number of uncertainty, and execution results that contain the execution time and observed uncertainty for each run. Based on

these two key inputs, we formalized our *uncertainty-wise*, *time-aware, multi-objective test case prioritization problem* as a search problem and solved it using the well-known multi-objective search algorithm *NSGA-II* [14]. Corresponding to the four objectives, we defined four cost-effectiveness measures: 1) the total execution time of prioritized test cases (to minimize); 2) the average uncertainty measure (adopted from Uncertainty Theory [15]) of the prioritized test cases (to maximize); 3) the average number of observed uncertainties of prioritized test cases (to maximize); and 4) the transition coverage (to maximize). Based on these objectives and measures, we define a fitness function to guide the algorithm towards finding optimal solutions.

We evaluated *NSGA-II* and compared it with *Greedy* and *Random Search* (*RS*), with an industrial case study (Quuppa by FPX) requiring prioritizing 336 test cases. We further evaluated the performance and scalability of the algorithm with 72 simulated problems, carefully constructed based on a test case repository containing 2085 test cases. Results show that *NSGA-II* achieved significantly better performance than *RS* and *Greedy* for solving the uncertainty-wise and time-aware test case prioritization problem for the industrial case study and the 72 simulated problems. Please refer to the online technical report [8] and also attached as TR7.pdf in the deliverable, for more details. A summary of our work in TR7.pdf is provided in the Appendix under the Technical Report 3: Uncertainty-Wise and Time-Aware Test Case Prioritization with Multi-Objective Search [8].

3 Summary and Conclusion

3.1 UTF at the Application Level

Achievements of M4

The fourth milestone was achieved by the improvements of the genetic algorithm with mutation operators for guards and effects in form of UML activities that are exploiting further information from modelled uncertainty. It was specified which concepts of the Uncertainty Taxonomy (implemented by the Uncertainty Modelling Framework) are used to select mutation operators and to which elements they are applied. Furthermore, refinements of the fitness function framework were introduced that allow to detected further kinds of uncertain behaviour related to non-continuous behaviour of a CPS application.

Plan for achieving M5

For the fifth milestone, we will focus on test strategies aiming at discovering unknown uncertain behaviours, also by improving the genetic algorithm. This will be done by

- investigating further improvements of the existing crossover operators and new options for them,
- extending the genetic algorithm with configuration points for random variation (i.e. decreasing the amount of information taken from modelled uncertainties used to guide the mutation) and
- feeding back first results from the evaluation of the model evolution algorithm on the pilots.

3.2 UTF at the Infrastructure Level

Achievement of M4

The fourth milestone was achieved through the introduction of Rule-based Uncertainty Discovery and Model Evolution Strategy. The strategy aims at discovery of new unknown uncertainties as well as providing recommendations for model evolution, while making use of profiles created in WP2. Initial version of model evolution algorithm was achieved as a set of rules, documented and implemented in OCL. Rules aim at discovery of new realistic (unknown) uncertainties, evolve the model and provide recommendations for further model evolution. The evolved models can be used as an input to previously developed test strategies, as well as an input to strategies implemented in U-CertifyIT, to generate new test cases.

Plan for achieving M5

For the fifth milestone, we will focus on extension of the initial set of rules (with emphasis towards current generic evolution of CPS towards Edge/Fog/IoT), as well as on provision of a methodology for further profile and rule creation for different aspects of CPS (data, functional, business, human, trustworthiness, timing, boundaries, composition, lifecycle and security) centred around the core CPS profile. Additionally, we plan to investigate the Machine Learning based Uncertainty Analysis approach, aimed at analysis of uncertainty patterns related to particular infrastructure elements, i.e. to investigate whether particular types of uncertainties can be linked to specific CPS elements

3.3 UTF at the Integration Level

Achievements of M4

We have successfully reached the milestone M4 regarding the UTF V.2 for uncertainty testing at the Integration level of CPS. More specifically, the main improvements for uncertainty testing at the Integration level as compared to the UTF V.1 include:

1) The update of UncerTolve for supporting the evolution of test-ready models using real operation data (Section 2.4.1.1);
2) The development of an initial version of the new model evolution framework (UncerPlore), which evolves test-ready models using genetic programming (GP) [9] [10] and benefiting from runtime test ready model execution on the dedicated test infrastructure (Section2.4.1.2);
3) An uncertainty-wise test prioritization solution to optimize the order of executing test cases in the cost-effective way (Section 2.4.4).

