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

Report on Uncertainty Testing Framework V.3

D 3.3

Project Acronym	U-TEST	**Grant Agreement Number**		H2020-ICT-2014-1. 645463	
Document Version	1.5	**Date**	2017-10-25	**Deliverable No.**	3.3
Contact Person	Luca Berardinelli	**Organisation**		TU Wien	
Phone	+43158801184904				

Document Version History

Version No.	Date	Change	Author(s)
0.1	2017-06-08	First empty template	Luca Berardinelli (TUW)
0.2	2017-07-27	Timetable updates/ Reviewer Updates	Luca Berardinelli (TUW)
0.3	2017-08-02	Timetable updates/ New TUW TOC Outline	Luca Berardinelli (TUW) Martin Schneider (FF)
0.3	2017-08-08	Updated reviewers	Luca Berardinelli (TUW)
0.5	2017-08-29	Updated TUW Sections	Luca Berardinelli (TUW)
0.6	2017-10-02	Updated Timeline	Luca Berardinelli (TUW)
1.0	2017-10-15	First Integrated Version	All
1.1	2017-10-18	Fixed Comments for TUW by Robert	Luca Berardinelli (TUW)
1.2	2017-10-19	Fixed Comments for FF by Robert	Martin Schneider (FF)
1.4	2017-10-23	Integrated all comments by reviewer 1	All
1.5	2017-10-24	Integrated all comments by all reviewers	All

Contributors

Name	Partner	Part Affected	Date
Luca Berardinelli	TUW	1, 2.1, 2.3, 3.2	2017-10-23
Martin Schneider	FF	1, 2.1, 2.2, 3.1	2017-10-23
Phu Nguyen	SRL	1, 2.1, 2.4. 3.3	2017-10-23
Man Zhang	SRL	1, 2.1, 2.4. 3.3	2017-10-23

Reviewers

Name	Partner	Part Affected	Date
Robert Magnusson (ok)	NMT	All	2017-10-16

Karmele Intxausti (ok)	IKL	All	2017-10-22

Table of Contents

Executive Summary

This deliverable presents the last achievements – the Uncertainty Testing Framework (UTF) V.3 - with model evolution algorithms and test strategies. It extends the works from V.2 of the Uncertainty Testing Framework and extends its focus to

- The encoding for the search-based algorithm used for uncertainty testing at the application level,
- Rule-based infrastructure level uncertainty evaluation and detection at design-time,
- New components for uncertainty-wise evolution, test case generation, minimization, prioritization, and execution at the integration level.

Keywords: Cyber-Physical Systems, Uncertainty Testing, Testing Framework, UML, Model Evolution

1 Introduction

This report describes the third version of the UTF. We report the last improvement and refinements of UTF.

1.1 Objectives of the Deliverable

The goal of this deliverable is to present improvements of the UTF. 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.3) 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.

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) and (D2.3)
- the UTF (D3.1, D3.2)

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.

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.

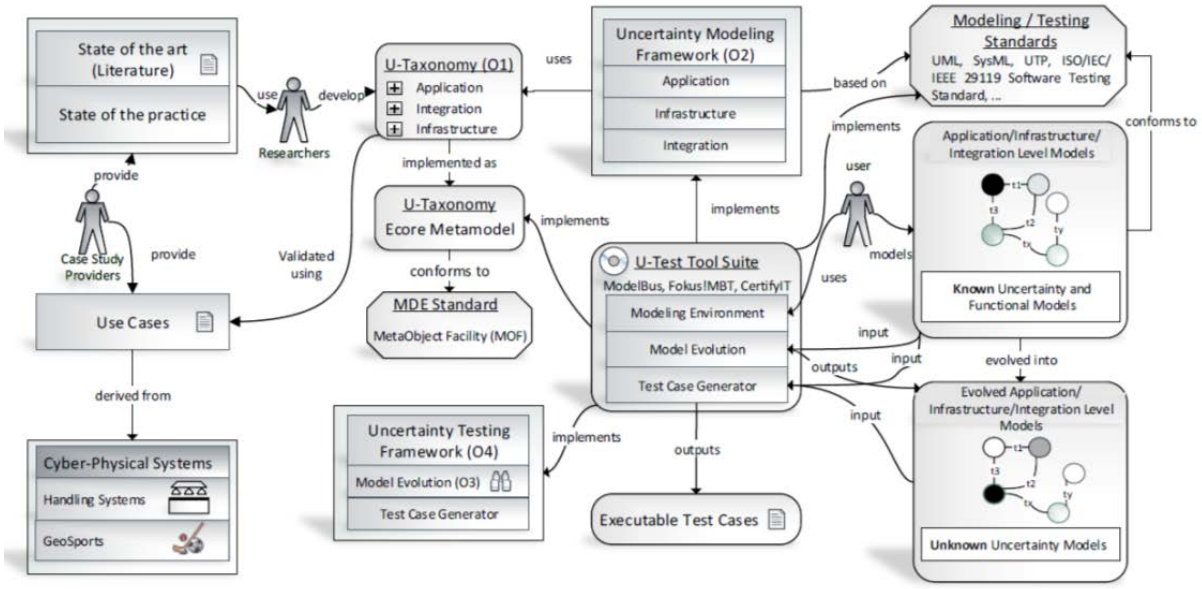

Figure 1. U-Test Workflow

Input:

- All previous Deliverables

Consumers of D3.3 (that are currently active)

- D4.2 (EGM, FF): Tool(s) Demonstrator
- D5.2 (FPX and ULMA): Report on test case executions
- D5.3 Validation with or without U-Test
- D5.4 Empirical Evaluation of Test Strategies
- 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. 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). TR8, providing more technical details for Section 2.4, is in

forms of a separate PDF file attached with this document. We summarize the whole deliverable and give our conclusions in Section 3.

2 Uncertainty Testing Framework

This section gives The UME approach assumes that any UML profile can be source of uncertainty. Indeed, different profiles can be applied to represent domain specific concepts to make the model ready for different engineering activities, as we do for infrastructure and uncertainty modelling purposes. Therefore, the number of applied profiles cannot be known beforehand. Moreover, each profile can evolve over time to reflect changes in the supported domain. This is particularly true for non-standard, user-defined UML profiles.

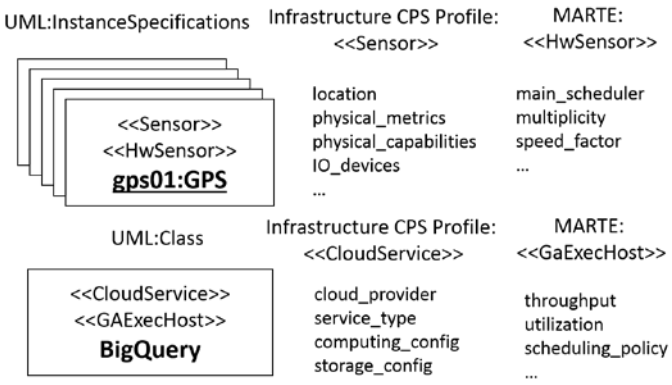

Figure 2 Example of infrastructural elements as UML Class and UML Instance Specification annotated with multiple stereotypes from different profiles.

In addition to the profiles and stereotypes devised for U-Test purposes, standard OMG profiles like UTP [3] and MARTE [4] are expected to be applied on the same UML model. To give an example, Figure 9 shows a hardware IoT element, the GeoSport GPS *Sensor*, and a software component, the BigQuery storage *CloudService*. The former can further be annotated with detailed hardware information via MARTE *HwSensor* stereotype and the latter can be annotated with *GaExecHost* for the sake of performance analyses (throughput, utilization, scheduling).

It is clear that the modelling power brought by UML and profiles can be overwhelming in terms of number of available stereotypes, properties and different annotation options determined by extended UML metaclasses) if not properly managed. For example, the MARTE profile defines around 360 stereotypes and more than 1000 properties. In this respect, stereotypes and their properties can be seen as potential source of uncertainties.

2.1.1 Uncertainty Modeling and Evaluation (UME) and supporting Tool (T4UME)

Figure 10 details illustrates our Uncertainty Modelling and Evaluation (UME) approach as part of a model- driven *Design activity*, which, in turn, is part of a wider engineering process consisting of many other model- driven activities (e.g., requirement specification, testing, analysis) that are expected to benefit from the artefacts generated at design-time.

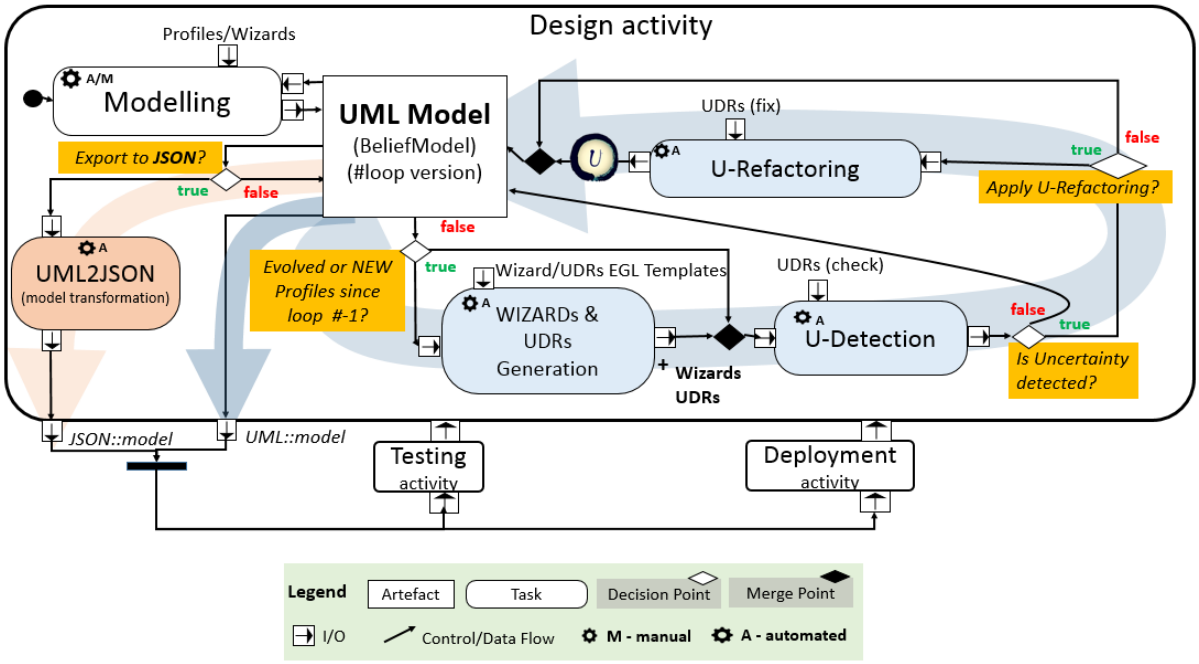

Figure 3 Detailed Design activity with UME methodology and T4UME tool support

UME aims at detecting and evaluating infrastructural uncertainties in the UML model caused by missing stereotype's property values, defined in any profile applied to the UML model representing the system under study.

UME (i) provides modelling facilities, i.e., wizards, to help modellers to represent IoT and cloud infrastructural components, and uncertainty detection rules (UDRs). Each UDR can detect new, potential uncertainties of IoT Cloud infrastructural elements, designed and represented via UML models, validates them against the U-Taxonomy, and, in case of positive detection, suggests and eventually executes uncertainty-wise model refactoring actions in order to make the model suitable for further engineering activities (e.g., model-based testing [5]).

