Uncertainty-Wise Evolution of Test Ready Models

Man Zhang, Shaukat Ali, Tao Yue and Roland Norgre

Abstract

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 behavior 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 modelers (*belief agents*) with their assumptions about the expected behavior of a CPS, its expected physical environment, and potential future deployments. Thus, these models explicitly contain subjective uncertainty of the test modelers.

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 modelers. 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 (modeling test oracles) and guards of transitions (modeling 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.

Keywords— Uncertainty; Belief Model; Belief Test Ready Model; Model Evolution; Model-Based Testing.

– – – – – – – – – – ◆ – – – – – – – – – –

1 INTRODUCTION

Handling the inherent uncertainty in Cyber-Physical Systems (CPSs) is a well-known challenge, which requires novel approaches for understanding, discovering and modeling uncertainty, and verifying and validating CPSs under uncertainty [5-8]. Typically, a CPS is developed by integrating various physical units (e.g., devices), which are usually black-boxes (with or without the API access) with known and uncertain assumptions on its physical operating environments and deployments. Thus, when testing a CPS, not only assumptions are made about the internal behavior of the CPS, but also its operating environments and deployments. More specifically, when performing model-based testing (MBT), the expected behavior of a CPS is modeled with the explicit consideration of uncertainty, including uncertain behaviors of its physical environments and uncertain deployments (the focus of

- *Man Zhang is a Ph.D. Student at Simula Research Laboratory, Oslo, Norway. E-mail: manzhang@ simula.no.*
- *Shaukat Ali is a Senior Research Scientist at Simula Research Laboratory, Oslo, Norway. E-mail: shaukat@simula.no.*
- *Tao Yue is a Chief Research Scientist at Simula Research Laboratory and an Adjunct Associate Professor at the University of Oslo, Oslo, Norway. E-mail: tao@simula.no.*
- *Roland Norgre is a Process Manager R&I with Future Position X, Gävle, Sweden. E-mail: roland.norgren@fpx.se*

our previous work [9]). Such models are typically created by one or more test modelers (belief agent(s)) based on his/her/their assumptions about a CPS, its operating environments, and deployments and thus the captured uncertainty is *subjective* to the test modeler(s).

Naturally, these test ready models, named as *Belief Test Ready Models (BMs)* in the rest of the paper, can be continuously evolved based on real operational data (which introduce *objective* uncertainty) of the current deployment of the CPS such that the evolved models can be used to generate new test cases to test future deployments of the CPS with both captured subjective uncertainty and evolved objective uncertainty.

1.1 Challenges and Objectives

Testing is mainly concerned with sending stimulus (via e.g., test APIs) with test data to a CPS and checking the correctness of changes of corresponding states (e.g., test oracles). In the uncertainty-wise MBT context, BMs are the key artifacts for generating executable test cases. Therefore, the quality of BMs is critical for ensuring the quality of generated test cases and consequently the quality of the CPS under various deployments. Hence, the overall scientific challenge is how to ensure the quality of BMs such that they are ready for being used to generate test cases. It is challenging because in the context of uncertainty-wise MBT for CPSs such BMs are complex (e.g., specified in multiple UML state machines) and subjective uncertainty (reflecting test modelers' belief and specified as part of BMs) need to be continuously validated with *evidence* (e.g., real operational data) continuously collected from existing deployments of the CPS.

Correspondingly, our overall objective is to address this challenge by proposing a model evolution framework, called *UncerTolve*, for evolving BMs, with real operational data collected from real CPS applications. This is feasible, as in the context of continuously deploying a CPS for various applications (details in Section 1.2), real operational data can be collected from already deployed applications of the CPS. Collected real operational data are valuable resources to enhance the initial BMs from the perspective of the correctness and completeness, including uncertainty information, test oracles, and test data specifications. Moreover, such a process can be continuous in the sense that as long as there is new operational data available, the BMs can be evolved to accommodate information contained in the data. Evolved BMs will be therefore more complete and correct. Subsequently, testing the CPS for future deployments, based on the evolved BMs, will be much better supported. We provide a clear correspondence between the sub-challenges and sub-objectives in Table 1.

Table 1. Sub-challenges and Sub-objectives of *UncerTolve*

Sub-challenges	Sub-objectives
How to ensure the syntactic and semantic correctness of BMs?	**Model Validation**: Validate and update (with proposed heuristics) BMs with real operational data, via model execution.
How to ensure the quality of uncertainty information captured in BMs?	**Derivation of Objective Uncertainty Measurements**: Derive objective uncertainty measurements from real operational data and enhance BMs by integrating them with subjective uncertainty measurements already specified in the BMs.
How to ensure the quality of test oracles (represented as state invariants) and test data specifications (represented as guard conditions) of BMs?	**Inference of Test Oracles/Test Data Specifications**: Abstract invariants (both related to test oracles and test data specifications) from real operational data, by relying on existing dynamic invariant inference techniques.
How to achieve the above sub-challenges in an integrated manner?	**Methodologies/Heuristics/Process**: Define methodologies and heuristics on how to update BMs. Suggest a practical process that integrates model validation, objective uncertainty measurement derivation, and test oracles and test data specification inferences, based on real operational data and model execution.

1.2 Context, Scope and Overview

In the context of an EU project [10], we are developing a model-based and search-based framework for testing CPSs under *known* and *unknown* uncertainties to assure that CPSs deal with uncertainty during their operation and do not harm anyone or anything. Evolving BMs in a systematic manner for preparing them for enabling the generation of executable test cases is one of the key components of the model-based and search-based framework.

The overall context is presented in Figure 1, where *UncerTum* [9] is a UML-based, uncertainty modeling framework for constructing BMs, and *UncerTest* [11] is a model-based and search based test case generation and minimization framework. *UncerTolve* (with its key features, inputs and outputs indicated as white boxes in Figure 1) is the framework we propose in the paper for evolving BMs developed with *UncerTum*. Evolved BMs are input for *UncerTest* to generate executable test cases.

A CPS may be deployed to more than one applications of the same or different application domains. For example, as discussed in [12], in the avionics industry, multiple system instances (i.e., multiple deployments) of the same CPS type can be deployed to achieve a common goal. In the context of our project, the industrial CPS of GeoSports[1] can be deployed for a variety of sports including Bandy and Ice Hockey. Each application corresponds to a unique deployment, denoted as D_1, D_2, ..D_n. *UncerTolve* evolves BMs developed for a CPS with real operational data collected from available deployments of

[1] http://www.u-test.eu/use-cases/

the CPS. Test cases generated using *UncerTest* from the BM evolved with *UncerTolve* can be used to test both existing deployments (D_1, D_2, …D_n) and new ones (Figure 1). Note that the process is naturally iterative as the process of introducing new deployments, collecting real operational data, based on which the BM is updated, testing new deployments based on the evolved BMs, are all iterative.

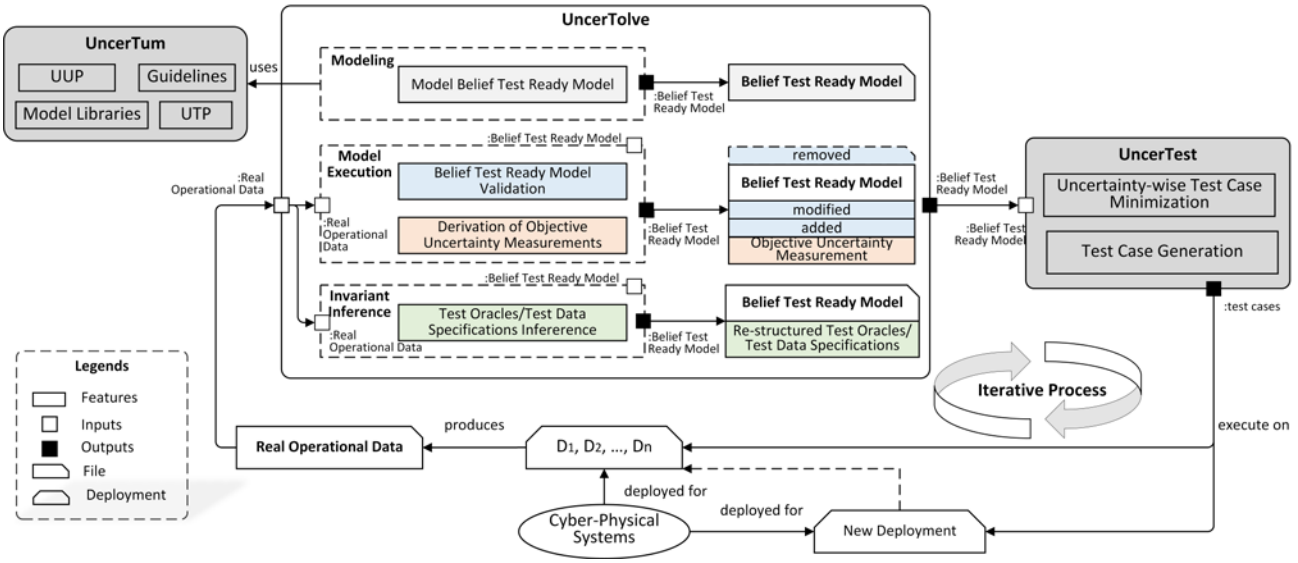


Figure 1. The Overall Context and Scope of *UncerTolve*

UncerTolve consists of three activities (i.e., Modeling, Model Execution, and Invariant Inference) and four components (denoted with different colors in the *UncerTolve* box), as shown in Figure 1.

The kickoff activity of *UncerTolve* is about modeling BM. We develop the initial BM for a CPS, specified with the Unified Modeling Language (UML) [13]. Such a UML model includes composite structure diagrams, class diagrams, constraints specified in the Object Constraint Language (OCL) [14], and state machines capturing testing interfaces and behaviors of the *application*, *infrastructure*, and *integration* levels of the CPS [9, 15]. *UncerTolve* relies on *UncerTum* [9] to explicitly model known and subjective uncertainties specified by modelers (i.e., belief agents [15]), as part of the initial BM. *UncerTum* consists of the UML Uncertainty Profile (UUP) [9] and a set of model libraries and utilizes the UML Testing Profile (UTP) V.2 [16]. To enable model execution, as part of the *UncerTolve* framework, in this paper, we also propose a modeling methodology (which extends *UncerTum*) particularly for the purpose of developing executable BMs.

The second activity is to execute BMs with real operational data. This activity involves two components: validation of BM and derivation of objective uncertainty measurements. The initial BM created in the first activity is executed to validate their syntactic correctness and conformance against real operational data. Missing or incorrect model elements might be identified during the model execution process and therefore the initial BM can be updated accordingly, based on a set of heuristics newly defined as part of *UncerTolve*. Through model execution with real operational data, *objective*

uncertainty measurements can also be obtained. During this activity, existing model elements in the initial BM can be removed or modified, and new ones can be added. Obtained objective uncertainty measurements can also be appended to the BM.