Plan for achieving M5

For the fifth milestone, we will work on

1) Providing the recommendation of how to configure the proposed test strategies for the test case generation;
2) Developing additional uncertainty-wise problems for test case minimization and prioritization, and conducting experiments with additional search algorithm;
3) Finalizing the new model evolution framework UncerPlore and performing experiments.

Appendix

Technical Report 1: Implementation Recommendations for Rule-based Uncertainty Discovery and Model Evolution Strategy

The Rule-based strategy algorithm iterates through each test path entry containing a state and the transition to the next path state. It traverses through all states, not only uncertainty-affected ones. Unlike with test path generation, this should not cause an overhead problem as described in D4.2, since this strategy checks on properties of only the current state at particular moment without consideration of another state. During the path traversal, it checks on the properties of particular state or transition, as shown in Listing 1.

```
ALGORITHM rule_based_uncertainty_discovery_and_model_evolution_strategy
INPUT: test_paths
OUTPUT: textual_or_console_output

1 FOR EACH test_path IN test_paths DO
2
3    IF ANY state in test_path.entries HAS RuleCondition
4      textual_or_console_output.add(ruleOutput)
5    IF ANY transition in test_path.entries HAS RuleCondition
6      textual_or_console_output.add(ruleOutput)
7
8         ... -- rest in similar state machine diagram iteration sense as
generate_transition_correctness_tests in D3.2
```

Listing 1: Test strategy 2 – Checking particular state property during path traversal

`RuleCondition` refers to rule condition attribute of rules (as described in Section 2.3), e.g. `state.timestampMechanism==notImplemented`. `RuleOutput` refers to textual output of a rule, i.e. rule description, model evolution recommendation, model evolution recommendation domain, source or justification (as described in Section 2.3), and the name of a state or transition where an unknown uncertainty is discovered.

Additional set of implementation recommendations includes:

- ability to categorize rules in different aspect (e.g. a set of data rules, behavioural rules etc. defined as different documents)
- ability to select a set of rules to execute
- ability of run-time addition or removal of rules (e.g. rules defined in specific document with Java syntax, which are then imported into the path traversal code. This feature would also remove the need of recompilation of the implemented plugin every time a new rule is defined)
- instant output in console view

Technical Report 2: Uncertainty-Wise Evolution of Test Ready Models

The details of this technical report [7] can be found in a separate self-contain document (TR4.1.pdf) attached with this deliverable. This technical report paper describes our detailed approach for the

Uncertainty-Wise Evolution of Test Ready Models (UncerTolve), which we have briefly presented in Section 2.4.1.1. An summary of this technical report is given as follows.

Context: Cyber-Physical Systems (CPSs), when deployed for operation, are inherently prone to uncertainty. Considering their applications in critical domains (e.g., healthcare), it is important that such CPSs are tested sufficiently, with the explicit consideration of uncertainty. Model-based testing (MBT) involves creating test ready models capturing the expected behaviour of a CPS and its operating environment. These test ready models are then used for generating executable test cases. It is, therefore, necessary to develop methods that can continuously evolve, based on real operational data collected during the operation of CPSs, test ready models and uncertainty captured in them, all together termed as Belief Test Ready Models (BMs)

Objective: Our objective is to propose a model evolution framework that can interactively improve the quality of BMs, based on operational data. Such BMs are developed by one or more test modellers (belief agents) with their assumptions about the expected behaviour of a CPS, its expected physical environment, and potential future deployments. Thus, these models explicitly contain subjective uncertainty of the test modellers.

Method: We propose a framework (named as UncerTolve) for interactively evolving BMs (specified with extended UML notations) of CPSs with subjective uncertainty developed by test modellers. The key inputs of UncerTolve include initial BMs of CPSs with known subjective uncertainty and real data collected from the operation of CPSs. UncerTolve has three key features: 1) Validating the syntactic correctness and conformance of BMs against real operational data via model execution, 2) Evolving objective uncertainty measurements of BMs via model execution, and 3) Evolving state invariants (modelling test oracles) and guards of transitions (modelling constraints for test data generation) of BMs with a machine learning technique.