UME approach includes three main tasks: Modelling, Uncertainty Detection (U-Detection), and Uncertainty Refactoring (U-Refactoring).Our *Tool for Uncertainty Modelling and Evaluation* (T4UME), an Eclipse-based tool built atop EPSILON [6], supports such activities.

Indeed, for each stereotype, properties can be defined to further detail IoT Cloud components configurations and uncertainty characteristics (e.g., period or persistent manifestation, causes, see [2]). Due to the evolving nature of information in UML model, profiles and uncertainty domains, UME and its tool T4UME are designed to suitably adapt to these changes.

UME and its T4UME provides model management facilities to help users to perform design-time tasks, namely *wizards* for modelling IoT and Cloud components and *uncertainty detection rules* (UDR) to detect (U-Detection) and to evolve (U-Refactoring) uncertainty-agnostic UML model with uncertainty-specific model elements and annotations. Both UME and T4UME are domain-independent and adapt its design-time support to uncertainty modelling and evaluation by (re-)generating profile-specific wizards and UDRs on new profile applications and/or evolution of already applied profiles.

In particular, we tailored UME/T4UME

- to support **modelling of infrastructural components** and to **detect potential infrastructural uncertainties at design-time**. For this purpose, we provide wizards and UDRs for uncertainty agnostic UML model to make them suitable to next model-based testing activities as required in U-Test project.
- to support **deployment** and **testing of infrastructural elements** [2] based on JSON representation. For this purpose, we provide a UML2JSON export functionality to generate JSON serialization of UML model elements for further processing by third-party tools.

It is worth noting that we expect continuous updates of wizards, UDRs, and JSON output formats due to the following reasons:

- Continuous updates to number, content and usage of UML profiles.
- Heterogeneity of specific system requirements (e.g., functional or non-functional),
- Heterogeneity of UML modelling guidelines proposed by different methodology and tool providers (e.g., EGM, FF)
- Different tool user (e.g., ULMA and FPX) expertise level and needs.

In this respect, we consider the Wizards and UDRs Generation step in Figure 10 very important and a peculiarity of the UME/T4UME pair.

The following section gives the necessary background to understand the model-driven design rationales of UME and model-driven technologies behind the implementation of T4UME.

2.1.2 UME and T4UME in detailed design and usage examples

In the following paragraphs, we describe how T4UME supports UME tasks. We detail the design of the T4UME software components by instantiating the design pattern in Figure 1. T4UME is, at the same time, a tool for MDE processes, like UME, and a tool developed following MDE principles.

2.1.2.1 Adoption of Model-Driven Engineering technologies in T4UME

Modelling, U-Detection, U-Refactoring and UML2JSON model-driven steps in Figure 10 are model management tasks implemented in T4UME.

T4UME relies on a set of technologies to implement UME model-driven tasks. In particular, it implements *model transformations* systematically define mappings to manipulate and integrate models and then realize model-driven engineering processes and supporting tool chains.

In a general sense, a model transformation is a program executed by a transformation engine which takes one or more models as input to produce one or more models as output as illustrated by the model transformation pattern [7] in Figure 11. It shows a recurrent pattern of model-driven tasks where a source model, specified using a source language, is transformed into a target model via *model transformations*, which maps source and target language concepts through executable transformation specifications.

We collected all the technologies mentioned in this section in a technology stack overview depicted in Figure 6. A UML model is an XML-based artefact whose content can be conveniently inspected by stakeholders via tree-based editors or displayed on an arbitrary number of diagrams. Model-driven tool, as T4UME, relies on ad-hoc APIs to query, validate, and modify the XML-based model content. These changes show up to modellers through capabilities of the chosen UML editor.

We select Papyrus [12], an open source, Java-based UML modelling environment to support modelling and profiling tasks. Papyrus, in turn, is developed on top of Eclipse Modelling Framework (EMF) [13]and EclipseUML2 . EMF provides, among others, modelling language definition and XML-based serialization capabilities. Eclipse- UML2 relies on EMF to define the UML language and to generate Java API for UML modelling tasks. Rational Software Architect (RSA) represent a viable alternative to Papyrus since it is built atop the same stack of Eclipse technologies (Eclipse EMF and Eclipse UML).

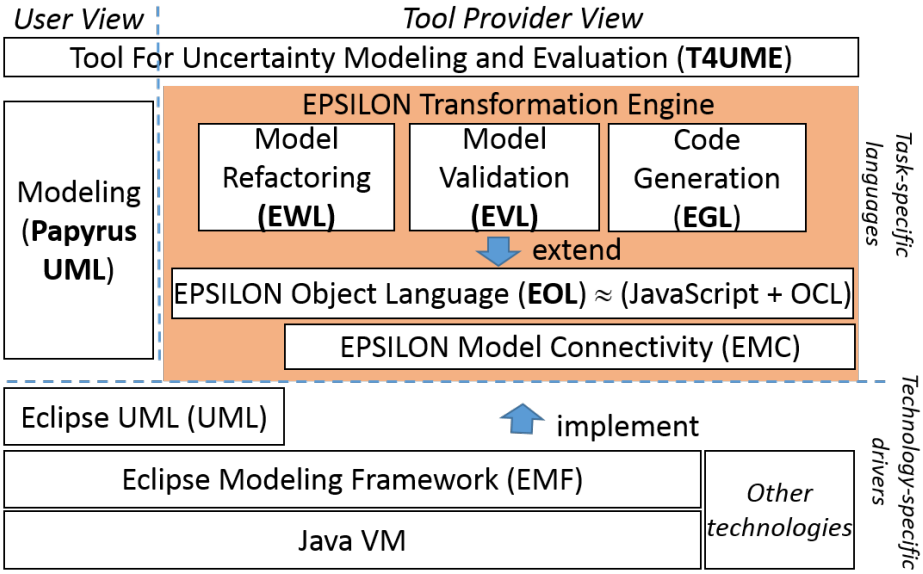

Figure 4 Technology stack of T4UME based on [12]

We then choose **Extensible Platform of Integrated Languages for mOdel maNagement framework (Epsilon)** [6] to implement the model management tasks provided by UME, i.e., wizards for modelling, U- Detection, and U-Refactoring.

Epsilon plays the role of transformation engine in Figure 11 and provides domain-specific languages (DSLs) to implement model management tasks. In particular, we used the following DSLs by Epsilon:

- Epsilon Object Language (**EOL**): An imperative model-oriented scripting language that combines theprocedural style of JavaScript with the powerful model querying capabilities of Object Constraint Language (OCL).
- Epsilon Wizard Language (**EWL**): A language tailored to interactive in-place model transformations on model elements selected by the user.
- Epsilon Validation Language (**EVL**): A model validation language that supports both intra and inter-model consistency checking.
- Epsilon Generation Language (**EGL**): A template-based model-to-text language for generating code, documentation and other textual artefacts from models.

EOL libraries can be imported and invoked by EWL, EVL, and EGL scripts and each DSLs can connect to external Java libraries. Moreover, both EWL and EVL provide out-of-the-box integration with EMF-based editors, like Papyrus [12].

2.1.2.2 Modelling

Figure 13 shows the models, languages, and tools involved in the UME Modelling task, and their role with respect to the design pattern in Figure 11.

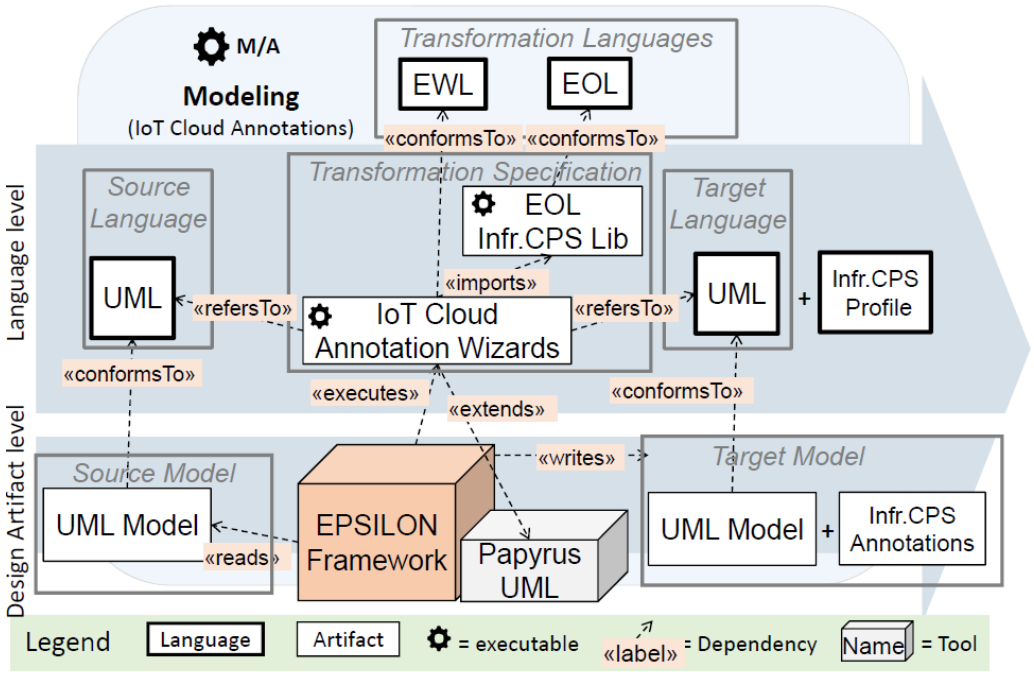

Figure 5. UME Modeling step: language and artifact perspective.

Modeling in UME is facilitated by *wizards*. In order to help developers with modeling, we currently provide different infrastructure-specific *wizards*.

```
wizard ApplyToClass {
guard : self.isTypeOf(Class)
and self.isStereotypeApplicable("Sensor")
and not self.isStereotypeApplied("Sensor")

title : "Apply Sensor to " + self.name

do {
var iot_cloud_profile = Profile.all.selectOne(e|e.name = "IoTCloud");
var stereotype = iot_cloud_profile.getSensorStereotype();
self.applyStereotype(stereotype);
 }
}
```

Listing 1 Wizard for Sensor Stereotype application to Class elements.

```
wizard InstantiateManyTimesSensor {
 guard : self.isKindOf(Class) and self.isStereotypeApplied("Sensor")
 title : "Generate Multiple Sensor instances."
 do {
  var plugin : new Native("at.ac.tuwien.dsg.t4ume");
  var ModelerAgent = plugin.getModelerAgent();

  var num_instances : Integer;
  var counter = 1;
  num_instances = UserInput.promptInteger("How many?");
  var package = self.getNearestPackage();
  while (counter <= num_instances) {
    var name = self.getName() + "_instance" + counter;
        var instance =
        ModelerAgent.createInstanceSpecification(package, self, name);
        instance.attachStereotype("Sensor");
        counter = counter + 1;
  }
 }
}
```

Listing 2 Wizard to instantiate Class with Sensor stereotype applied.

Each EWL artefact (.ewl) accesses a source UML model, modifies it, generate a target UML model with our InfrastructureCPS profile applied and new IoT/Cloud infrastructural elements have been represented.

T4UME currently provides wizards for **stereotype application to** and **instantiation of** infrastructural elements.A distinct **stereotype application wizard** (see Listing 1) is generated for each stereotype defined in the InfrastructureCPS profile. It applies the given stereotype to selected model element, if applicable. A distinct **instantiation wizard** (see Listing 2) is generated for each stereotype extending UML Class concept from which InstanceSpecfiications can be generated (e.g., to describe test configurations). Such wizards prompt an input window, asks users to insert the number of instances to be generated, and propagate Class annotation to the generated instances.