In the third activity is about inferring test oracles (i.e., state invariants) and test data specifications (guard conditions) with real operational data using dynamic invariant inference techniques [4, 17, 18]. In this paper, we apply one solution, Daikon [4], which produces a set of invariants (corresponding to test oracles and test data specifications) with an implemented machine learning technique. These invariants are then merged with OCL constraints specified as part of the BM, based on newly defined heuristics, which therefore leads to another round of the updating of the BM, i.e., restructuring test oracles and test data specifications. Numerous techniques (e.g., aka automata learning [19], data mining [20, 21]) have been proposed in the literature in the field of automated inferences of various types of information (e.g., properties, protocols, interfaces, specifications) from programs. Although, our work relies on an existing work, i.e., Daikon, our work differentiates itself from the existing works in terms of the core challenge it tackles, i.e., evolving BMs with both subjective and objective uncertainty to eventually support MBT of CPSs under *known* and *evolved* uncertainties discovered based on real operational data.

Note that the modeling activity of *UncerTolve* is the foundation of the other two activities. The other two activities are independent to each other, although we recommend to apply them sequentially as doing so will improve the overall quality of evolved BMs and this is also how our industrial case study was conducted. In summary, theoretically, the output of each activity can be used as the input to *UncerTest*; however, sequentially applying model execution and invariant inference are strongly recommended in practice for ensuring the quality of delivered BMs. This is especially important when BMs are complex, which is quite common in industrial settings.

1.3 Contributions

UncerTolve evolves BMs specified with *UncerTum* [9], which are essentially stereotyped UML class diagrams, composite structure diagrams, and state machines, and therefore contain richer information than a typical specification representation (e.g., Finite State Machines (FSMs)) that can be inferred with existing techniques (e.g., [19, 22]).

Distinguishing itself from existing works, *UncerTolve* takes into account both subjective uncertainty information specified as *belief elements* of the BM and objective uncertainty information derived from real operational data and evolves them as part of the integrated BM evolving process.

Similar to some existing dynamic inference approaches (e.g., [1-3]), *UncerTolve* uses a machine learning technique implemented in Daikon to dynamically infer state invariants (modeling test

oracles) and guard conditions (modeling test data specification) of UML state machines. However, *UncerTolve* relies on real operational data collected from real applications of CPSs, instead of execution traces of programs. Note that *UncerTolve* aims to evolve BMs developed for CPSs and therefore existing approaches relying on execution traces of programs cannot be applied or at least cannot be directly applied without adaptation for the CPS context.

In conclusion, we summarize the key contributions of *UncerTolve* as below:

- *UncerTolve* has a modeling methodology for creating executable BMs with real operational data to support validation of the syntactic correctness of a BM modeled using *UncerTum* and checking conformance of the BM with the real operational data;
- *UncerTolve* defines a systematic and automated process for validating a BM with both subjective and objective uncertainty and defines a set of heuristic rules (named as tolveR-E) to guide test modeler(s) to update the BM based on validation results;
- *UncerTolve* is equipped with an automated solution for calculating and abstracting objective uncertainty measurements from the real operational data and the obtained measurements are appended to the BM;
- *UncerTolve* applies a machine learning technique to infer test oracles (state invariants in UML state machines) and test data specifications (guard conditions in UML state machines);
- *UncerTolve* defines a set of heuristic rules to evolve a BM with inferred state invariants, guard conditions and objective/subjective uncertainty measurements;
- *UncerTolve*, as a proof-of-concept, is evaluated with one industrial CPS, i.e., GeoSports from the healthcare domain.

1.4 Results and the Structure of the Paper

With *UncerTolve*, we managed to evolve 51% of belief elements, 18% of states, and 21% of transitions as compared to the initial BM. Thus, we conclude that *UncerTolve* is successful in evolving BMs with subjective and objective uncertainty information.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 represents the background. Section 4 represents terminology and running example. Section 5 represents the overall workflow of *UncerTolve*. Section 6 represents the methodology of *UncerTolve*. Section 7 presents the evaluation and discussion, whereas we conclude the paper in Section 8.

2 RELATED WORK

In this section, we compare *UncerTolve* with existing works in Section 2.1, whereas comparison with our own previous related works in Section 2.2.

2.1 Comparison with Existing Works

Several works (e.g., [4, 23-31]) have been published in the literature that infer, e.g., FSMs, their extensions, Live Sequence Charts (LSCs), and properties of software applications from execution traces. Most of these works rely on Daikon [4] to dynamically infer invariants from execution traces.

The work reported in [23] infers deterministic FSMs of black box components from their execution information to understand their behavior in the absence of a formal specification. The inferred FSMs are further generalized into intentional behavior models by synthesizing graph transformation rules. The process involves identifying invariant properties in a similar way as Daikon. The approach was evaluated with three different sets of classes implementing data abstractions such as Queue and MinSet.

An empirical study is reported in [24] to evaluate four strategies of inferring FSMs: 1) traces-only, 2) invariants-only, 3) invariant-enhanced-traces, i.e., inferring models from execution traces followed by enhancing them with invariants), and 4) trace-enhanced-invariants, i.e., inferring models from invariants followed by enhancing them with execution traces). Nine open-source libraries were used to compare the four strategies based on the quality of the resultant FSMs. The second and third strategies were evaluated to be the best ones.

Lo et al. [25] proposed an approach with a tool to enhance the precision of mining FSMs from code and traces by inferring temporal properties and incrementally merging equivalent states. Similarly, Walkinshaw and Bogdanov [26] proposed an approach to allow additional inference of state machines, based on temporal logic formulas and an extra capability to introduce new formulas during the inference process. The proposed approach was evaluated with two software applications. Gabel and Du [32] presented a general specification mining framework (Javert) for learning complex temporal properties (specified as specification patterns in FSMs) from execution traces.

Krka et al. [18] proposed an automated approach to infer object-level FSMs from execution traces and program invariants. First, it derives an FSM that captures legal invocation sequences of an object's public interfaces based on inferred data-value invariants. Second, it uses collected dynamic invocation traces to refine the invariant-based FSM to an object-level FSM.

Tonella et al [27] [28] proposed an approach to infer FSMs for supporting MBT based on a combination of clustering, invariant inference and genetic algorithm (GA). GA was used to optimize the quality attributes of inferred FSMs. The approach was evaluated with a small e-commerce application.

Walkinshaw and Taylor [33] proposed an approach to infer deterministic Extended FSMs (EFSMs) with WEKA [34] and Daikon and evaluated the approach with five Java and Erlang programs.

An algorithm is proposed in [26] to extract FSMs with parameters (FSAMs) from interaction traces:

sequences of method invocations. FSAMs put constraints on the values of parameters. The algorithm has three sequential steps: merging similar traces, deriving constraints with Daikon, and merging equivalent states. The Builder design pattern was used to evaluate the proposed approach.

In [29], an approach was proposed to infer communicating FSMs (CFSMs) from execution traces of concurrent programs that has three steps: 1) mining temporal properties (invariants), 2) creating an initial CFSM model, and 3) refining the CFSM model. The proposed approach was evaluated with three networked systems. The authors of [31] proposed an approach to infer resource-aware FSMs from execution logs of the software application by following similar steps. The proposed approach was evaluated with a case study on the TCP protocol.

Berg et al. [30] proposed a way to adapt regular inferences of FSMs from observations of component behaviors to construct models of communication protocol entities. The challenge that the authors tried to tackle is to infer state machines where messages have arbitrary parameters; however, it only handles Boolean parameters. Later on, Berg et al. [19] also made an effort to infer state machines with an infinite state space. First, the proposed approach infers finite-state Mealy machines by observing the behavior of a communication protocol from a small domain. Second, it transforms them into infinite-state Mealy machines by folding the inferred finite-state Mealy machines into compact symbolic models.

Lo et al. [20] presented an approach to mine specifications as restricted LSCs from execution traces that are transformed into UML sequence diagrams with a modal profile applied. Later on, Lo and Maoz [21] made an effort to integrate the value-based specification mining approach of Daikon with a sequence-based approach to mine specifications as LSCs. A scenario-based slicing technique was applied to obtain sliced traces. Value-based invariants mining is then applied to both on the original traces and the sliced traces to identify scenario-specific invariants. Four software applications were used to evaluate the proposed approach.

Beschastnikh et al. [35] proposed an automated approach to infer invariant constrained models from system execution logs, by intentionally reducing the involvement of developers. First, the approach mines temporal invariants from logs and generates trace models, from which it generates initial models (in the form an authors-defined, edge and node style graphical representation). These initial models are then refined and coarsened to explore the space of models.

Raz et al. [36] proposed a way to infer invariants based on the observations of the behavior of dynamic data feeds (i.e., a time-ordered sequence of observations) to detect semantic anomalies in online data sources. The approach relies on an augmented Daikon and Mean (i.e., a statistical method for estimating a confidence level for the mean of a distribution). The approach was evaluated with real-world data. In [37], the authors proposed a heuristics based algorithm to scale up dynamic

inferences of properties/invariants of software applications from execution traces. The approach was evaluated on JBoss and the Windows kernel. Hangal and Lam [38] proposed an approach (similar to Daikon) to detect program invariants from program executions. It also reports detected dynamic invariant violations.

The work reported in this paper builds on an existing work, i.e., Daikon to infer invariants based on execution information. However, in our case, real operational data was used from real applications of CPSs. To compare with these related works, we distinguish *UncerTolve* from the following four aspects. First, most of the related work directly take programs as input to infer, e.g., specifications and API. *UncerTolve*, however, takes test ready UML models together with explicitly captured subjective uncertainty as input and evolve them based on model execution using real operational data, based on dynamic inference with Daikon. Second, *UncerTolve* aims to handle CPSs, not just programs. This means that we not only evolve models of applications but also infrastructures and their interactions. Third, *UncerTolve* evolves belief models including discovering previously unknown belief elements (in stereotypes), states, and transitions. Fourth, the ultimate objective of *UncerTolve* is to facilitate MBT of CPSs under known and unknown uncertainty, instead of program comprehension and bug detection like most of the related works do.