Results: As a proof-of-concept, we evaluated UncerTolve with one industrial CPS case study, i.e., GeoSports from the healthcare domain. Using UncerTolve, we managed to evolve 51% of belief elements, 18% of states, and 21% of transitions as compared to the initial BM developed in an industrial setting.

Conclusion: UncerTolve can successfully evolve model elements of the initial BM, in addition to objective uncertainty measurements using real operational data. The evolved model can be used to generate additional test cases covering evolved model elements and objective uncertainty. These additional test cases can be used to test the current and future deployments of a CPS to ensure that it will handle uncertainty gracefully during its operations.

Technical Report 3: Uncertainty-Wise and Time-Aware Test Case Prioritization with Multi-Objective Search

The details of this technical report [8] can be found in a separate self-contain document (**TR7.pdf**) attached with this deliverable. This technical report paper describes our detailed work in uncertainty-wise test case prioritization, which we have presented in Section 2.4.4. An summary of this technical report is given as follows.

Context: Complex systems (e.g., Cyber-Physical Systems) that interact with the real world, behave in an unstipulated manner while operating in uncertain environments. Testing such systems in uncertainty is a big challenge. Devising uncertainty-wise testing solutions can be considered as a mandate for dealing with this challenge. Though uncertainty-wise testing is gaining attention in the last few years, industry-strengthening solutions are still missing.

Objective: Our objective is to propose an uncertainty-wise test case prioritization approach that can significantly improve the cost-effectiveness of test case execution to maximize the occurrence of uncertainty.

Method: In this paper, we propose an uncertainty-wise, search-based, multi-objective test case prioritization approach, with a fitness function defined based on four cost-effectiveness measures: one subjective and one objective uncertainty measures, execution time, and transition coverage.

Results: We evaluated the well-known multi-objective search algorithm NSGA-II by comparing it with Greedy and Random Search (RS), with a real industrial case study. In addition, we created 72 additional simulated problems of varying complexity based on the real case study. Results show that NSGA-II achieved significantly better performance than RS and Greedy for both the real industrial case study and the simulated problems. On average, NSGA-II improved prioritization by 18% and 22% as compared to RS and Greedy respectively.

Conclusion: This paper presented an uncertainty-wise and time-aware test case prioritization, which was specifically developed to improve the cost and effectiveness of test case execution and at the same time maximizing the occurrence of uncertainties.

Bibliography

[1] U-Test Consortium, “U-Test Deliverable D1.2: Report on Taxonomy.”

[2] U-Test Consortium, “U-Test Deliverable D2.2: Report on Uncertainty Modelling Framework V2.”

[3] G. Weissenbacher and (editor), “D 3.1b - Fault Models (Final Version),” 2008. [Online]. Available: https://www.mogentes.eu/public/deliverables/MOGENTES_3-09_1.0r_D3.1b_Fault_Models_Mutations.pdf.

[4] NIST – National Institute of Standards and Technology, “CPS PWG Cyber-Physical Systems (CPS) Framework Release 1.0.” 2015.

[5] Object Management Group (OMG), “Object Constraint Language.” .

[6] G. Copil, D. Moldovan, H. L. Truong, and S. Dustdar, “SYBL: An Extensible Language for Controlling Elasticity in Cloud Applications,” in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 112–119.

[7] M. Zhang, S. Ali, T. Yue, and R. Norgren, “Uncertainty-wise evolution of test ready models,” *Information and Software Technology*, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2017.03.003. [Accessed: 26-Apr-2017].

[8] S. Ali, Y. Li, T. Yue, and M. Zhang, “Uncertainty-Wise and Time-Aware Test Case Prioritization with Multi-Objective Search,” *Technical report 2017-03, Simula Research Laboratory*. [Online]. Available: https://www.simula.no/publications/uncertainty-wise-and-time-aware-test-case-prioritization-multi-objective-search.

[9] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA, USA: MIT Press, 1994.

[10] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, 1992.

[11] S. Luke *et al.*, “ECJ: A Java-based Evolutionary Computation Research System.” [Online]. Available: https://cs.gmu.edu/~eclab/projects/ecj/.

[12] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, “Generating Test Data from OCL Constraints with Search Techniques,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 10, pp. 1376–1402, Oct. 2013.

[13] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A Field Guide to Genetic Programming*. Lulu.com, 2008.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[15] B. Liu, *Uncertainty Theory*. Springer, 2015.