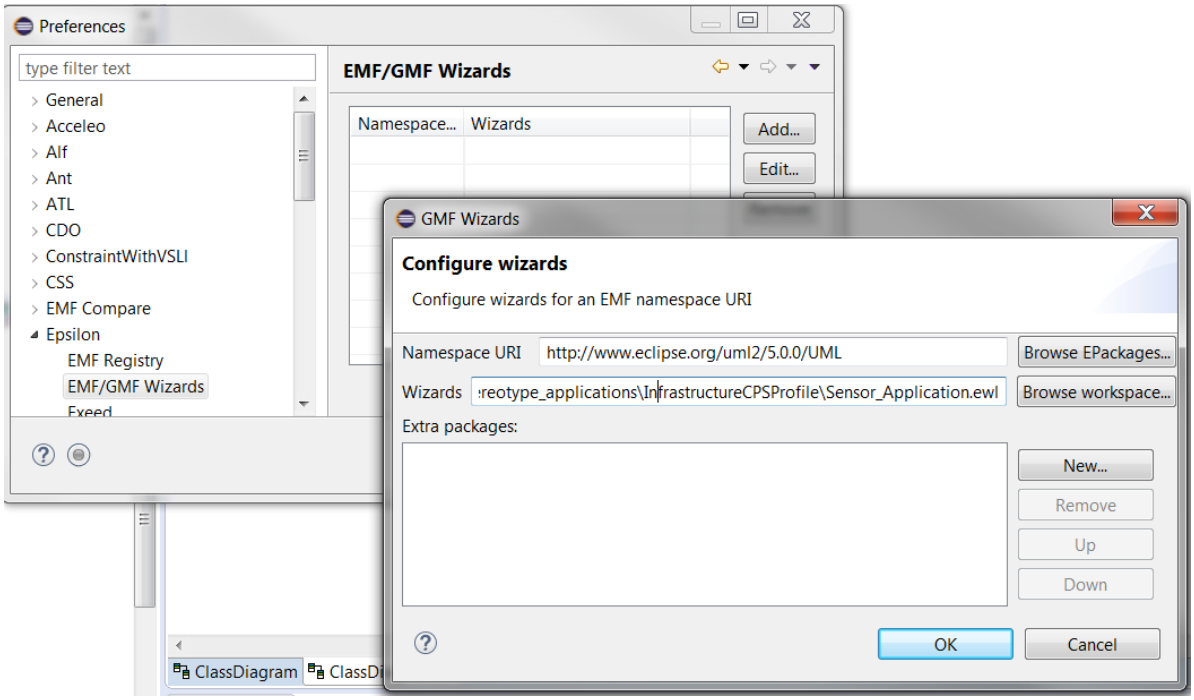

Figure 6 Extending EMF-based editors (e.g., Papyrus and RSA) with EWL wizards provided by T4UME.

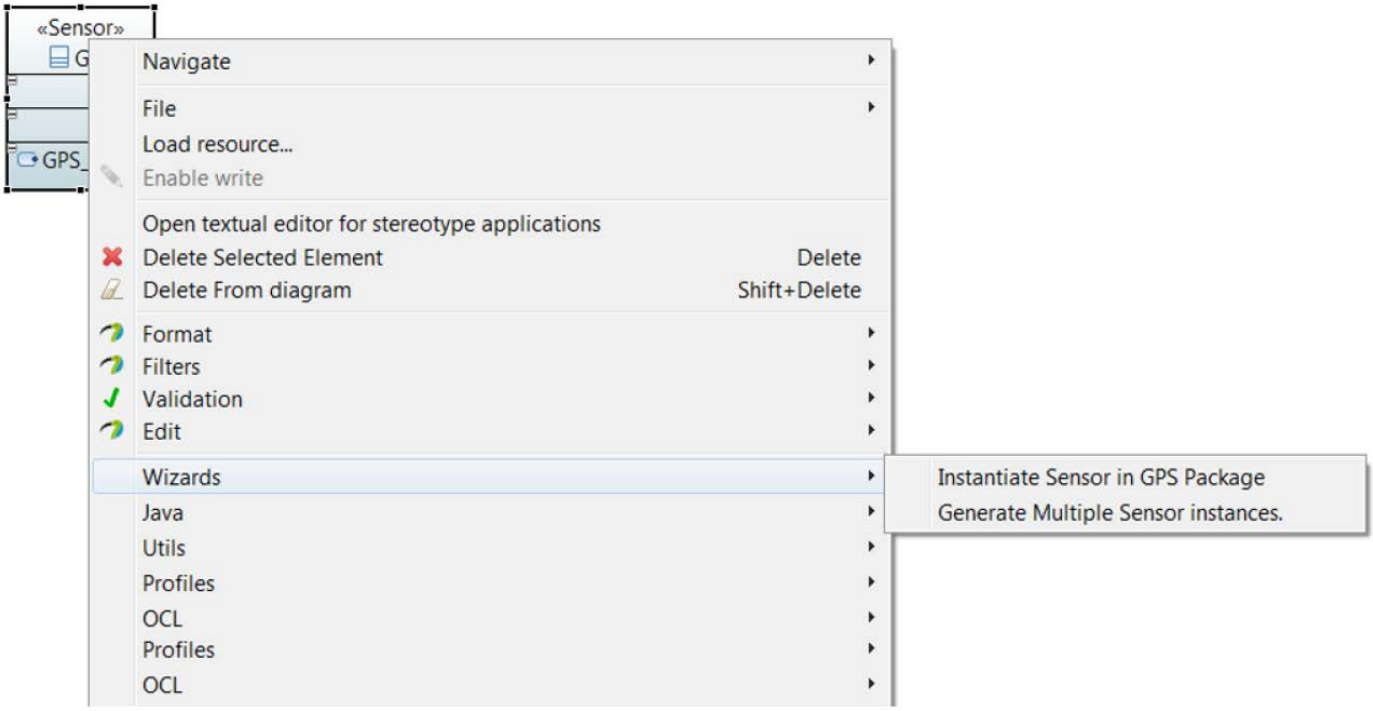

Figure 7 Invoking wizards on selected model elements in Papyrus

Thanks to the guard clause of EWL, (see Listings 1 and 2) each wizard can guide the user to modify the underlying UML model following proper modelling guidelines. Epsilon plays the role of the transformation engine. It executes the wizards and extends the functionalities of the select EMF-based editor.

Thanks to the native integration of Epsilon EWL with EMF-based editors, wizards can be loaded and used on demand. In this way, T4UME user can choose which wizards should be loaded (see Figure 14) to extend the modelling tool capabilities to help her during the modelling task, according to user's expertise. Once loaded, wizards can be invoked as shown in Figure 15.

2.1.2.3 Uncertainty Detection and Refactoring

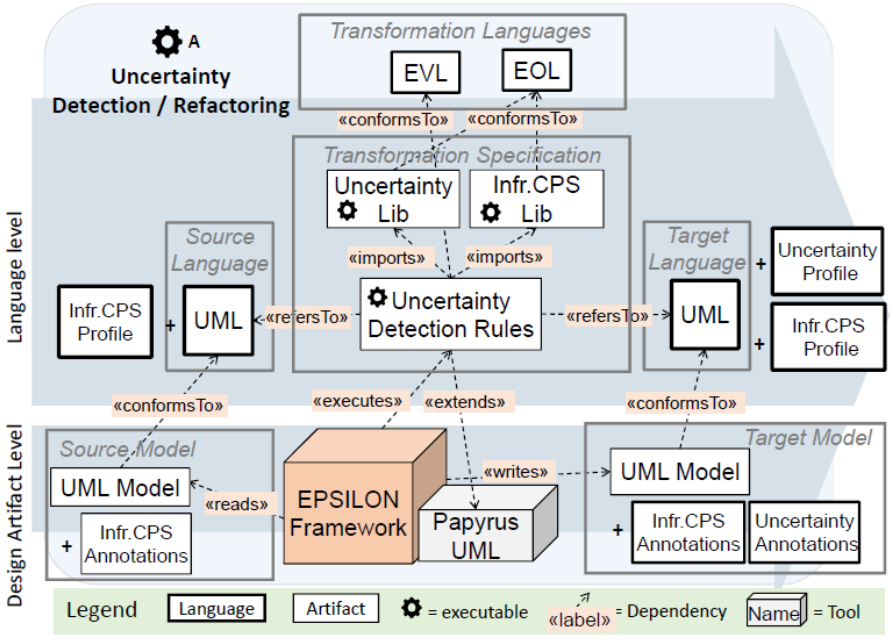

Figure 8 T4UME U-Detection and U-Refactoring steps: language and artifacts perspective.

The UML model obtained after the Modelling step is uncertainty agnostic. We expect that

- The UML model is annotated with stereotypes of our Infrastructure CPS profile, hopefully with the help of T4UME wizards, and
- Additional profiles can be applied on demand by users to satisfy specific needs.

If no new profiles applications and no updates occurs in applied UML profile specifications (i.e., in profile.uml files), the UME approach can proceed with U-Detection and U-Refactoring steps.

By completing the inner loop depicted in Figure 10, **UME performs an uncertainty evolution by executing UDRs on uncertainty-agnostic UML model**. Each UDR can detect and refactor a source UML model by inserting (infrastructural) uncertainties on (infrastructural) model elements.

Both U-Detection and U-Refactoring in UME are realized by UDRs. **UDRs and executed by T4UME on top of the Epsilon framework** (see Figure 16). In particular, U-Detection is implemented as a model validation task while U-Refactoring corresponds to a model refactoring task. Both validation and refactoring actions are specified within the same UDR artefact, which in turn is specified using the Epsilon EVL language.

```
context U-CloudService {
 critique cloudProviderProperty {
  check {
   return not self.cloudProvider.isEmpty();
  }
  message: "Missing cloudProvider value"
  fix {
   title : "Generate ElasticityUncertainty State"
   do {
    var elem = UMLUtil.getBaseElement(self);
    elem.createElasticityUncertaintyStateMachine();
    var elasticityUncertainty = coreProfile.getElasticityUncertaintyStereotype();
    elem.applyStereotype(elasticityUncertainty);
   }
  }
  fix{title:"Generate GovernanceUncertainty"...}
  fix{title:"Generate ActuationUncertainty"...}
  fix{title:"Generate StorageUncertainty"...}
  fix{title:"Generate ExecEnvUncertainty"...}
  fix{title:"Generate DataDeliveryUncertainty"...}
 }
 critique cloudProviderProperty {...}
 critique serviceTypeProperty {...}
 critique computingConfigsProperty {...}
 critique storageConfigsProperty {...}
 critique communicationConfigsProperty {...}
}
```

Listing 3 Excerpt of UDR for the CloudService stereotype (U-CloudService).

The first task of UME is uncertainty detection (*U-Detection*). It is performed by the check clauses of UDRs. We currently provide **41 UDRs for 41 stereotypes of our InfrastructureCPS profile** that together check the presence/absence of **203 stereotype properties**.

For example, The *U-CloudService* UDR in Listing 3 is an Epsilon EVL file that validates CloudService-annotated model elements representing service types (i.e., Class) or instances (i.e., InstanceSpecification). The U-CloudService UDR shows *critique* warnings (see Figure 17) to T4UME users by checking presence/absence (*isEmpty()*) of six different stereotype properties.

Each UDR artifact (.evl) can be launched from a launcher configuration that refers to the model to be validated (see the Project Explorer in Figure 16). The UDR accesses a source UML model annotated

with our InfrastructureCPS profile [1], selects elements annotated with stereotypes of the InfrastructureCPS profile and checks whether values have been set for each stereotype property. If not, the UDR rises a warning to the user assuming that potential uncertainties may be caused by such missing information. Thanks to the native integration of EPSILON with EMF-based editors, validation results can be shown on a Validation tab in Eclipse-based environments (see Figure 16).