2.2 Comparison with Our Previous Works

To understand uncertainty in general, in our previous work [15], we developed a generic conceptual model called U-Model. Our aim was to precisely define uncertainty and its associated concepts for CPSs. The U-Model was implemented in two ways: 1) As an extension of an existing restricted use case specification language (named as RUCM) [35], to specific uncertainty in use case specifications called as U-RUCM [39], 2) Implementation of U-Model as a UML profile—the UML Uncertainty Profile (UUP) to enable MBT of CPSs under uncertainty. UUP together with other related profiles and model libraries were implemented as a modeling framework—*UncerTum* [9]. With *UncerTum*, test modelers can create BMs with explicit consideration of subjective uncertainty. As shown in Figure 1, BMs created with *UncerTum* [9] are the key inputs of *UncerTolve*—the key contribution of this paper. *UncerTum* only focuses on creating BMs for test case generation and cannot be used to further enhance BMs into executable ones such that these models can be validated with real operational data. In the context of *UncerTolve*, we propose an extension to *UncerTum* for converting BMs developed with *UncerTum* into executable ones.

In [11], we reported *UncerTest* [11], an uncertainty-wise testing framework. *UncerTest* implements various uncertainty-wise test case generation and minimization strategies that can be used to generate test cases from BMs developed with *UncerTum* [9]. *UncerTest* [11] can be used to generate test cases

from BMs evolved with *UncerTolve* to test CPSs. However, we may need to define additional test strategies in *UncerTest* [11] to focus specifically on the evolved parts of the evolved models. We plan to implement these test strategies in our future work.

3 BACKGROUND

In this section, we present the background that is necessary to understand the rest of the paper. In Section 3.1, we define a CPS at a generic level, along with three logical levels, at which uncertainty may occur. In Section 3.2, we introduce UTP, which is one of the key profiles applied to BMs to enable MBT. Section 3.3 presents U-Model, a conceptual model defining uncertainty and its associated concepts. Section 3.4 introduces *UncerTum*, an uncertainty-wise modeling framework to create BMs of CPSs with subjective uncertainty, i.e., the key input of *UncerTolve*. Section 3.5 presents *UncerTest*, an uncertainty-wise test case generation, and minimization framework to generate test cases from BMs created with *UncerTum* and evolved with *UncerTolve*.

3.1 Cyber-Physical Systems and Uncertainty Levels

A CPS is defined as [15]: *"A set of heterogeneous physical units (e.g., sensors, control modules) communicating via heterogeneous networks (using networking equipment) and potentially interacting with applications deployed on cloud infrastructures and/or humans to achieve a common goal"* and is conceptually shown in Figure 2. Uncertainty in a CPS can occur at the following three logical levels [15]. First, uncertainty at the *Application level* is due to events/data originating from an application (one or more software components) of a physical unit of the CPS. An example of application-level uncertainty is the indeterminate behavior of a human interacting with a CPS, e.g., not wearing a device sensing heart rate properly which leads to uncertain heart rate readings. Second, uncertainty at the *Infrastructure level* is due to data transmission via information network enabled through networking infrastructure and/or cloud infrastructure. An example of infrastructure level uncertainty includes the uncertain behavior of a CPS due to packet loss in an information network. The third level is the *Integration level*, due to either interactions of applications across the physical units at the application level, or interactions of physical units across the application and infrastructure levels (e.g. the abnormal heart rate captured by heart rate sensor due to the loss of the connection between heart rate sensor and a computer system analyzing the heart rate assuming the heart rate sensor and a computer system are connected via wireless network). More details and examples are provided in [15].

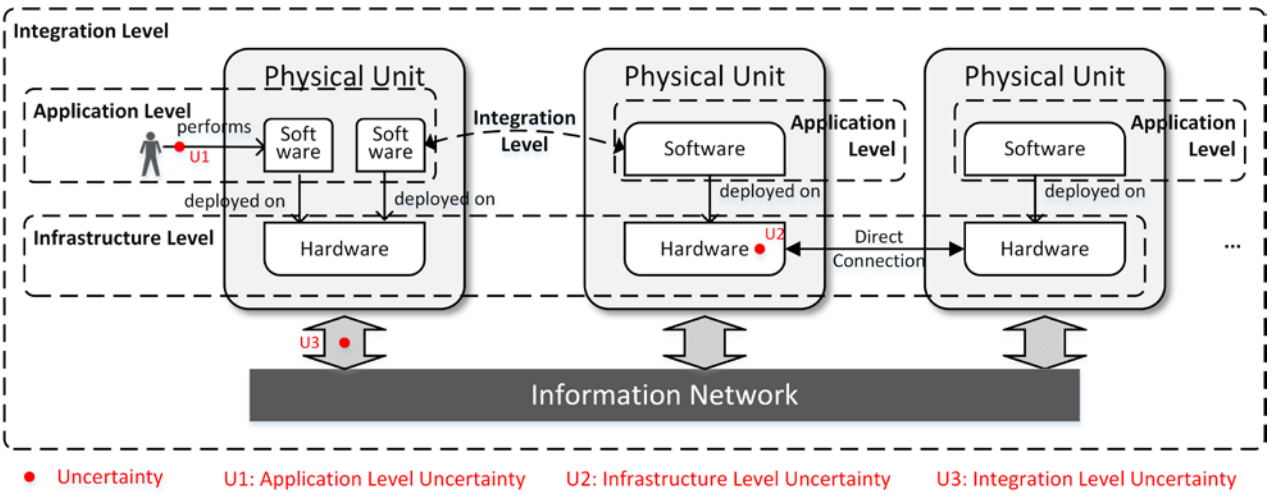


Figure 2. Conceptual model of CPSs and the Three Uncertainty Levels

3.2 UML Testing Profile

UML Testing Profile (UTP) [39] is a standard at Object Management Group (OMG) for enabling MBT. With UTP, the expected behavior of a system under test can be modeled, from where test cases can be derived. UTP can be also used to directly model test cases. Transformations from models specified with UTP to executable test cases can be performed using specialized test generators. Since UTP is defined as a UML profile, it is often applied on UML sequence, activity diagrams and state machines for describing behaviors of a system under test or test cases. The key purpose is to introduce testing related concepts (e.g., *Test Case*, *Test Data*, and *Test Design Model* and model libraries such as various types of test case Verdict (pass, fail)) to UML models for the purpose of enabling automated generation of test cases. UTP V.2 is the latest revision of the UTP profile, which is conceptually composed of five packages of concepts: Test Analysis and Design, Test Architecture, Test Behavior, Test Data and Test Evaluations. Various numbers of stereotypes have been defined for some concepts of these packages. Similar to other modeling notations, it is never been an objective to exhaustively apply all the stereotypes when using UTP V.2 to annotate UML models with testing concepts. Which stereotypes to apply and how to apply them are however problem/purpose specific and should be defined by users of the profile. More information about UTP V.2 and the team can be found in [27; 38].

To enable MBT of CPSs under uncertainty, we rely on UTP V.2 to model the testing aspect of BMs. In our context, only a subset of UTP V.2 was used.

3.3 U-Model

To understand uncertainty in the general context of software engineering, we developed a conceptual model called U-Model [15] to define uncertainty and its associated concepts. The U-Model was developed based on an extensive review of existing literature on uncertainty from several disciplines including philosophy, healthcare and physics and two industrial case studies.

The U-Model takes a subjective approach to represent uncertainty, which is modeled as a state (i.e., worldview) of some agents (called *BeliefAgent*s), who, for whatever reason, do not have complete and fully accurate knowledge about some subjects of interest. A *Belief* is an abstract concept that can be expressed in the concrete form via one or more explicit *BeliefStatement*s (a concrete and explicit specification of some *Belief* held by a *BeliefAgent* about possible phenomena or notions belonging to a given subject area). Uncertainty (i.e., lack of confidence) represents "a state of affairs whereby a *BeliefAgent* does not have full confidence in a belief that it holds" [15]. This may be due to several factors: lack of information, inherent variability in the subject matter, ignorance, or even due physical phenomena such as the Heisenberg uncertainty principle [21]. While uncertainty itself is an abstract concept, it can be quantified by a corresponding *Measurement*, which expresses in some concrete form the subjective degree of uncertainty that the agent ascribes to a *BeliefStatement*. As the latter is a subjective notion, a *Measurement* should not be confused with the degree of validity of a *BeliefStatement*. Instead, it merely indicates the level of confidence that the agent has in a statement. Further details on U-Model may be consulted in [15].

3.4 UncerTum

UncerTum [9] (Figure 1) is uncertainty-wise modeling framework to support the development of BMs of CPSs, which consists of specialized UML-based modeling notations (named as UUP) for specifying uncertainties to enable MBT of CPSs under uncertainty.

UUP is at the core of *UncerTum* and implements U-Model [15] (Section 3.3). UUP consists of three parts (i.e., *Belief*, *Uncertainty*, and *Measurement* profiles) and an internal library containing enumerations required in the profiles. To ease the development of BMs with uncertainty, *UncerTum* additionally defines four sets of UML model libraries: *Pattern*, *Time*, *Measure*, and *Risk* libraries, by extending an existing UML profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [40]. *UncerTum* also includes a small UML profile called the CPS testing levels profile to allow stereotyping (labeling) test ready model elements with three CPS test levels (e.g., integration level). The purpose is to differentiate model elements from different levels and facilitate defining level specific test strategies. Moreover, *UncerTum* relies on UTP V.2 (Section 3.2) to model BMs for the purpose of enabling MBT. Finally, *UncerTum* defines a set of concrete guidelines (i.e. *Measurement Modeling*) on how to use its modeling notations to construct BMs with uncertainty explicitly specified.

3.5 UncerTest

UncerTest [11] (shown in Figure 1) consists of two main part: test case generation and uncertainty-wise test case minimization. *Test case generation* takes the BM using *UncerTum* as input to automatically and systematically generate abstract test cases, according to two proposed test case generation strategies:

All Simple Paths (No Loops) and *All Paths with a Fixed Maximum Length*. These two strategies were inspired from the ones reported in [41], but were extended for BMs specified in *UncerTum* and considered various uncertainty aspects such as the number of uncertainties in a test path and overall uncertainty of a test path, defined based on Uncertainty Theory [42]. *Uncertainty-wise test case minimization* was proposed because the number of abstract test cases generated by *Test Case Generation* is typically very large for any non-trivial CPS and it is impossible to execute all of them. The uncertainty-wise test case minimization problem is a multi-objective search problem with four objectives: 1) The average number of uncertainties covered by the subset of the test cases after minimization; 2) The average percentage of *uncertainty space* (defined in Uncertainty Theory [42]) covered by the subset of the test cases after minimization; 3) The average *uncertainty measure* (defined in Uncertainty Theory [42]) of the subset of test cases after minimization; and 4) The average number of unique uncertainties covered by the subset of test cases after minimization.

4 TERMINOLOGIES AND RUNNING EXAMPLE

In this section, we will briefly present a running example together with relevant terminologies. The running example will be used in the rest of the paper to explain the key steps of *UncerTolve*. The *UncerTolve* itself will be presented in Sections 5-6.