By right clicking on a warning message, the user can choose among different refactoring actions, one for each infrastructural uncertainty stereotype defined in the InfrastructureUncertainty profile. Each refactoring action is implemented by a fix clause of UDR specifications in EVL (see Listing 3). By selecting one of the refactoring options, the user completes the U-Detection task (i.e., she recognized the presence of a particular uncertainty) and starts the U-Refactoring step by executing the fix clause of the UDR (see Listing 3).

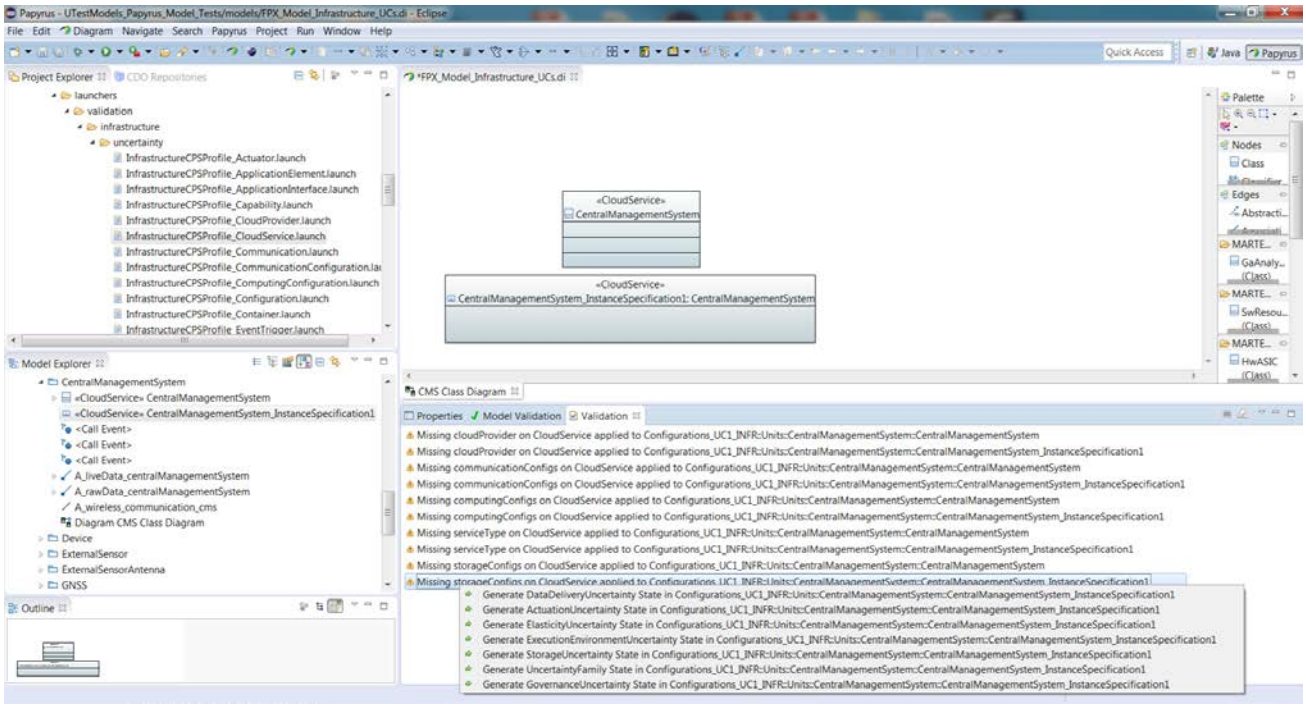

Figure 9 U-Detection step executed for CloudService stereotype applied on Class and InstanceSpecification model elements. Warnings and refactoring actions are displayed on Eclipse Validation tab.

The result of refactoring action is a target UML model with InfrastructureCPS and InfrastructureUncertainty profiles applied and additional uncertainty annotations on new and/or updated model elements generated as output of the U-Refactoring step. Figure 18 shows the effect of the fix clause Generate StorageUncertainty State on the Central Management System of the GeoSport

case study, previously annotated as a Cloud Service but without additional values for stereotype properties.

The result of refactoring action is a target UML model with InfrastructureCPS and InfrastructureUncertainty profiles applied and additional uncertainty annotations on new and/or updated model elements generated as output of the U-Refactoring step.

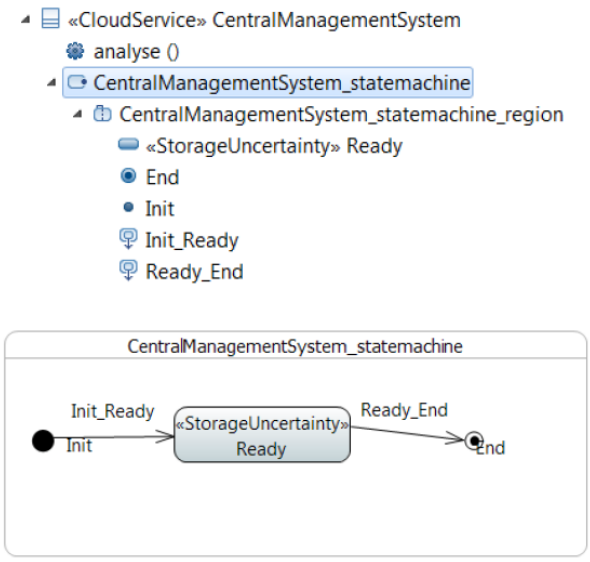

Figure 10 Results of refactoring actions: new annotated state machine for the sake of U-Test model-based testing methodologies.

2.1.2.4 Wizards and UDRs Generation

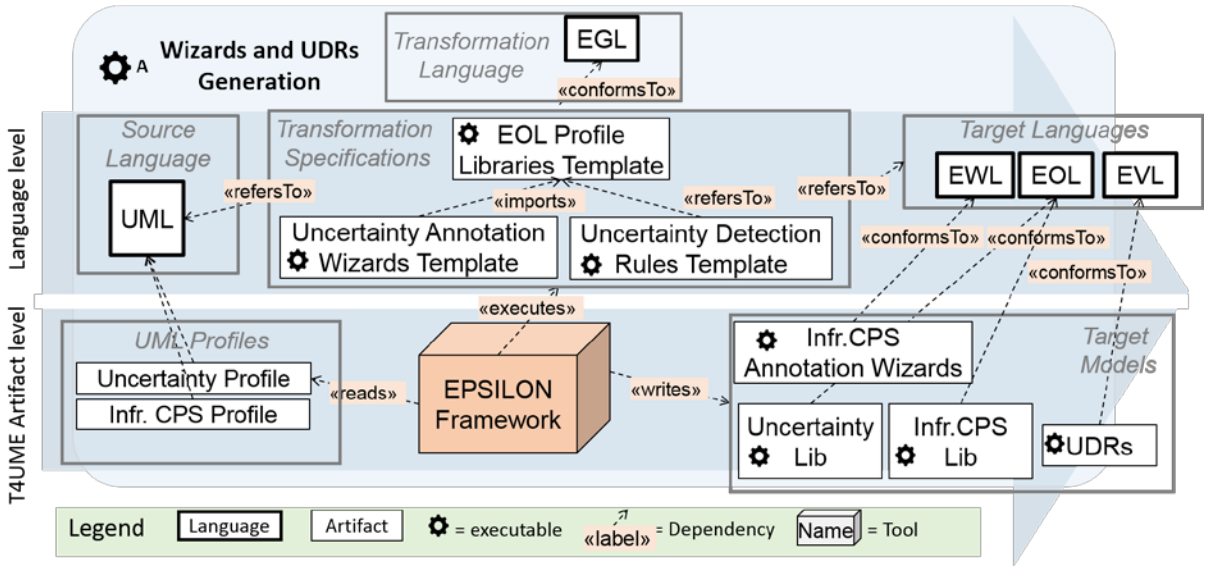

Figure 11 T4UME Wizards and UDRs Generation step: language and artifacts perspective

U-Detection and U-Refactoring steps can be customized to deal with different users' and MDE process needs by providing new and/or editing existing wizards and UDRs.

Indeed, we consider both UME and T4UME evolving methodology and tool that should adapt to a continuously evolving uncertainty domain, different modelling guidelines, and different users' profiles. The U-Taxonomy and related UML profiles devised in U-Test may evolve over time. In particular, continuous updates can affect non-standard and user-defined profiles. Moreover, different companies and/or research institutions with different needs could adopt UME and T4UME. Finally, a heterogeneous set of customers, from modelling experts, to developers and testers may need to use T4UME with different modelling needs

In order to face the methodology and tool customization challenge, we explicitly introduce an adaptation step in UME that generates new wizards and UDRs when new profiles are applied on the source UML model or updates happen on already applied profiles (see Figure 10).

Figure 19 shows the models, languages, and tools involved in the Wizards and UDRs Generation step, and their role with respect to the design pattern in Figure 11.

The **Wizards and UDRs Generation step is implemented as a higher-order transformation (HOT)** [7] using Epsilon technologies. This UME step differs from the other ones because it generates executable textual artefacts, instead of UML models, by accessing the content of UML profiles applied on the source UML model. These executable artefacts are the wizards (.ewl files), UDRs (.evl files), and EOL supporting libraries (.eol), that support the other three UME steps, i.e., Modelling, U-Detection, and U-Refactoring.

Both wizards and UDRs are textual artefacts. For this reason, we used Epsilon EGL to create wizard and UDR *templates* that are invoked for each stereotype. Listing 4 shows an EGL transformation rule that binds each UML stereotypes to an EGL template that generates an UDR specified in Epsilon EVL (.evl).

```
rule Stereotype2UDR
transform stereo : Stereotype {
 template : "UDR_Template.egl"
 target : stereo.getName() + "_UDR.evl"
}
```

Listing 4 Excerpt of EGL model to text transformation rule to generate UDRs as EVL files.

Listing 5 shows the EGL textual template that is invoked for each stereotype defined in InfrastructureCPS and InfrastructureUncertainty profiles. Static text is mixed with dynamic parts (in blue) where EGL code snippets access the stereotypes and attribute definitions to create check and fix clauses as shown in Listing 3

```
context [%=stereo.name%] {
[% for (attribute in stereo.AllAttributes()) { %]
 critique [%=attribute.name%]_Specification {
  check:  not self.[%=attribute.name%].isEmpty()
  message: "Missing" + [%=attribute.name%] +"value"
[%
var uncertanty_families =
stereo.getInfrastructureUProfile().getUTaxonomy();
for (family in uncertanty_families) {
%]
  fix {
   title : "Generate"+ [%=family.name%] +"State"
   do {
    var baseElement = UMLUtil.getBaseElement(self);
    baseElement.create[%=family.name%]StateMachine();
    ...
   }
  }
[%}%]
```

Listing 5 Excerpt of the UDR template.

2.1.2.5 Exporting UML to JSON

IoT and Cloud infrastructural resources modelled, communication protocols, expected uncertainties to be tested, etc., can be extracted from models into JSON-based descriptions. The key thing is to enable various tools to use the extracted information for different purposes. In this paper, the extracted information is used to determine test configuration and deployment. We implement the information extraction using the EPSILON framework.

Infrastructural resources modelled in UML model can be extracted into JSON-based descriptions. The key thing is to enable various tools to use the extracted information for different purposes. We implement the UML2JSON functionality using the Epsilon framework. Figure 19 instantiates the design

pattern for model-driven tasks introduced in Section 2.3.3.1 and shows the models, languages, and tools involved in the UML2JSON step.

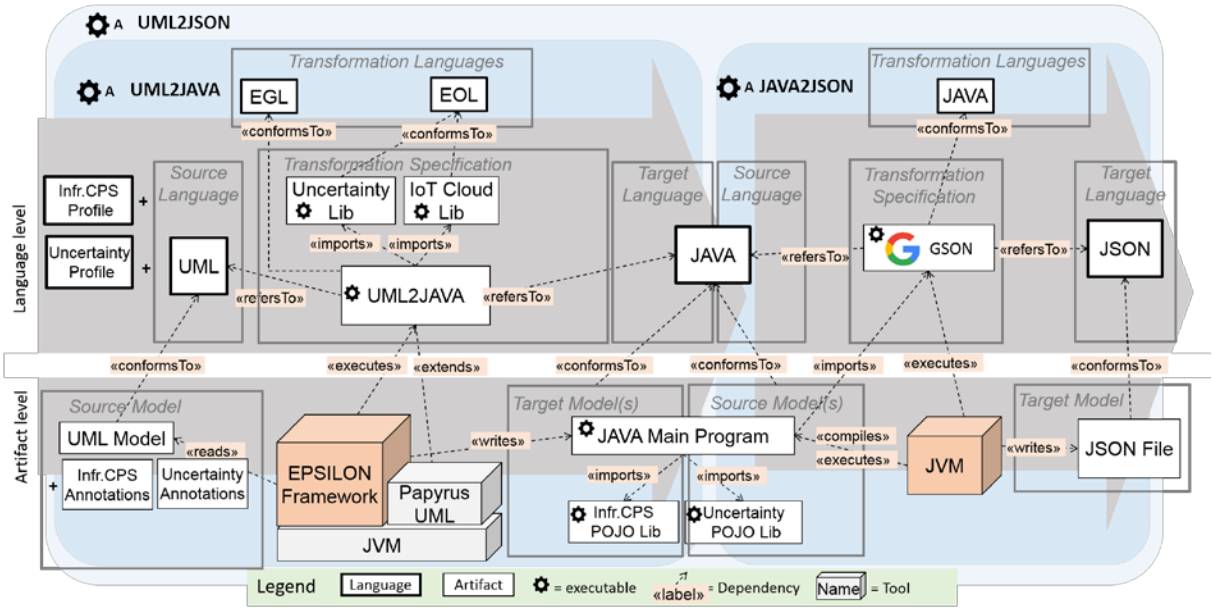

Figure 12 T4UME UML2JSON step: language and artifacts perspective

The UML2JSON step is split in two sub-steps: UML2Java and Java2JSON. The former takes an annotated UML Model as input and generates a Java main program that, in turn, invokes profile-specific Java APIs with getters/setters for each stereotype and property defined on profiles applied on the source UML model. The latter import the Google Gson library to generate the JSON representation of the Java objects instantiated during the Java main program execution. Each Java object, instantiated by invoking the profile-specific Java APIs, corresponds to the annotated UML model elements. Figure 20 shows the UML2JSON step in action to generate the JSON representation of an electricity sensor modelled through a UML class annotated by the VirtualSensor stereotype.

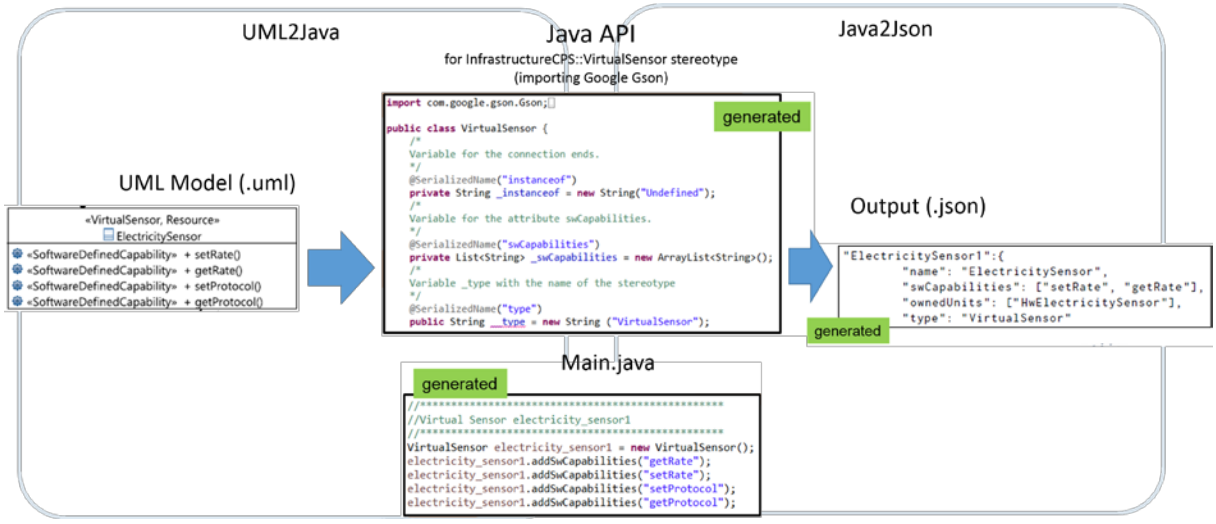

Figure 13 UML2JSON step example

The UML2Java step is implemented as a model to text transformation through EGL textual template while the Java2JSON step is realized by the Google Gson library.

We intend to use the UML2JSON capability of T4UME to bind infrastructural model elements represented in a UML model to concrete artefacts and testing utilities, usually stored public (e.g., Docker hub and Google Registry) and user-provided (e.g., Google Storage and Docker Registry) repositories. For the developer of infrastructure level,T4UME needs to connect different repositories to search suitable artifacts. Therefore, we develop a metadata service based on MongoDB for our artifacts. While artifacts can be stored in different repositories, the developer will need to provide metadata so that our configuration generation tool can search the right artifacts for the right underlying infrastructures. We also need to rely on resource information services to provide information about running instances of artefacts and infrastructural elements. For runtime information, tools like HINC [11] and SALSA [12] can be used.

2.1.2.5.1 Selecting Infrastructure-related artefacts

To enable the search of suitable infrastructure level artifacts, we impose a set of guidelines to describe artifact capabilities (there is no model that describing testing artifacts). After such artifacts are built, they are deposited into a repository and metadata will be stored into our services. In order to search the right artifacts, we have different metadata associated with artifacts. Such metadata will be searched by using information extracted as JSON description from the UML model (e.g., type of service units, and protocol supports). We use the following convention for metadata:

- *t4u/abstract_element/concrete_element:tag* where *t4u* is the name of the system under test, *abstract_element* indicates the type of infrastructural resource (e.g., VirtualSensor stereotype defined in [2]) and *concrete_element* indicates concrete types of infrastructural elements (e.g., model elements annotated with VirtualSensor stereotype), and *tag* is used to add new

information. For example, the string *t4u/VirtualSensor/ElectricitySensor:raspberrypi* indicates images and testing utilities of electricity sensor in Raspberry PI

- Each artifact has meta information about how to invoke and reconfigure it. For this, we use a convention: startup, shutdown, configure scripts with input (JSON) parameters. To be generic, we do not guarantee the correctness of these functions but we require these functions in order to reconfigure and start infrastructural elements. This requirement is conventional, widely used in practice, for dealing with configuration of software components, which can be implemented through REST (for Web service), gRCP (for RPC call-based objects), and shell scripts (for executable artifacts).

For example, a developer can i) develop a virtual sensor in Python/Java as an element of the CPS and ii) create a Docker file for the sensor. A Docker image can be built and deposited into the repository.

Listing 6 shows an example of a Docker file that bind a concrete MQTT broker to the corresponding cloud service, and make it available to developers on a repository.

```
$docker tag mqttsensor localhost:5000/t4u/
    cloudservice/mqttbroker:v01

$docker push localhost:5000/t4u/cloudservice/
    mqttbroker

$t4u_metadata add localhost:5000/t4u/cloudservice/
    mqttbroker:v0
```

Listing 6 Example of storing artifacts and metadata

From the extracted information, we connect to repositories of artifacts and we have different techniques to generate deployment configurations and deployment descriptions. Figure 20 shows the design. Configuration Generator will provide different possibilities of deployment configurations for elements of SUTs, e.g., whether a software sensor will be executed in a small virtual machine or not. After that, several drivers will be used to provide the detailed deployment descriptions, which are used to deploy several instances of infrastructural elements for testing.

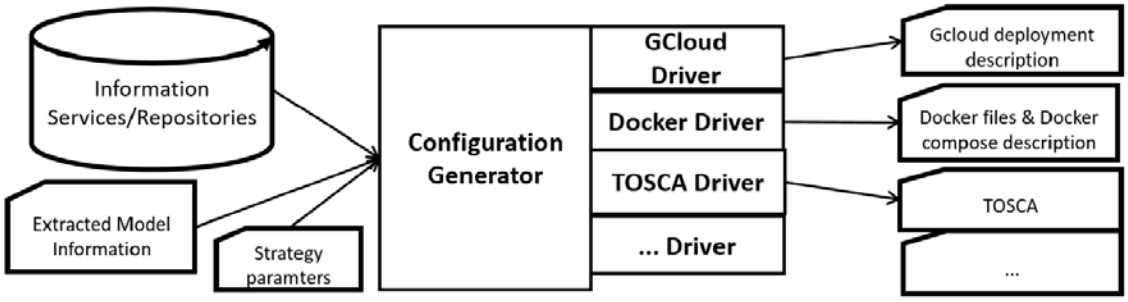

Figure 14 Deployment configuration and description generator

For example, a deployment description can include how many sensors, virtual machines, message brokers, cloud data services, etc., should be deployed and to where (e.g., local cloud or Google). SALSA, Docker tools, and cloud-specific tools (e.g., Google gcloud) can be used for deployment. We provide further information in [7].

2.1.3 How to adopt UME and T4UME in UTFv3

The UME methodology (Figure 10) focuses on uncertainties caused by missing values of properties of stereotypes defined in profiles applied to the UML model.

The UME approach is domain-independent and adapt its design-time tasks (modelling, U-Detection, and U- Refactoring) to different domains and engineering activities by generating new wizards and UDRs depending on stereotypes and properties defined in applied profiles. To show the feasibility of our approach, we provide generate wizards, for stereotype annotation and instantiation, and UDRs for detection of infrastructure uncertainty families (see D2.3).

T4UME provides model management facilities to help users to perform modelling, with the help of wizards, and uncertainty evolution, via UDRs. Each UDR is capable to detect (U-Detection) and to evolve (U-Refactoring) uncertainty-agnostic UML model with uncertainty-specific model elements and annotations.

Depending on the nature of annotations on UML Models brought by stereotypes and properties, the UME can be carried out:

- Before test case generation and execution to generate UML StateMachines for the sake of uncertainty-wise test case generation.
- After test case execution to detect uncertainty and carried out uncertainty-wise refactoring based on feedback from test result.

Indeed, we do not make assumptions on the origins of stereotype property values that:

- Can be directly annotated by modelers.
- Can be generated by methodology-specific steps (e.g., model evolution algorithms proposed by FF and SRL in Sections 2.2 and 2.4, respectively).
- Store test case execution results[1].
- Can be measured at runtime (e.g., execution time and number of invocations of system functionalities) and then annotated back to the UML model[2].

[1] UTP2 defines the TestLog stereotype to capture information on the execution of a test case.

[2] MARTE provides the stereotype property source of type SourceKind to distinguish different origins of non-functional properties. Predefined kind of sources for values are estimated, calculated, required, and measured.