4.1 Belief Test Ready Model

A *belief test ready model* (BM) consists of three types of models: a Composite Structure (CS) diagram, a set of class diagrams (CDs), and a set of Belief State Machines (BSMs). The belief aspect of BMs is from the perspective of modelers (i.e., belief agents), who create the BMs and therefore the BMs reflect their (subjective) beliefs on the information specified in the models.

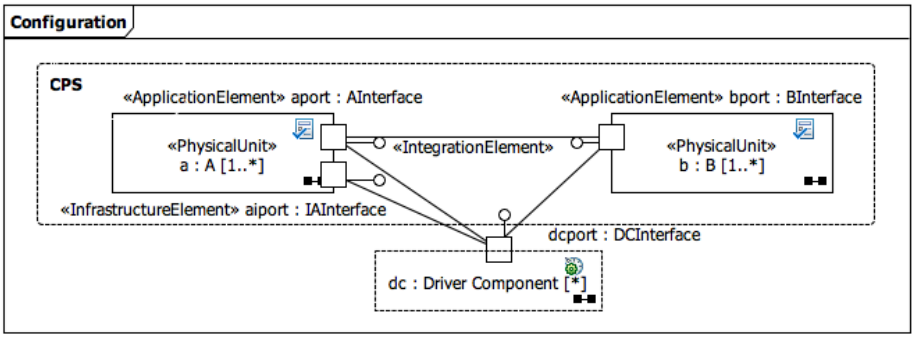


Figure 3. Composite Structure Diagram of BM (Running Example)

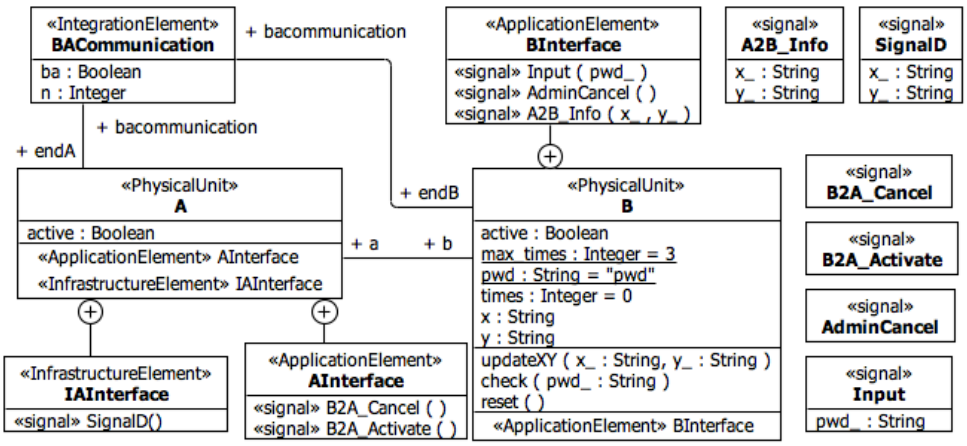


Figure 4. Class Diagram of BM (Running Example)

The CS diagram of a BM model represents a high-level test/model evolution configuration (referred as *Configuration* in Figure 3) of a CPS under Test. It captures various physical units that constitute the CPS, such as components *A* and *B* with «PhysicalUnit» applied. The stereotype is defined in the CPS profile of *UncerTum* [9]. Application and infrastructure level testing ports and interfaces of each physical unit are also explicitly modeled in the CS diagram. For example, as shown in Figure 3, *A* has one application-level port (*aport :: AInterface*) and one infrastructure level port (*aiport :: IAInterface*), which are stereotyped with «ApplicationElement» and «InfrastructureElement», respectively. Each port has a corresponding interface specified in the class diagram (Figure 4) such as *AInterface*. The integration level interface is stereotyped with «IntegrationElement» (represented as class *BACommunication* in Figure 4) and it is associated with *A* and *B*.

The structure of physical units is modeled as a set of UML class diagrams. Classes in the UML class diagrams capture various types of information required for testing, including APIs (e.g., the *reset()* operation of *B* in Figure 4), state variables (e.g., the *active:Boolean* attribute of *A* in Figure 4), test configuration parameters (e.g., the cardinality of instances of *A* in Figure 3), signals (e.g., *AdminCancel* in Figure 4), and signal receptions (e.g., *AdminCancel()* in *BInterface* in Figure 4). The class diagrams have the CPS profile (Section 3.1) applied to distinguish model elements of the three different levels. For example, *AInterface* is stereotyped as «ApplicationElement» to signify that it is an application level interface.

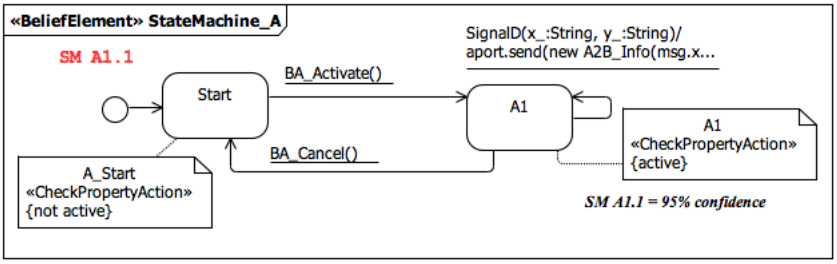


Figure 5. Belief State Machine of *A* (V1.1) (Running Example)

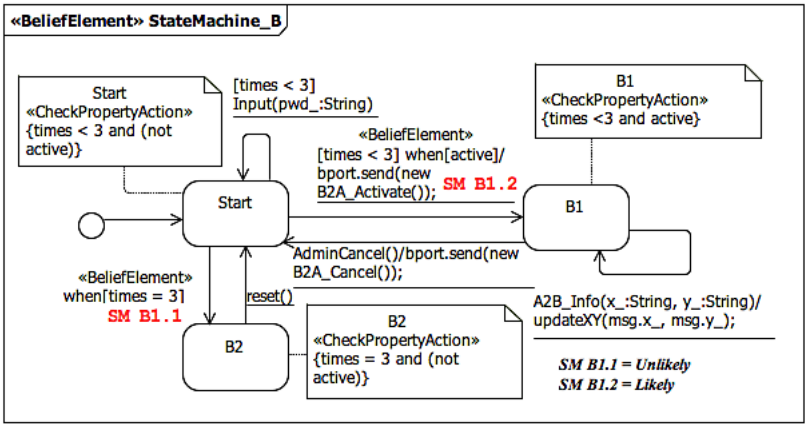


Figure 6. Belief State Machine of *B* (V1.1) (Running Example)

Each physical unit's test behavior is modeled as one or more BSMs using *UncerTum* (Section 3.4), e.g., as shown in Figure 5 and Figure 6 for *A* and *B* respectively. For example, as shown in the BSM for physical unit *A* (Figure 5), «BeliefElement» from UUP is applied to the state machine for *A*, where the confidence of the belief agent about this state machine is specified as 95%. Two key types of OCL constraints are defined in BSMs. Each basic state in a BSM is precisely defined with a state invariant (e.g., *not active* associated with the *Start* state of *A*, Figure 5) specified as an OCL constraint based on state variables defined in the CDs (e.g., attribute *active* of class *A* in Figure 4). These state invariants serve as test oracles and can be checked at runtime using existing OCL evaluators such as Eclipse OCL [43]. Second, each guard condition (e.g., guard *[times<3]* on the transition from *Start* to *B1* in *B*, Figure 5) is specified as an OCL constraint on the input parameters of the associated trigger, which defines the valid range of inputs. These constraints in our case are used to automatically generate test data to trigger transitions. An OCL solver (e.g., EsOCL [44]) can take these constraints as input and automatically generate test data [45]. BSMs are further enriched with UAL such that they can be directly executed with the IBM Rational Simulation Toolkit [46]. For example, the self-transition of State *A1* (Figure 5) has an effect whose body is written in UAL as below:

```
aport.send(new A2B_Info(msg.x_, msg.y_));
```

This effect simply sends signal *A2B_Info* from *A* to *B* via *aport*. Notice that *A2B_Info* is a signal reception in *BInterface*. Notice that signals in UML are typically used for modeling communications across state machines. In our running example, for instance, the state machine of physical unit *A*

(Figure 5) communicates with the state machine of physical unit *B* (Figure 6) via the UAL code of the effect of self-transition of state *A1* (Figure 5) as also shown in the last paragraph. Similarly, the state machine of physical unit *B* (Figure 6) communicates with the state machine of physical unit *B* (Figure 5) via the UAL code written in the effect of the transition from *B1* to *Start* in Figure 6.

4.2 Executable Belief Test Ready Model

An executable BM is a Java code, which is semantically equivalent to a BM discussed in Section 4.1. Executable BM Java code can be executed either directly with the IBM Rational Simulation Toolkit or as a standalone application by simply introducing a *main()* method. In Section 6.1, how to develop executable BM will be discussed in details.

4.3 Driver Model

In order to apply *UncerTolve*, we need to develop a component called *Driver Component* (e.g., *dc: Driver Component* in the composite structure diagram in Figure 3). A *Driver Component* is connected to physical units of a CPS via UML ports and connectors (Figure 3). A *Driver Component* has its own class diagram (e.g., Figure 7) and state machine (*Driver State Machine (*DSM*)*, e.g., Figure 8). The class diagram contains attributes and operations that are specifically needed to model DSM such as attribute *isCorrectInput* of *Driver Component* in Figure 7. A DSM is a UML state machine that is specifically defined to trigger the execution of BSMs based on real operational data. Data types of the real operational data (e.g., *x* and *y*) are specified in the class diagrams of BM. Data on signals (e.g., *SignalD* of Figure 4) are sent via ports (e.g., *dcport* in Figure 3) from DSM to BSMs. Such data includes the input of a system actor, environment changes, etc. A DSM can also be enriched with UAL to make it executable. The class diagram and DSMs are all together called Driver Model (DM).

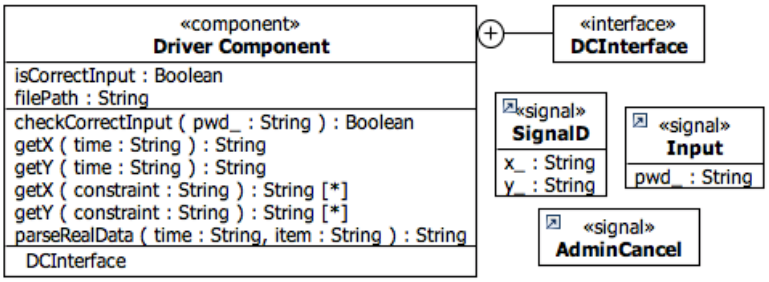


Figure 7. Class Diagram of DM (Running Example)

The DSM of our running example is shown in Figure 8. The DSM has two regions, i.e., the top region is used to communicate with *A*, whereas the bottom region is for communicating with *B*. In the top region, there is only one state called *Sending Data* having a self-transition "after 0.01s". This means that every 0.01 seconds, data is sent from *Driver Component* to *A*. The following UAL code is embedded inside the entry activity of the *Sending Data* state:

```
dcPort.send(new SignalD(getX(time), getY(time)));
```

The above code obtains values of *x* and *y* at a given point in *time* (from real operational data) and sends them to *A* via *SignalD* through *dcPort*. In the bottom region, in the *Sending Input Data* state, the following UAL code is added:

```
String pwd_ = parseRealData("input_pwd");
      dcPort.send(new Input(pwd_));
isCorrectInput = checkCorrectInput(pwd_);
```

The above code obtains the password from real operational data (the first line), sends it to *B* via the *Input* signal through *dcPort* (the second line), and checks whether or not the password is correct with the *checkCorrectInput (pwd_)* operation in the class diagram of *Driver Component*.