In all the aforementioned cases, we assume that the UML model continuously evolves together with its applied profiles, stereotypes and properties. In this case, a new UME loop can detect, at design-time, new potential uncertainties caused by missing information on evolved model elements.

In this deliverable, we start customizing the UME/T4UME pair to support uncertainty detection and evolution at the infrastructure level and JSON serialization for the sake of deployment and provisioning of IoT and cloud infrastructural elements. Indeed, we expect that different UME users can further adapt T4UME to their modelling and uncertainty evolution needs

- By implementing wizards and UDRs with custom detection and refactoring actions (i.e., the corresponding check and fix clauses of EVL specifications, respectively) for the sake of uncertainty evolution (as those presented by FF and SRL in Sections 2.2 and 2.4).
- By updating the UML2Java sub-step (i.e., the underlying EGL textual template) while preserving the correctness of the JSON output guaranteed by the Google Gson library.

We adopted the Eclipse Epsilon framework[3] to implement the UME functionalities. The Epsilon framework can be smoothly integrated with EMF-based editors, as Papyrus (used by FF) and Rational Software Architect (used by EGM) are. Therefore, wizards and UDRs can be generated automatically by T4UME for both modelling tools and further customization can be implemented by combining Epsilon domain-specific languages and Java routines invoking the Eclipse UML API provided by external tools[4].

2.2 Uncertainty Testing at Integration Level

This section presents the overview of the work related to UTF at the integration level. UTF at the integration level includes the following five components as shown in Figure 20:

1) Uncertainty-wise Model Evolution,
2) Uncertainty-wise Test Case Generation,
3) Uncertainty-wise Test Case Minimization,
4) Uncertainty-wise Test Case Prioritization and
5) Uncertainty-wise Test Execution.

In this section, we provide the overview of each these components and updates as compared to the previous versions of WP3 deliverables.

As shown in Figure 20, the initial input of the UTF at the Integration Level is *belief test ready models (BMs)*. These BMs are the output of the UncerTum (C0). We presented UncerTum in WP2 deliverables, i.e., D2.1 to D2.3. Figure 20 provides the overall workflow of the UTF at the integration level, whereas below we briefly describe the five components together with which sections and technical reports provide complete details. In addition, we also summarise the updates as compared with D3.2 and D3.1:

[3] https://www.eclipse.org/epsilon/

[4] https://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.tools

1. Belief test-ready models are evolved based on the uncertainty-wise model evolution component (C1). There are two solutions developed at the integration level: 1) UncerTolve (presented in the D3.2 and submitted as a **TR4.1pdf**), 2) UncerPlore (Section 2.4.1.2);
2. The uncertainty-wise test case generation component (C2) takes (initial or evolved) belief test-ready models as input and generate abstract/executable test cases (Section 2.4.2);
3. By taking generated abstract test cases as input, the uncertainty-wise test case minimisation component (C3) can be optionally used to minimise the number of abstract test cases with configurable four test minimization problems using multi-objective search algoriths (Section 2.4.2);
4. The uncertainty-wise prioritisation component (C4) takes abstract test cases and test results as input and prioritises the sequence to execute test cases cost-effectively with multi-objective search algorithms (Section 2.4.2);
5. The uncertainty-wise test case execution component (C5) takes the (minimised/prioritised) test cases as input to execute on test infrastructure, which outputs the test results with the occurrence of uncertainties (Section 2.4.3).

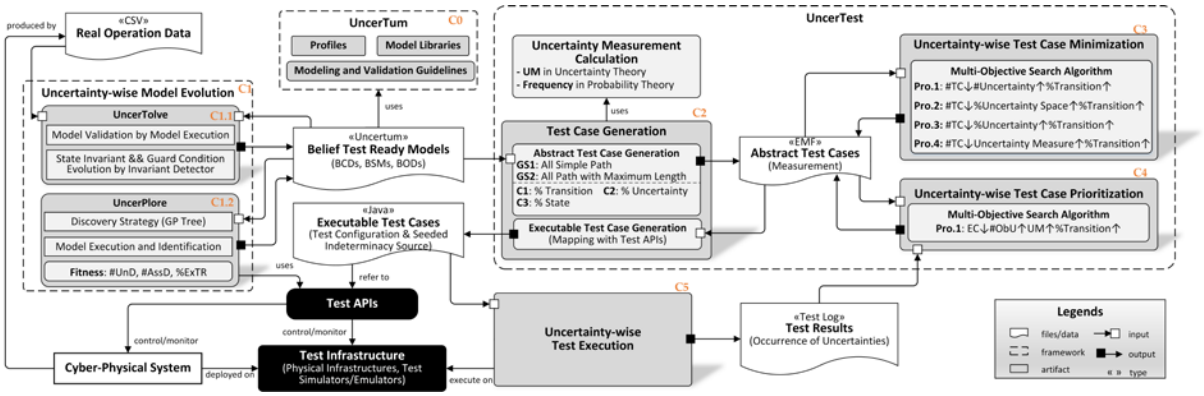

Figure 15. Overview of Uncertainty Testing Framework at Integration Level

2.2.1 Updates on Test Ready Model Evolution

This section presents the uncertainty-wise model evolution, which has two distinct methodologies: UncerTolve and UncerPlore.

2.2.1.1 UncerTolve

There is no update on UncerTolve. The final version was submitted in D3.2. We published the work in the Information and Software Technology Journal and is available with open access:

Man Zhang, Shaukat Ali, Tao Yue and Roland Norgren, Uncertainty-Wise Evolution of Test Ready Models, Information and Software Technology Journal, Volume 87, July 2017, Pages 140-159, Open Access at http://www.sciencedirect.com/science/article/pii/S0950584917302161

2.2.1.2 UncerPlore

We present the methodology to evolve belief state machine as shown in Figure 21. The methodolgoy has two parts: 1) discovering new uncertainties and new associations between uncertainty and indeterminacy sources with Genetic Programing—one of search algorithms (Section 2.4.1.2.2), 2) updating the objective uncertainty measurements for the known uncertainties and newly discovered uncertainties (Section 2.4.1.2.3).

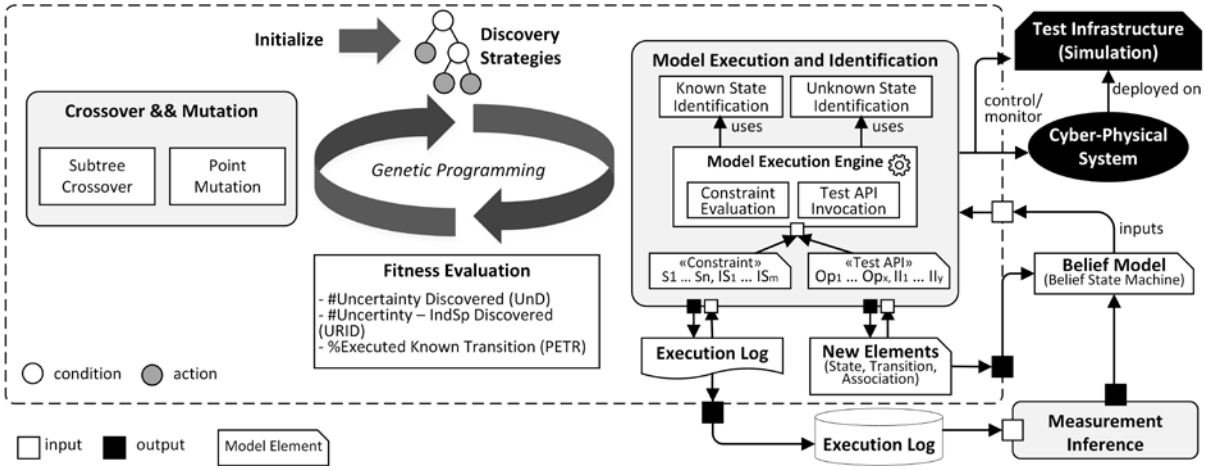

Figure 16. Overview of UncerPlore

2.2.1.2.1.1 *Definitions*

This section presents the definitions for uncertainty-related concepts. These definitions are necessary to explain UncerPlore.

- ***Belief State Machine*** (BSM) is a state machine developed with UncerTum, i.e., Uncertainty Modelling Framework at the integration level. It consists of a set of UML profiles and modelling guidelines as described in [14] [15].
- ***Uncertainty*** (*U*) of (st_x, tr_y, st_z) is a situation whereby the belief agent does not have full confidence that the source state, i.e., st_x transits to the target state, i.e., st_z with the tr_y transition in a BSM [15].
- ***Uncertainty Measure*** (UM) is a way to measure uncertainty with Uncertainty theory [5]. The value is a belief degree ranging from 0 to 1 represented as: (st_x, tr_y, st_z) = $\mathcal{M}\{(st_x, tr_y, st_z)\}$ [15].
- ***Indeterminacy Source*** (IndS) describes the uncertainties in the physical environment that leads to the observed uncertainties in a CPS [15].
- ***Indeterminacy Specification*** (IndSp) describes the condition that must be true for an indeterminacy source to occur [15].

2.2.1.2.2 Evolving Belief Test Ready Models using Genetic Programming

This section describes the methodology to evolve test ready models using Genetic Programming (GP) [16] [17] and benefiting from runtime test ready model execution on the dedicated test infrastructure (physical infrastructure or Simulators/Emulators) to evaluate and identify runtime state of the system

under test. Note that NMT and ULMA provide the test infrastructures including detailed simulators/emulators for physical environment (ULMA) and physical infrastructure, i.e., test rigs (NMT).

2.2.1.2.2.1 *ExBSM*

To execute model elements defined in the belief model on a CPS, we convert belief state machines to the ExBSM, i.e., executable belief state machine. This enables to evaluate constraints and invoke Test APIs as shown in Figure 22.

In Figure 22, the white concepts are the elements derived from belief state machines. Table 2 presents the definitions of concepts. The black concept (EObject) associated with *Model E&I* indicates that the element defines the test configuration generated by the UncerTest. The detailed *Model E&I* are described in Section 2.4.1.2.2.3.

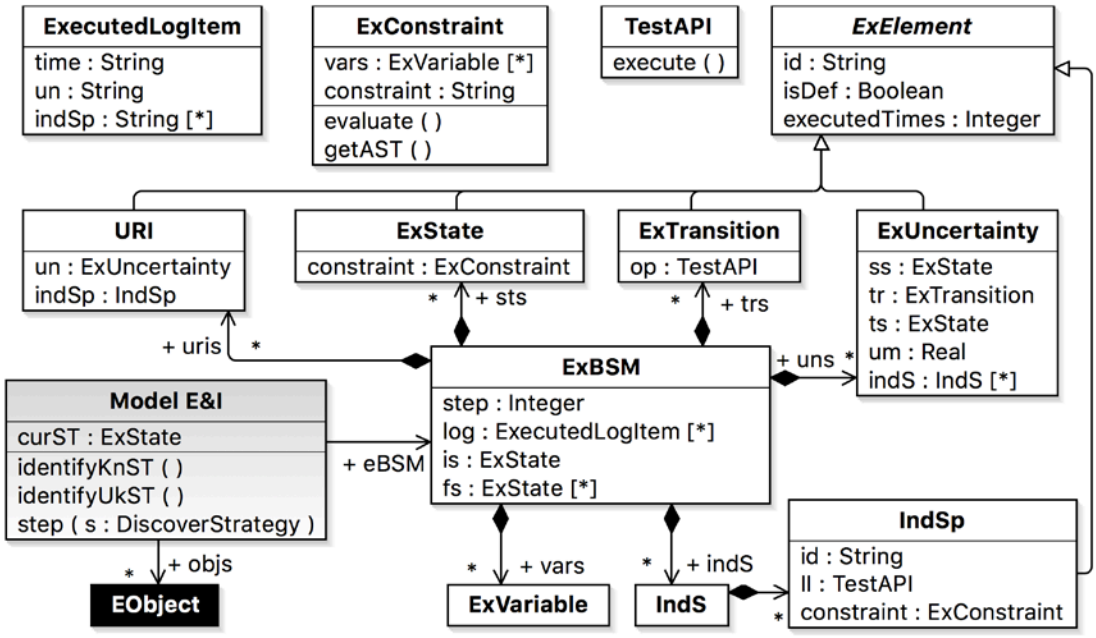

Figure 17. Conceptual Model of Model Execution and Identification of Model Elements and Uncertainties

Table 1. Definitions of ExBSM

Concept Name	Definitions
ExBSM	This concept maps with belief state machine («BeliefElement» state machine) defined in the belief model. *is* indicates the initial state. *fs* indicates the set of final states.
ExState	This concept maps with the state defined in the belief state machine. Also, the state invariant for each state is converted to *ExConstraint*.
ExTransition	This concept maps with the transition defined in the belief state machine. Also, we map the trigger in the transition to the specific Test API.
ExUncertainty	This concept maps with the uncertainty defined in the state stereotyped «BeliefElement».
URI	URI (Uncertainty Related to Indeterminacy Specification) is the occurrence association between uncertainty and indeterminacy specification defined in the *relatedIndSp:Uncertainty*.