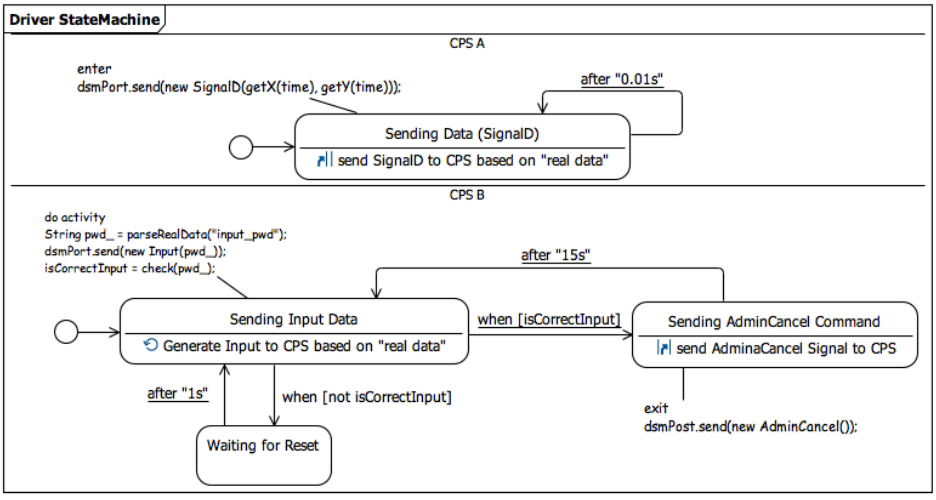


Figure 8. State Machine of DM (Running Example)

5 ARCHITECTURE AND CURRENT IMPLEMENTATION OF *UNCERTOLVE*

In the rest of the section, we first present the overall architecture of *UncerTolve* (Section 5.1), followed by the current implementation of *UncerTolve* (Section 5.2).

5.1 Architecture

The architecture of *UncerTolve* is represented in Figure 10. The key input of the architecture is real operational data collected from existing deployments. Real operational data can be collected continuously; therefore a process of using *UncerTolve* for evolving BMs can be iterative. As long as new operational data available, *UncerTolve* can be used to evolve the current BM and therefore the evolved BM can be used to generate new test cases for testing new/future deployments. Real operational data are invoked by *Driver Model* and *Executable Belief Test Ready Model* (executable UML) for enabling model execution. Model execution results (i.e., discovered previously unknown objective uncertainty measurements and errors in the initial BM) are used to evolve the BM with a set of heuristics (i.e., *tolveR-E*). Real operational data are also used to support the dynamic invariant

inference, which produces *Invariants*, representing either test oracles (i.e., state invariants) or test data specifications (i.e., guard conditions). Results of the inference are used to further evolve the belief BM, based on another set of heuristics: *tolveR-D*.

Figure 2 shows the necessary components of the *UncerTolve* architecture. An initial *Belief Test Ready Model*, semantically equivalent *Executable Belief Test Ready Model*, *Driver Model*, and *Real Operational Data* are key artifacts that need to be constructed in order to use *UncerTolve*. Definitions and examples of these artifacts are presented in Section 4 for references. For each of the activities (i.e., modeling, model execution and invariant inference), a set of guidelines (i.e., *S1*, *S2*, and *S3*) is also defined to guide users through a non-trivial model evolution process. As part of the guidelines, *tolveR-E* and *tolveR-D* are the two sets of heuristics defined for refining the initial BM based on model execution and invariant inference results.

There are three evolution ports defined on *Belief Test Ready Model*: 1) following *tolveR-E*, based on *Execution Log* (output of model execution), to evolve UML class diagrams, composite structure diagrams and BSMs, 2) following *tolveR-D*, based on invariants derived via Dynamic Invariant Inference, to evolve test oracle and test data specifications specified as state invariants and guard conditions of BSMs, and 3) appending objective uncertainty measurements derived from model execution to the BM. Though, the *UncerTolve* architecture provides these three evolution ports, it is not necessary to use them all at once. Depending on needs and contexts, which one(s) to use and how to use them can be customized.

Note that this architecture is generic since it can be integrated with different technologies (e.g., different invariant inference engines) to achieve the same or similar objectives. Section 5.2 discusses the current implementation of this architecture.

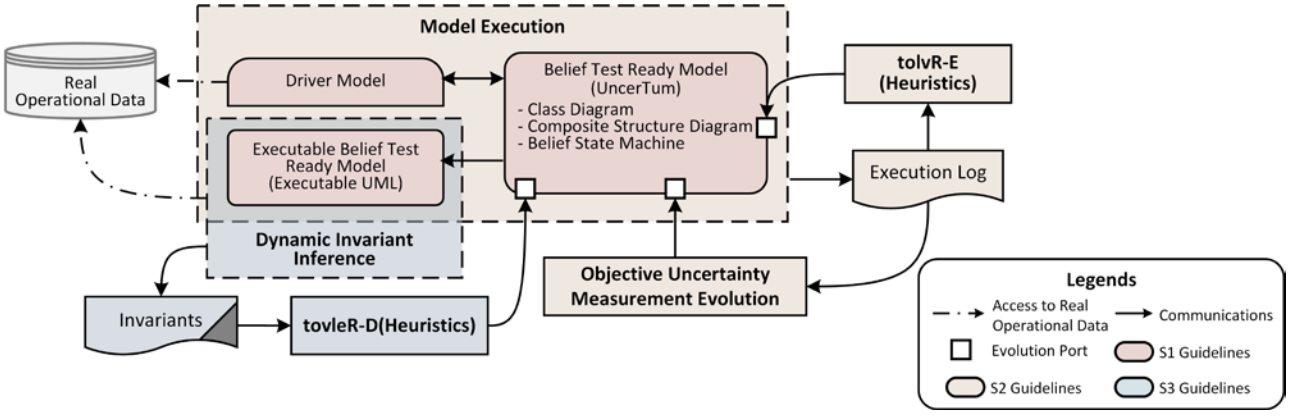


Figure 9. The Overall Architecture of *UncerTolve*

5.2 Current Implementation of *UncerTolve*

In this section, we discuss our current implementation of *UncerTolve*, focusing on the selection of

technologies and the integration of them. The overall workflow of the current implementation of *UncerTolve* is presented in Figure 10. The selection of technologies and corresponding justifications are summarized in Table 2. The recommended methodology for using the current implementation of *UncerTolve* is however discussed in Section 6.

Table 2. Steps, techniques/tools/languages, and corresponding justifications of the current implementation of *UncerTolve*

Step	Techniques/tools/languages	Justification of using selected techniques/tools/languages
S1, S2	IBM Rational Software Architect (RSA) IBM Simulation Toolkit UML Action Language (UAL) Eclipse OCL Java Implementation of heuristics *tolveR-E*: *Execution Logger* and *Log Analyzer*	The overall approach of the U-Test project is implemented in the CertifyIt[4] tool, which is a plug-in to IBM RSA. UAL is implemented based on the OMG Alf standard and is used by the IBM RSA Simulation Toolkit. Thus, IBM RSA, Simulation Toolkit, and UAL were selected in the current implementation of *UncerTolve*. Eclipse OCL is one of the commonly used OCL evaluation tools, which is built on EMF and fits well with the tooling of our overall technical solution. Given that execution log cannot be used automatically to modify BMs, heuristics *tolveR-E* are implemented to propose a set of actions to the user to modify BMs with uncertainty.
S2	Java Implementation of *Objective Uncertainty Measurement Analyzer*	Our approach is based on *subjective* uncertainty. To further validate it, we calculate the frequency (objective uncertainty measurements) based on the real operational data.
S3	Daikon Invariant Detector *Invariant Converter* (Java Implementation planned)	Several dynamic inference tools exist in the literature [1-3]; however, we decided to use Daikon because it implements a set of optimizations that facilitates its applications to complex problems [4].
S3	Java Implementation of heuristics *tolveR-D*: *Invariant Analyzer*	Daikon outputs invariants. Links must be established between the inferred invariants and models elements of BMs. Thus, we developed heuristics *tolveR-D* to link Daikon invariants with state invariants and guard conditions (specified as OCL constraints and representing test oracles and test data specifications) of the BMs.
S1-S3	Java implementation of the integrated solution (Figure 10)	None

The first activity is to develop and execute BMs (S1/S2 in Figure 10). This activity takes place in IBM's Rational Software Architect (RSA) and its Simulation Toolkit plugin [46]. As shown in Figure 10, *UncerTum* (Section 3.4) is currently implemented in IBM RSA. A user of *UncerTolve* develops BMs (S1 in Figure 10) using the guidelines developed for *UncerTum* (see [9]). To validate BMs, such models must be executed with real operational data. To achieve so, we extended *UncerTum* to provide a set of new guidelines to convert BMs into executable ones. The methodology for creating executable models is described in Section 6.1. Corresponding to BMs, equivalent Java code is automatically generated by the Simulation Toolkit [46], which can either be executed with the Simulation Toolkit or as a standalone Java application. The user executes the developed BMs with the real operational data (S1 in Figure 10) using the Simulation toolkit [46].

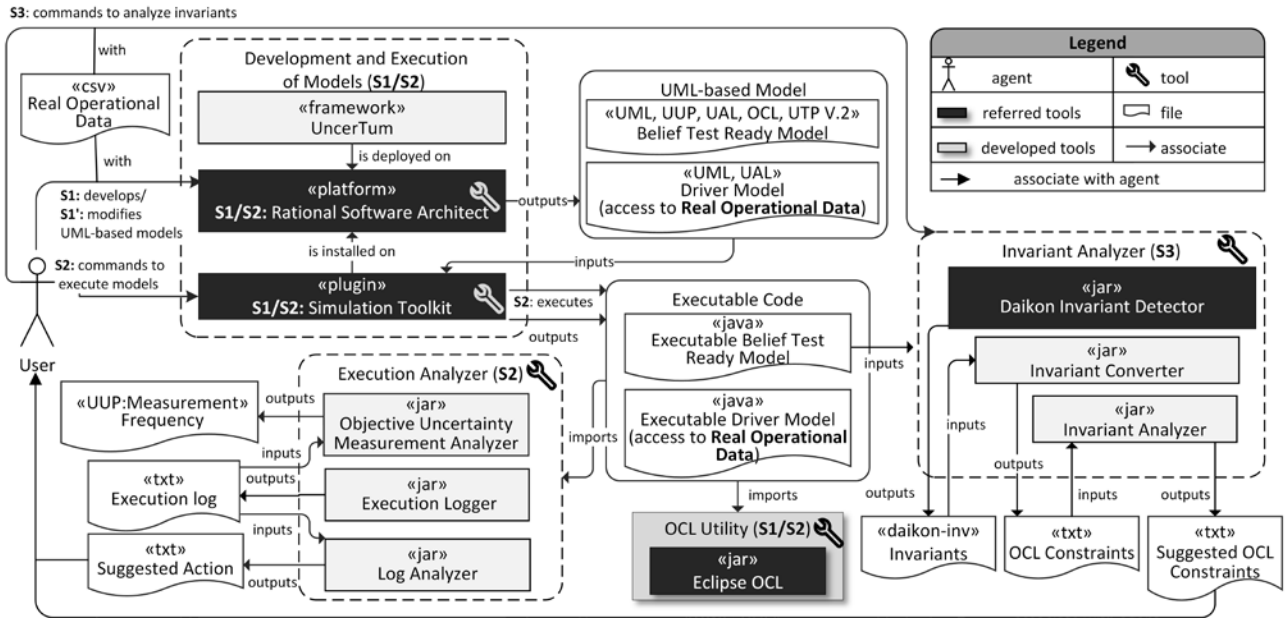


Figure 10. The Overall Work Flow of the Current Implementation of *UncerTolve*

The second activity of is *Execution Analyzer* (S2), which is used to analyze the results of the execution of BMs based on real operational data. *Execution Analyzer* is composed of *Execution Logger*, *Log Analyzer*, and *Objective Measure Analyzer*. Once the BMs are executed, *Execution Logger* logs the execution as execution log. The execution log includes information such as at one point of time, which trigger was fired with which data. Such log is used by *Log Analyzer* as an input to suggest various actions for the user to update BMs based on the set of heuristics of *tolveR-E* (Section 6.2.1). Based on suggested actions, the user may update BMs (S1' in Figure 10). This log is also used by *Objective Uncertainty Measurement Analyzer* to calculate conditional probabilities, e.g., the frequency of occurrence of a particular transition (details in Section 6.2.2).

The third activity is about the analysis of invariants using a machine learning technique implemented in the Daikon Invariant Detector [4]. The user may command (S3 in Figure 10) to infer invariants based on real operational data using the API we developed to invoke Daikon and access the real operational data. As a result, Daikon outputs a set of invariants, which are converted to OCL constraints by the *Invariant Converter* that we implemented in Java. The converted OCL constraints are inputted to *Invariant Analyzer* to further evolve invariants in BMs based on the set of heuristics of *tolveR-D* (Section 6.3), and the output is suggested OCL constraints as a feedback to the user. The user may accept, reject, or modify the suggested OCL constraints.

6 RECOMMENDED METHODOLOGY

The recommended methodology for applying the current implementation of *UncerTolve* is presented in Figure 11, from which one can see that it is iterative and has three sequential steps. In the rest of the

section, each of these steps is discussed in details. Section 6.1 presents our proposed modeling methodology to create executable BMs including activities for creating BMs and UML models for *Driver Component*; Section 6.2 presents a set of activities for validation BMs and DMs and evolve objective uncertainty measurements (S2); Section 6.3 presents the process of evolving BMs in terms of invariants using dynamic invariant inference (S3).

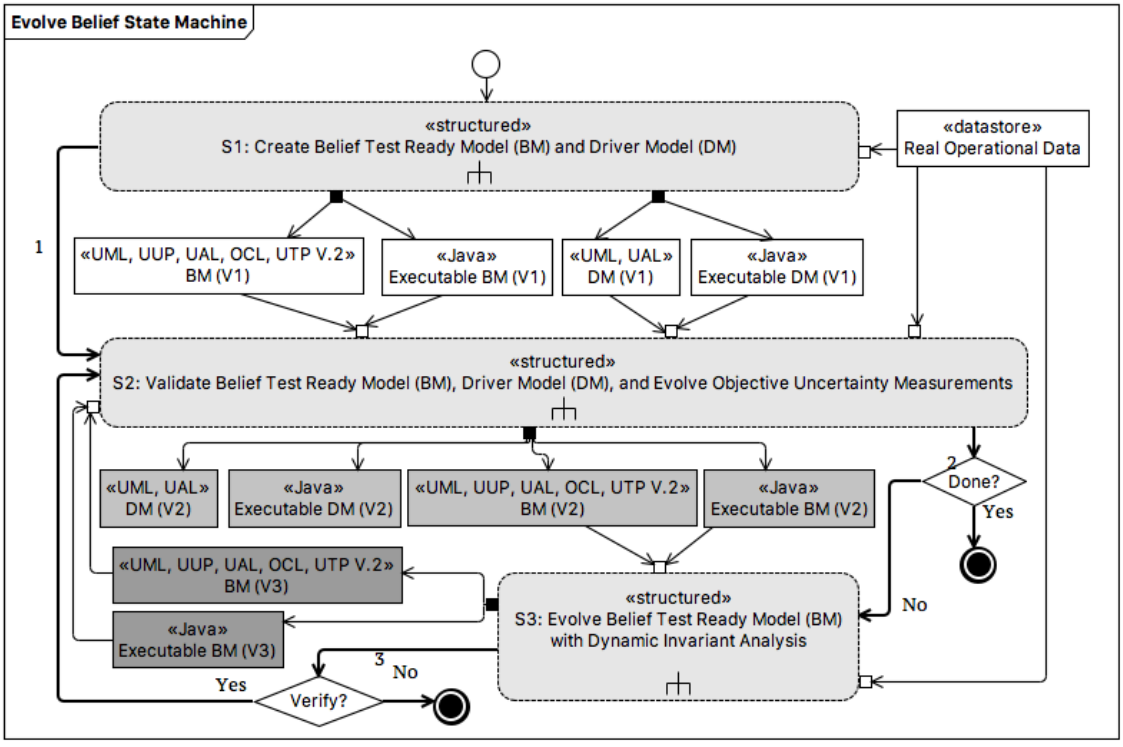


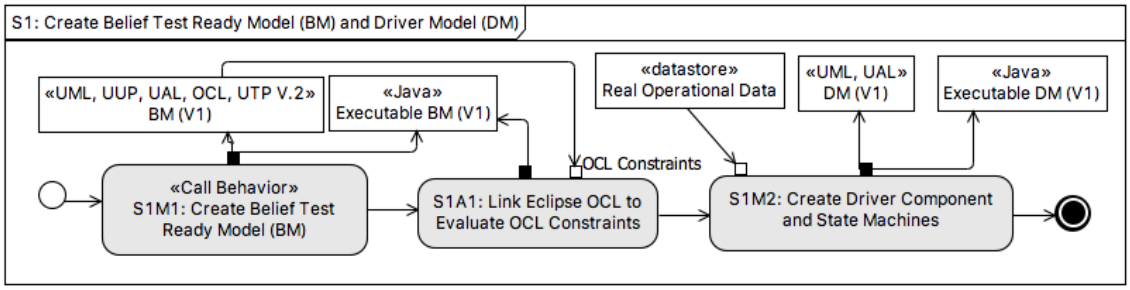
Figure 11. Recommended Methodology of using the Current Implementation of *UncerTolve*

6.1 Creating BM and Driver Model (S1)

As we previously discussed, with *UncerTum* [9], BMs can be created. Using *UncerTest*, executable test cases can be generated from the BMs specified in *UncerTum*. However, an extension of *UncerTum* is required to make BMs executable such that they can be validated against real operational data. This section only focuses on the aspects that are required to make BMs executable and other details on *UncerTum* are provided in [9]. As shown in Figure 12, S1 is broken down into three activities: S1M1, S1A1, and S1M2.

As the first step of the *UncerTolve* methodology, a modeler (belief agent) needs to apply *UncerTum* (which integrates the modeling notations of UML, UUP for specifying uncertainty, UAL for model execution, OCL for specifying constraints, and UTP V.2 for capturing testing aspects, e.g., «BeliefElement» in Figure 5 and Figure 6) to create BM (V1), i.e., the initial version of BM for a CPS under test (S1M1). UUP and UTP V.2 profiles are implemented in IBM RSA. As discussed in Section 4.1, the BM created by the modeler based on her/his subjective opinions and the BM is composed of a

set of class diagrams, a composite structure diagram and a set of BSMs.



S1M1, S1M2 – manual action; S1A1 -- automated action

Figure 12. The Structured Activity of Create Belief Model and Driver Model

To make the BM executable, UAL code should be added to relevant model elements of BSMs of the BM such as *entry*, *exit*, and *do* activities of a *state* and *effect* on a *transition*. The second output of S1M1 is Executable BM (V1), which is Java code automatically generated from the initial version of BM (V1) by IBM RSA, and can be automatically executed by the IBM Simulation Toolkit [46] or as a standalone program. For example, Figure 5 and Figure 6 present the diagrams of the BM (V1) of the running example. The key elements of the model have been discussed in Section 4.

Note that a modeler can specify a *subjective* uncertainty measurement as part of the applied «BeliefElement» on a model element on the BM model. For example, as shown in Figure 6, the subjective uncertainty measurement (denoted as *SM B1.2*) for «BeliefElement» applied on the transition from *Start* to *B1* is '*Likely*'. The transition from *Start* to *B2* however '*Unlikely*' occurs (see *SM B1.1*). Note that *SM* means Subjective Measurement and encoding of *BX.Y* means that the *X* round of the derivation of subjective uncertainty measurement for the *Y* element with «BeliefElement» applied.