IndS	IndS is the indeterminacy source which maps to the element stereotyped as «indeterminacySource» in belief state machine.
IndSp	IndSp is the indeterminacy specification that is constraint defined in the belief model stereotyped «IndeterminacySpecification». The constraint is converted into *ExConstraint* associated with this IndSp.
ExConstraint	This concept maps with the state invariant defined on the state and indeterminacy defined in the indeterminacy source.
ExVariable	This concept maps with the *PropertyCallExp* defined in the constraint. Also, we convert each constraint into abstract syntax tree.
ExecutedLogItem	It records the process of execution. *un* indicates the occurrence of uncertainty. *time* indicates the time point when *un* is executed. *indSp* indicates a set of occurrence of indeterminacy specification at that specific time point.
ExElement	It is used to describe the execution situation. *isDef* indicates if the element is defined in the belief state machine. *executedTimes* is used to define the times that element is executed.

2.2.1.2.2.2 *Discovery Strategy*

Discovery Strategy describes a strategy to execute test ready models with the ultimate aim of discovering uncertainties not specified in the test ready models.

We represent a discovery strategy as a tree. We defined two types of nodes in the discovery strategy as shown in Table 3. The *Action* node is the leaf node used to select a transition or indeterminacy source. The *Condition* node is the internal node to evaluate the current execution situation. We defined the condition node from the following four perspectives,

1) the current identified state,
2) the possible next transitions,
3) the coverage of executed transitions,
4) the coverage of executed URI (Relationship between Uncertainty and Indeterminacy Source) defined in Table 2.

Also, the basic operators, i.e., *IfElse* and *Sequence* are used to connect *Action* and *Condition*.

Table 2. Definitions of Discovery Strategies

Type	Name	Definition
Action	*SelectLessEx*	This action is used to select the next transition that is executed less than other known transitions.
	SelectLowUM	This action is used to select the next transition whose uncertainty measure is less than other known ones.
	SelectHighUM	This action is used to select the next transition whose uncertainty measure is more than other known ones.
	SelectExcluded	This action is used to select the next transition that are not any possible known transitions whose source is current state.
	IntroduceLessExIndS	This action is used to introduce an indeterminacy source, which is introduced less number of times than the other related indeterminacy sources.
	IntroduceExcluded	This action is used to introduce an indeterminacy source that are not any related indeterminacy source with current state.
	DisableIndS	This action is used to disable related indeterminacy source.
Condition	*hasOneKnNext*	This condition is used to evaluate if the number of possible next transition is only one.

	hasUncertainties	This condition is used to evaluate if any possible next transition will lead to occurrence of an uncertainty.
	currentIsNewState	This condition is used to evaluate if the current state is a newly discovered state.
	isTRCoverage25	This condition is used to evaluate if the coverage of executed transitions is between 0% and 25%.
	isTRCoverage50	This condition is used to evaluate if the coverage of executed transitions is between 25% and 50%.
	isTRCoverage75	This condition is used to evaluate if the coverage of executed transitions is between 50% and 75%.
	isTRCoverage100	This condition is used to evaluate if the coverage of executed transitions is between 75% and 100%.
	isURICoverage25	This condition is used to evaluate if the coverage of the occurrence of uncertainties with known indeterminacy sources is between 0% and 25%.
	isURICoverage50	This condition is used to evaluate if the coverage of the occurrence of uncertainties with known indeterminacy source is between 25% and 50%.
	isURICoverage75	This condition is used to evaluate if the coverage of the occurrence of uncertainties with known indeterminacy sources is between 50% and 75%.
	isURICoverage100	This condition is used to evaluate if the coverage of the occurrence of uncertainties with known indeterminacy sources is between 75% and 100%.
Basic	*IfElse*	This operation is used to manage conditional construct.
	Sequence	This operation is used to manage sequence construct.

2.2.1.2.2.3 *Model Execution and Identification of Model Elements and Uncertainties*

Figure 23 shows the process of executing belief state machines and identifying the new model elements and uncertainties. Each execution is started from the initial state defined in *ExBSM*. Based on the current execution, *Model E&I* executes the specific transition (*ExTransition*) and indeterminacy source (*IndSp*) determined by the *Discovery Strategy*. Afterwards, *Model E&I* proceeds with identifying the known state (the algorithm in Figure 24). Based on the number of identified states, there are three options:

1) there is no known state that is identified (nds = 0). In this case, we go to the process of identifying unknown state (the algorithm in Figure 25);
2) there are more than one known states identified (nds >1). In this case, we create a new state that combines all state invariants using *and* logical operator;
3) there is only one known state identified. In this case, we set the current state as the identified one.

After the current state is identified, *Identify IndeterminacySource* (Figure 23, C) is used to identify the current indeterminacy specification (the algorithm in Figure 26). Execution log is updated followed by the identification of current state and indeterminacy specifications. Two conditions terminate the execution of the evaluation of one generation of *Discovery Strategy*:

1) the executed steps reach the maximum of steps (step = max);
2) there is no specific transition generated by discovery strategy (isTerminate).

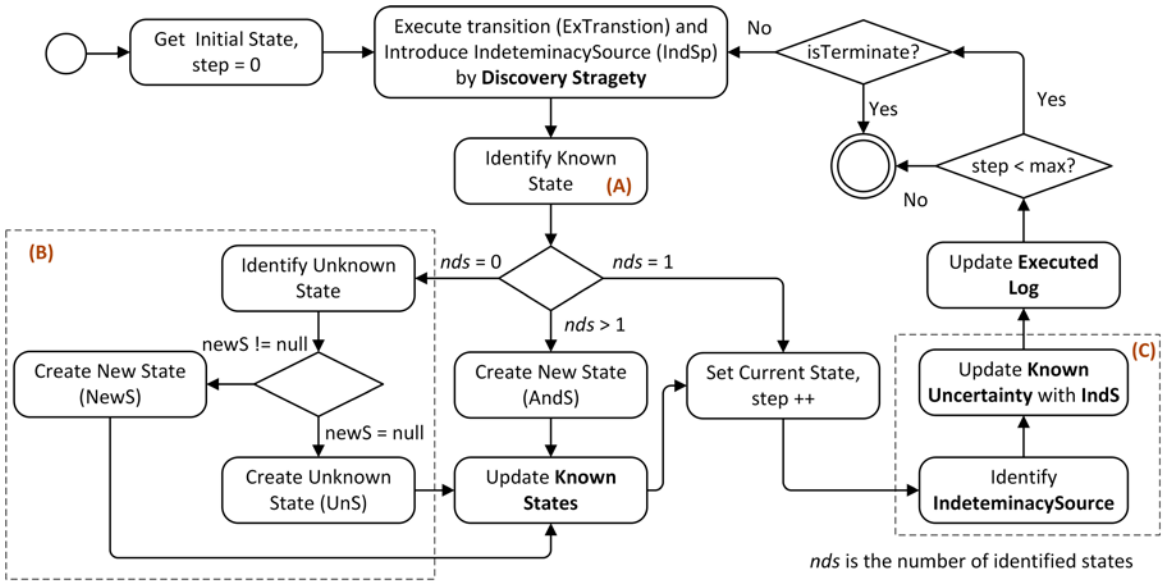

Figure 18. Process of Model Execution using Discovery Strategy

Identify Known State

```
input     knSTs:ExState[]
output    curSTs:List<ExState>
1   n ←len(knSTs)
2   for i ← 1 to n
3     if knSTs[i].constraint.evaluate()
4       curSTs.add(knSTs[i])
```

Figure 19. The Algorithm *Identify Known State* (Figure 20, A)

```
Identify Unknown State
input       spST:ExState
output      newST:ExState
1   newCons : List<ExConstraint>
2   //get variables based on specified state
3   vars = getVars(spST)
4   for var ← vars
5     //get constraints related to this var
6     cons = getConstraints(var)
7     for con ← cons
8       if con.evaluate()
9         newCons.add(con)
10  if len(newCons) = 0
11    for var ← vars
12      cons = getConstraints(var)
13      for con ← cons
14        //get set of constrains based on Table 4
15        wcons = weak(con)
16        //stop to find weaken constraint when one is satisfied
17        FW: for wcon ← wcons
18          if wcon.evaluate()
19            newCons.add(wcon)
20            break FW
21    if len(newCons) = 0
22      newST = createUkST()
23    else newST = createAndST(newCons)
24  else
25    newST=createAndST(newCons)
```

Figure 20. The Algorithm *Identify Unknown State* (Figure 20, B)

```
Identify IndeterminacySource
input       u:ExUncertainty, indsps:IndSp[]
output      newURI:List<URI>
1   newCons : List<ExConstraint>
2   //get variables based on specified state
3   vars = getVars(spST)
4   for indsp ← indsps
5     if indsp.constraint.evaluate() and not uoa.indsp.contain(indsp)
6       newURI.add(new URI(uoa, indsp))
```

Figure 21. The Algorithm *Identify IndeterminacySource* (Figure 20, C)

Table 4 describes a set of rules to weaken the constraint.

Table 3. Weaken Rules for the Operations

Operation	Expression	Weaken Rules
forAll	**var**->forAll(exp)	**var**->exists(exp)
		var->select(exp)->size()=0
exists	-	-
one	**var**->one(exp)	**var**->select(exp)->size()>1
		var->select(exp)->size()=0
select	-	
reject	-	
includesAll	**varX**->includesAll(**varY**)	**varY**->select(y|**varX**->includes(y))->size()>0
		varX->excludesAll(**varY**)
includes	-	

excludesAll	**varX**->includesAll(**varY**)	**varY**->select(y|**varX**->excludes(y))->size()>0
		varX->includesAll(**varY**)
excludes	-	-
isEmpty	-	-
notEmpty	-	-
not	-	-
= (numeric)	**var**=NumericLiteralExp	**var** < NumericLiteralExp
		var > NumericLiteralExp
	varX = **varY**	**varX** > **varY**
		varX < **varY**
= (String, Boolean)	-	-
<>	-	-
>	**var** > NumericLiteralExp	**var** < NumericLiteralExp
		var = NumericLiteralExp
	varX > **varY**	**varX** < **varY**
		varX = **varY**
<	**var** < NumericLiteralExp	**var** > NumericLiteralExp
		var = NumericLiteralExp
	varX > **varY**	**varX** > **varY**
		varX = **varY**
>=	-	-
<=	-	-

2.2.1.2.2.4 *Genetic Programming Problem*

Table 5 shows the settings of a GP in UncerPlore.