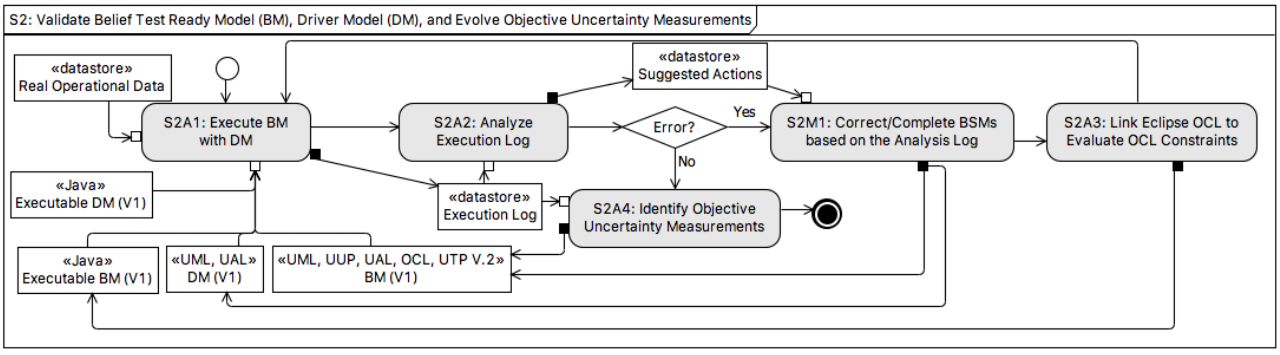
Since the executable UML implemented in IBM RSA doesn't support converting OCL constraints into Java code and consequently cannot evaluate constraints at the runtime, we implemented *OCLUtility* in Java. This utility links IBM Simulation Toolkit with the Eclipse OCL library to evaluate OCL constraints at runtime (S1A2). Using this activity together with S1M1, executable BM (V1) is developed. Notice that *OCLUtility* is generic and needed to be developed once.

Activity S1M2 is to connect the *Driver Component* to the BM using the same composite structure diagram developed for the BM (e.g., Figure 3), create class diagrams to keep information required to create a DSM, and create DSMs to drive the execution of BSMs for the purpose of validating and evolving them (Section 4.3). Recall that all these models together are called DM. The outputs of this activity are then DM (V1) and its equivalent Java code Executable DM (V1).

For our running example, we show the composite structure diagram of the BM in Figure 3 (which is shared with the DM), the class diagram in Figure 7, particularly developed for the *Driver Component*, and the DSM in Figure 8. Please refer to Section 4.3 for a detailed discussion of the DM model.

6.2 Validate BM and Driver Model, and Evolve Objective Uncertainty Measurements (S2)

The second step (Figure 13) is to validate the BM and DM against real operational data and evolve *objective* uncertainty measurements on the BSMs based on the real operational data. Note that *subjective* uncertainty measurements are the ones defined by the belief agent when the initial version of the BM was created.



* S2M1 – manual action; S2A1~S2A4 – automated action

Figure 13. The Structured Activity of Validating BM, DM and Evolving Objective Uncertainty Measurements

The first step (S1A1) automatically executes the BM with real operational data using the DM. Results of the execution are stored in the *Execution Log*. Note that the UAL code for generating the execution log is added in the DSM and BSMs for this purpose. The second step (S2A2) automatically analyzes the generated execution log to identify errors and obtain objective uncertainty measurements such as the frequency of the occurrences of a transition in a BSM. If an error is obtained, manual correction and completion of BSMs (S2M1), based on the analysis results obtained in S2A2 are then required. Sequentially, *UncerTolve* automatically establishes the link with Eclipse OCL (S2A3). The process of identifying errors continues until no error is identified, in which case *UncerTolve* automatically adds discovered objective uncertainty measurements to the BM (S2A4). Notice that the whole process of keeping updating the BM (V1) is continued until the validation is finished. At this moment, the BM (V2), along with the Executable BM (V2), DM (V2) and Executable DM (V2) are generated for S3 to take them as input to evolve BM (Figure 11). In the rest of the section, we discuss the key steps of S2.

6.2.1 Analysis of Errors and Fixing Models (S2A2, S2A3, and S2M1)

In S2A2, *UncerTolve* systematically and automatically checks the execution log for various types of errors. We classify errors into two high-level categories: *Syntactic* and *Semantic* errors. Syntactic errors are related to missing, incorrect, and redundant model elements in the BM and DM. For example, a redundant state means that its state invariant is subsumed by the state invariant of another state. A semantic error occurs when the models are syntactically correct, but the semantics of the models introduced using the UAL code have logical errors.

We proposed a set of heuristics for the validation purpose (i.e., *tolveR-E*, Appendix A) in *UncerTolve*. We provide below a subset of *tolveR-E* as examples:

1. If the state invariant of a state in a BSM evaluates to be *false*, then it leads to three possible fixing scenarios: adding a new state, changing an existing one, and/or deleting an existing one.
2. If a guard condition evaluates to be *false*, then it leads to two options: adding a new transition with an unknown trigger to an unknown state and changing an existing transition.
3. If a signal is sent from the DSM to a BSM (which is supposed to transit to a known state) but the signal is not received by the BSM, then this indicates that one or more model elements (e.g., connector) are missing from the BM model.
4. If a signal is sent from the DSM to a BSM (which it is supposed to transit to a known state) but the BSM transits to an unexpected state, it means that one or more model elements (e.g., the expected state) are incorrect.
5. If a signal is sent from the DSM to a BSM and more than one states of the BSM become active in one region at the same time, this may suggest redundant states.

Regarding the running example, one can observe the following changes to the *StateMachine_B* BSM of the BM (V1) (Figure 6): 1) adding new state *B3* (along with the definition of its state invariant as an OCL constraint), 2) adding two new transitions (between states *Start* and *B3*) and 3) applying «BeliefElement» on the two new transitions. The changes are reflected in the new version of the BSM (blue in Figure 14). This series of changes were triggered because, in S2A2, *UncerTolve* identified that the real operational data reflects the situation that from state *Start*, under the condition of *times*=4, the systems ends up at the *B3* state.

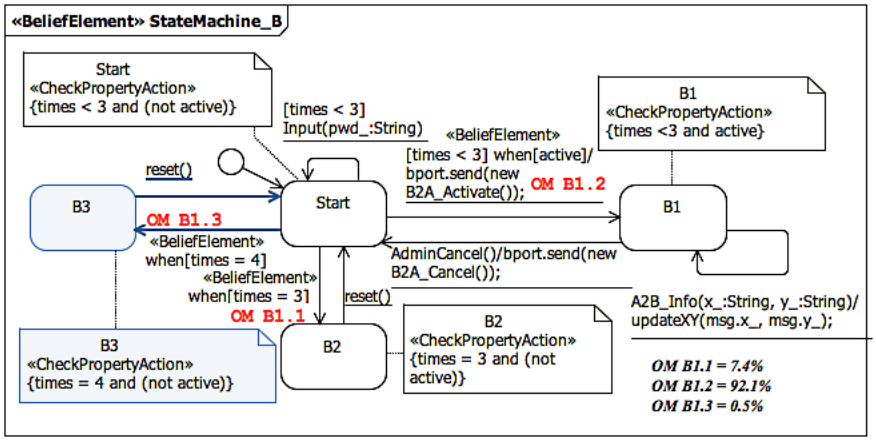


Figure 14. Belief State Machine of B (V2.1)

As discussed in Section 4, OCL constraints are used to specify state invariants (serving as test oracles) and guard conditions, which are for generating test data for the input parameters of associated triggers. Based on the real operational data, these OCL constraints are validated by executing the

executable BM and as the result, new constraints may be added or existing ones are changed by a user based on the suggested actions provided by *UncerTolve*. For example, as shown in Figure 14, a new OCL constraint is added to state B3 as its state invariant.

6.2.2 Identifying Objective Uncertainty Measurements (S2A4)

UncerTolve analyzes the execution log and calculates the frequency of traversing a state or transition, based on which it defines an *objective* uncertainty measurement for the state or transition. Especially for transitions, *UncerTolve* calculates conditional probabilities of the transitions leaving from the same state. For example, as shown in Figure 14, the *StateMachine_B* BSM of the BM (V2) contains three objective uncertainty measurements (i.e., *OM B1.1*, *OM B1.2* and *OM B1.3*). Note that *OM* means Objective Measurement and encoding of *BX.Y* means that the *X* round of the derivation of the objective uncertainty measurement for the *Y* element with «BeliefElement» applied. *OM B1.2=92.1%* implies that based on the real operational data, the probability of transiting from *Start* to *B1* via the transition is *92.1%*. Note that the subjective uncertainty measurement for this transition was initially defined as '*Unlikely*' by the modeler (Figure 6). In this case, the objective uncertainty measurement conforms to the subjective uncertainty measurement. In the case that a nonconformity is observed, *UncerTolve* alerts the modeler, but the evolving process of the models continues, as in steps S3 and S4, the objective uncertainty measurements might be updated, which provides more evidence to the modeler. The modeler can then decide whether or not to adjust her/his belief on the subjective uncertainty measurement in the next or future rounds of S2. Notice that more real operational data used in the evolving process leads to the higher precision of derived objective uncertainty measurements.

Intermediate versions of the subjective and objective uncertainty measurements can be saved such that different types of analyses can be performed and eventually advanced test generation strategies can be derived, which is one of the items of our future work.

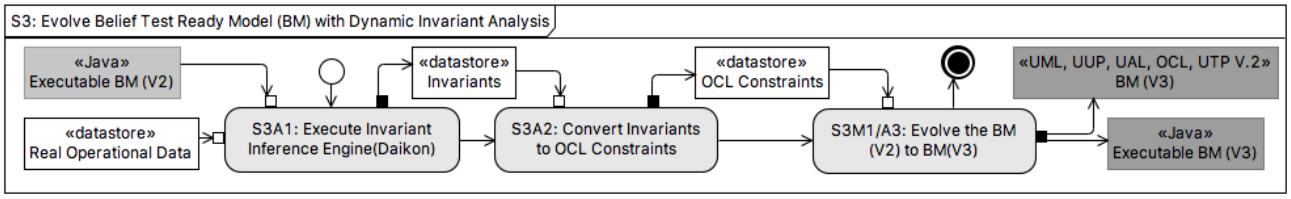
6.3 Evolve Belief State Machines with Dynamic Invariant Analysis (S3)

In the first step (S3A1), *UncerTolve* executes the Executable BM (V2), together with the real operational data in the Daikon tool, which produces a set of invariants (Figure 15). In S3A2, *UncerTolve* automatically converts Daikon invariants into OCL constraints. The obtained OCL constraints are then taken as the input of S3M1/A3 to evolve the BM (V2) to BM (V3). *UncerTolve* implements a set of heuristics for this step (see details in Appendix B), some of which are listed below as examples:

1. If an invariant inferred by Daikon supersedes an existing constraint, then there are three options for the modeler to manually evolve the models: 1) keep the original constraint, 2) split the original constraint such that one or more states (transitions) are newly introduced, or 3) keep the

original state (transition) but update the constraint.

2. If an invariant inferred by Daikon subsumes a set of existing constraints (named as *EConstraints*), there are three options for the modeler to manually evolve the models: 1) keep things unchanged (if the invariant inferred by Daikon is irrelevant), 2) merge the invariant inferred by Daikon with a set of existing states and transitions, corresponding to *EConstraints*, 3) create a composite state to group a set of existing states and transitions that are associated to *EConstraints*.