Table 4. The Setting of GP in UncerPlore

Name	Definition
Algorithm	GA
Terminal Set	*Action*
Function Set	*Condition, Basic*
Control Parameter	subtree crossover [16] [17], rate = 0.9
	point mutation [16] [17], rate = 0.01
	Population=100
	Max Generation=100
Termination Criterion	*step = max or isTerminate*
Number of runs	10

After execution, we collect a set of uncertainties related information as shown in Table 6.

Table 5. The Calculation of Uncertainty-Related Data

	Name	Definition
#UnS	Number of uncertainties specified	context ExBSM

		`def: getUnS:Integer = ExUncertainty.allInstances()->select(isDef)->size()`
#UnD	Number of uncertainties discovered	`context ExBSM` `def: getUnD:Integer = ExUncertainty.allInstances()->select(not isDef and um = 0.0)->size()`
#UriS	Number of URI specified	`context ExBSM` `def: getUriD:Integer = URI.allInstances()->select(isDef)->size()`
#UriD	Number of URI discovered	`context ExBSM` `def: getUriD:Integer = URI.allInstances()->select(not isDef)->size()`

PETR	Percentage of Executed Transition Coverage	*PETR* = $\frac{Number\ of\ specified\ transitions\ that\ are\ executed}{Number\ of\ Specified\ transitions}$, which can be defined as
		`context ExBSM`
		`def: getPETR:Double = (ExTransition.allInstances()->select(executedTimes > 0 and isDef)->size() * 1.0)/ExTransition.allInstances()->select(isDef)->size()`

The objective is to discover the uncertainty and its related indeterminacy source as many as possible under more transitions are executed.

Since the range of *#UnD* and *#UriS* is $[0, \infty)$, we use *Inverse tan function* (arctan) to normalize data [16] between 0 to 1.

$$Nor(x) = \frac{\arctan(x) \times 2}{\pi}$$

The overall fitness is defined based on *#UnD*, *#UriS* and *PETR*, which can be calculated as

$$Fitness = 0.2 \times PETR + 0.8 \times Nor(\#UnD + \#UriS)$$

2.2.1.2.2.5 *Evaluation*

As specified in D1.3, there are 3 uncertainties for UC2_INTE_2.3. Based on the UncerTum, we further refined these uncertainties and as the result we modelled 7 uncertainties in the belief state machine.

To evaluate the performance of evolving belief state machine using GP, we performed an experiment with one use case of ULMA (UC2_INTE_2.3). Given the randomness in GP, we repeated the experiments 10 number of times. In 10 number of repetitions, we identified on average 2 new uncertainties. As compared to the known uncertainties 7 in belief state machine defined by SRL, we managed to discover 28.6% ($\frac{\#UnD}{\#UnS} = \frac{2}{7}$) new uncertainties.

We are running experiments with other use cases at the time of submission of this deliverable. The detailed results based on the other use cases will be provided in the *Empirical Evaluation Deliverable* in WP5.

2.2.1.2.3 Measurement Inference

In this section, we present the way to update the objective uncertainty measurements based on the execution results.

The detailed algorithm is shown in Figure 27.

```
Measurement inference
input     log:ExecutedLogItem[], un
output    frequence:Double
1   total:int
2   occur:int
3   for item ← log
4     if sameSsTr(item, un)
5           total++
6       if sameTs(item,un)
7         occur++
8   frequence = (occur*1.0)/total
```

Figure 22. The Algorithm of Measurement inference

2.2.2 Updates on Test Case Generation, Test case Minimization, and Test Prioritization

This section presents updates on UncerTest, comparing with D3.2. The updated technical report (**TR8.pdf**) is also attached. The TR also provides the extensive experiments, we conducted with five use cases from Automated Warehouse and GeoSports case studies.

2.2.2.1 Test Case Generation and Minimization

The main updates in the technical report (**TR8.pdf**) include:

1. The application of UncerTest in the test process is presented in Figure 1 in **TR8.pdf**;
2. The executable test generation is updated in terms of introduction of indeterminacy sources (Section 4.3 in **TR8.pdf**);
3. An extensive experiment to evaluate four uncertainty-wise test cases minimization problems with eight multi-objective algorithms are made (Section 5).
4. The extensive experiment includes five use cases (UC2_APP11, UC2_INFR11, UC2_INTE11, UC2_INTE23 and UC1_INTE3) from both industrial case studies.
   The experiments were aimed to select the best algorithm out of eight commonly used multi-objective search algorithms, for each of the four minimization strategies, with five use cases of two industrial CPS case studies. The minimized set of test cases obtained with the best algorithm for each minimization strategy were executed on the two real CPSs. The results showed that our best test strategy managed to observe 51% more uncertainties due to unknown indeterminate behaviors of the physical environment of the CPSs as compared to the rest of the test strategies. In addition, the same test strategy managed to observe 118% more unknown uncertainties as compared to the unique number of known uncertainties. All the details are presented in **TR8.pdf**.

2.2.2.2 Test Case Prioritization

No update. It is same as the **TR7.pdf** submitted in D3.2.

2.2.3 Updates on Test Case Execution

The uncertainty-wise test case verdict is newly proposed in the updated technical report (**TR8.pdf**), which can be further used to define the uncertainty-related metric. For more details, (see Section 4.4 and Section 5.3 in **TR8.pdf)**.

3 Summary and Conclusion

3.1 UTF at the Application Level

Achievements of M5

The Uncertainty Testing Framework for the Application Level (UTF-AL) has been elaborated in more details. In particular the encoding and its relationship to the UML Testing Profile v2 (UTP2) has been presented, that allows a consistent usage of UTF-AL together with UTP2 and reduces technical barriers for testers who already employ model-based testing together with UTP2. Furthermore, details on two alternative crossover operators for uncertainty testing has been presented that allow testing recombination of uncertainties in the environment of a CPS. The possible extensions and configurations points of UTF-AL for users and developers were discussed in order to facilitate the application and improvement of UTF-AL.

3.2 UTF at the Infrastructure Level

Achievements of M5

We have introduced a new uncertainty modelling and evaluation methodology (UME) and implemented a companion tool (T4UME) based on uncertainty detection rules (UDR) as planned in D3.2.

We have implemented a UML2JSON extraction step to ease the adoption of uncertainty-wise UML model in JSON-based provisioning and deployment of infrastructural resources.

Due to (i) the heterogeneity and continuous evolution of uncertainty methodologies applied at application, infrastructure, and integration level and (ii) continuous evolving functionalities of UTF tools, UME has been devised to support uncertainty evolution at design-time, caused by missing stereotype property values.

T4UME, the UME companion tool, built on top of state of the art MDE technologies (Eclipse Epsilon), provides an advanced feature (i.e., a so-called higher-order transformation) to adapt its capabilities (i.e., new check clauses for U-Detection and/or new fix clauses for U-Refactoring) to different MDE tasks, like the test case generation algorithms implemented by U-Test tool providers.

3.3 UTF at the Integration Level

Achievements of M5

We have successfully reached the milestone M5 regarding the UTF V.3 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.2 include:

1) Uncertainty-wise test minimization has been improved significantly in terms of large scale experiments with additional search algorithms. The results showed that our best test strategy managed to observe 51% more uncertainties due to unknown indeterminate behaviors of the physical environment of the CPSs as compared to the rest of the test strategies. In addition, the same test strategy managed to observe 118% more unknown uncertainties as compared to the unique number of known uncertainties.
2) The development of uncertainty-wise test case verdicts, which are used to observe occurrence of uncertainties together with its related indeterminacy source(s).
3) The development of the new model evolution framework (UncerPlore), which evolves test-ready models using Genetic Programming (GP) [17] [18] and benefiting from runtime test ready model execution on the dedicated test infrastructures (Section 2.4.1.2). Our initial experiments based on one use case from ULMA demonstrated discovery of 28.6% new uncertainties as compared to the ones specified in the initial belief state machine, which is input for UncerPlore.

4 Bibliography

[1] *U-Test H2020 Deliverable: Revision of deliverable report D1.2: Updated Report on U-Taxonomy.*

[2] *U-Test H2020 Deliverable: Report on Uncertainty Modelling Framework V.3.*

[3] ObjectManagementGroup, *UML Testing Profile.*

[4] ObjectManagementGroup, *UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems.*

[5] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[6] E. Foundation, *Epsilon Project,* 2017.

[7] H.-L. Truong, L. Berardinelli, I. Pavkovic and G. Copil, “Modeling and Provisioning IoT Cloud Systems for Testing Uncertainties,” in *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, Melbourne, 2017.

[8] M. Brambilla, J. Cabot and M. Wimmer, Model-Driven Software Engineering in Practice, Morgan & Claypool, 2012.

[9] E. Foundation, *PapyrusUML.*

[10] E. Foundation, *Eclipse Modeling Framework.*

[11] D.-H. Le, N. Narendra and H.-L. Truong, “HINC-harmonizing diverse resource information across iot, network functions, and clouds,” in *Future Internet of Things and Cloud (FiCloud), 2016 IEEE 4th International Conference on*, 2016.

[12] D.-H. Le, H.-L. Truong and S. Dustdar, “Managing On-demand Sensing Resources in IoT Cloud Systems,” in *Mobile Services (MS), 2016 IEEE International Conference on*, 2016.

[13] M. Zhang, S. Ali, T. Yue and R. Norgre, “Uncertainty-wise evolution of test ready models,” *Information and Software Technology,* 2017.

[14] M. Zhang, T. Yue and M. Hedmnan, “Uncertainty-wise Test Case Generation and Minimization for Cyber-Physical Systems: A Multi-Objective Search-based Approach,” 2016.

[15] J. R. Koza, “Genetic programming II: Automatic discovery of reusable subprograms,” *Cambridge, MA, USA,* 1994.

[16] J. R. Koza, Genetic programming: on the programming of computers by means of natural selection, vol. 1, MIT press, 1992.

[17] V. Vinay, I. J. Cox, N. Milic-Frayling and K. Wood, “On ranking the effectiveness of searches,” in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, 2006.

[18] I. Object Management Group, *Unified Modeling Language, UML, version 2.5,* 2015.

[19] V. Vyatkin, “Software Engineering in Industrial Automation: State-of-the-Art Review,” *IEEE Trans. Industrial Informatics,* vol. 9, pp. 1234-1249, 2013.

[20] F. Pramudianto, I. R. Indra and M. Jarke, “Model Driven Development for Internet of Things Application Prototyping.,” in *SEKE*, 2013.

[21] F. Ciccozzi and R. Spalazzese, “MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering,” in *Intelligent Distributed Computing X: Proceedings of the 10th International*

Symposium on Intelligent Distributed Computing -- IDC 2016, Paris, France, October 10-12 2016 , C. Badica, A. El Fallah Seghrouchni, A. Beynier, D. Camacho, C. Herpson, K. Hindriks and P. Novais, Eds., Cham, : Springer International Publishing, 2017, pp. 67-76.

[22] E. Foundation, *Eclipse UML,* 2017.

[23] B. Liu, "Uncertainty theory," in *Uncertainty Theory*, Springer, 2007, pp. 205-234.

[24] *U-Test H2020 Deliverable: Report on Uncertainty Modelling Framework V.2.*