* S3A1 – manual action; S3A2 -- automated action; S3M1/A3 – semi-automated action

Figure 15. The Structured Activity of Evolve BM with Dynamic Invariant Analysis

In the running example, the input of S3 is Figure 14, and the output of S3 is Figure 16. In Figure 16, newly added and changed model elements are highlighted as green. Note that in the figure that state *B4* is newly introduced to the *StateMachine_B* BSM of the BM (V3). As a result, two transitions are added between *B4* and *Start*. Introducing the transition from *Start* to *B4* leads to the updates of a list of objective uncertainty measurements: *OM B1.1-OM B1.3*. This is because the sum of the objective uncertainty measurements for all the four transitions leaving state *Start* (to states *B1*, *B2*, *B3*, and *B4*) is *100%*; therefore, introducing a new transition triggers the change of *OM B1.2* (=*92.1%* in Figure 14) to *OM B2.2* (=*91.0%* in Figure 16). The objective uncertainty measurement for the newly added transition from *Start* to *B4* is calculated as: *OM B2.4* = *OM B1.2* – *OM B2.2* = *92.1% - 91.0% =1.1%*. The rationale behind the calculation is that in S3, *UncerTolve*, based on the real operational data, evolves the state invariant of *B1* by adding clause '*a.active*' to it, which leads to the discovery of the new state *B4* (whose state invariant contains the clause '*not a.active*', the negation of the newly added clause of *B1*'s state invariant). Therefore, *OM B2.2* and *OM B2.4* are the results of the splitting of *OM B1.2*. As also shown in Figure 16, the state invariants of states *B2* and *B3* are also updated in S3 by introducing the same clause '*not a.active*' to each of the invariant. The objective uncertainty measurements of *OM B1.1* and *OM B1.3* remain unchanged.

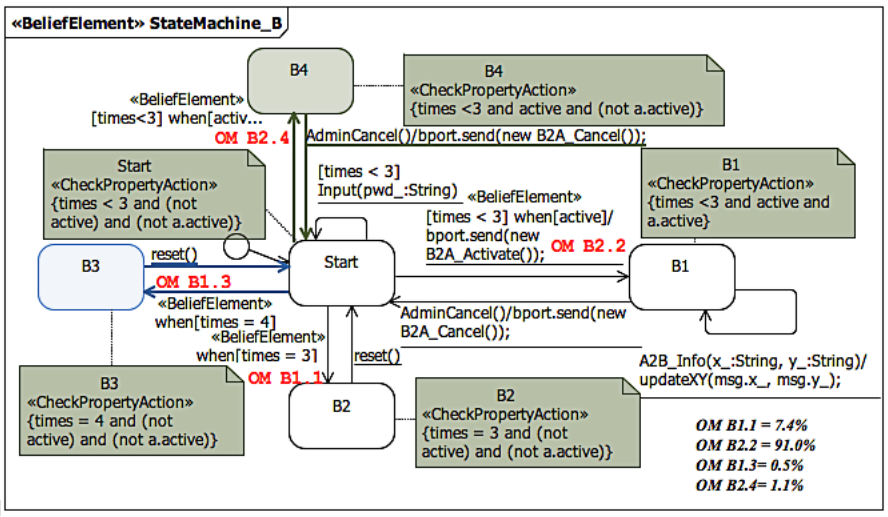


Figure 16. Belief State Machine of *B* (V3.1)

7 EVALUATION

In this section, we present the evaluation of *UncerTolve* as a proof-of-concept using the industrial case study available to us as part of the project. The case study is called GeoSports (GS) from the healthcare domain provided by Future Position X, Sweden[2]. The GeoSports case study is about monitoring Bandy players for their performance and health conditions during the game for early intervention and prevention. Coaches use data produced by the GeoSports system to improve the performance of individual players and the team together. We had access to real operational data of five real games that were used to evaluate *UncerTolve*. The first versions of the BMs with uncertainty were developed with *UncerTum*, together with the industrial partner during the four workshops hosted at its site. Below, we present the results of evaluation according to each key activity (i.e., S1, S2 and S3, Section 7.1 to Section 7.3). Section 7.4 presents the results of the overall validation of the final evolved models. Section 7.5 presents discussion and experiences. In 7.5, we report the effort required to build the test ready model of the GS case study, and the possibility of adopting UncerTolve in a commercial tool setting. The threats to validity are discussed in Section 7.7.

7.1 Results of Creating BM and DM (S1)

Table 3 presents the descriptive statistics of the initial versions (V1) of BMs and driver models for the case study. The #C column shows the total number of classifiers (including classes, components, signals, interfaces and data types) defined in a BM/DM. The #R column presents the total number of relationships among the classifiers such as associations and compositions. The #RP column presents the total number of signal receptions specified in all the class diagrams of a BM/DM. Similarly, for the

[2] www.fpx.se

composite structure diagram (CSD) developed for a BM, we present the total number of ports (#P) and connectors (#CN). For the state machines, we present the total number of states (#S) and transitions (#T). The #BE column presents the total number of model elements in a BM, where the «BeliefElement» stereotype was applied. For each driver model, we present the total number of classes and components (#C), states (#S) and transitions (#T).

Based on the descriptive statistics shown in Table 3, GS has the belief model with 62 classifiers, 56 relationships and 37 signal receptions in the class diagrams, 10 ports and 11 connectors in the composite structure diagram, and 82 states and 106 transitions in all the state machines of its BM.

Table 3. Descriptive statistics of the initial BMs (V1)* of the GS case study

#	Class Diagram			Composite Structure Diagram		State Machine		#BE
	#C	#R	#RP	#P	#CN	#S	#T	
Belief Model (BM)	62	56	37	10	11	82	106	49
Driver Model (DM)	1	0	0	2	NA	5	11	NA

C: Classifiers, R: Relationships, RP: Signal Receptions, P: Ports, CN: Connectors, BE: Belief Elements

7.2 Results of Validation and Evolution via Model Execution (S2)

Table 4 summarizes the results of S2 for the GS case study. We provide the total number of missing model elements (#MS), incorrect model elements (#IN), and redundant model elements (#RD). In addition, we report the total number of errors discovered in the semantics of the models (#SM). We report these descriptive statistics for the model elements of the BSMs of a BM: states (S), transitions (T), and elements with «BeliefElement» (BE) applied. Similarly, we report the statistics for the DSM of a BM, communications between the BSMs and the DSM and vice versa (BSM2DSM and DSM2BSM). In addition, we present the percentage of the elements evolved as compared to V1 in the % row with the following formula: (#MS - #RD)/(#V1 + #MS - #RD), where #V1 is the total number of the model elements of a BM V1 (the initial version of the BM, comparison baseline).

Table 4. Results of BM V2 and DM V2*

	Belief Model						Driver Model			
	BSM			BSM2DSM			DSM		DSM2BSM	
	#BE	#S	#T	#RP	#P	#CN	#S	#T	#RP	#P
#Missing	9	11	12	1	0	1	0	0	0	0
#Incorrect	0	3	3	2	1	2	0	1	0	0
# Redundant	0	1	2	1	0	0	0	0	0	0
#Semantic Problems	0	2	1	0	0	0	1	1	0	0
%	37%	11%	9%	0%	0%	8%	0%	0%	0%	0%

* BSM: Belief State Machine, DSM: Driver State Machine, RP: Signal Receptions, P: Ports, CN: Connectors, BE: Belief Elements, BSM2DSM: BSM to DSM communication, DSM2BSM: DSM to BSM communication, MS: Missing Model Elements, IN: Incorrect Model Elements, RD: Redundant Model Elements, SM: Semantic Problems, %: Percentage of evolved elements as compared to V1.

As it can be seen from Table 4, *UncerTolve* evolved 37% of the belief elements, 11% and 9% of states and transitions in the BM V2 as compared to the BM V1. Notice that the loop inside the S2 activity (S2A1→S2A2→S2M1→S2A3→S2A1, Figure 13) was executed seven times until no further errors were discovered. In addition, 8% of connectors for enabling the communications from the BSMs to the DSM (BSM2DSM) were evolved as shown in Table 4.

Table 4 also presents the errors discovered in the BMs and DMs of GS. For example, as shown in

Table 4 (#SM row, DSM column in GS block), *UncerTolve* found two semantic errors in its DSM, one error state, and one error transition. These two semantic errors are located in the UAL code in the entry/do/exit activity of the error state and the effect of the error transition. Notice that since the semantic errors were located in the UAL code of the DSM, it does not result in the evolution of any BM model element. This is why the % row in the DSM column for GS shows 0% for both #S and #T.

7.3 Results of Dynamic Inference (S3)

Table 5 shows the results of activity S3 for the case study. The #EP column presents the total number of model elements in the BSMs of a BM. These model elements were the points of evolution (e.g., State S4 in Figure 16). The #RF column presents the total number of refined model elements (e.g., state invariants in OCL and belief elements). The #ES column presents the total number of states that were newly added to or deleted from BSM V3 as the result of evolution. The #ET column represents the total number of transitions that were added to and deleted from BSM V3 as the result of evolution. Finally, the #EB column represents the total number of belief elements that were added to or deleted from BSM V3. The % row for #EP provides the percentage of model evolution points as compared to the total number of model elements in BSM, i.e., #EP/(#S+#T). Similarly, the percentage of #RF is calculated as #RF/(#S+#T). For ES, the percentage is calculated as #ES/#S, for ET as #ET/#T, and for EB as #EB/(#S+#T). In these formulas, #S and #T represent the total number of states and transitions in BSM V3.

Table 5. Results of the evolution of BSM V3*

Model Element Type	# Evolution points	# Modified/ Refined Elements	# Evolved States	# Evolved Transitions	# Evolved Belief Elements
#	5	56	8	18	32
%	2%	27%	8%	13%	29%

*#Evolved States: the total number of evolved states excluding the modified ones, #Evolved Transition: total number of evolved transitions excluding the modified ones, #EB: total number of evolved belief elements excluding the modified ones

As shown in Table 5, *UncerTolve* identified 5 model elements (2%) that can be evolved, whereas 27% of the existing model elements were refined. In the case of states, transitions, and belief elements, 8% of states, 13% of transitions, and 29% of belief elements were added/deleted to BSM V3 as compared to V2.

7.4 Overall Validation

Once the evolved version V3 of the BSMs was obtained after the S3 activity, we verified it with the real operational data by performing the S2 activity once again. Results are shown in Table 6. As one can see from the table, we found 11 validation problems in belief elements, whereas we discovered 3 validation problems with states and 2 with transitions. In total, we verified 99 belief elements. For states, we verified in total 100 states. Similarly, for transitions, we verified in total 134 transitions, but we couldn't verify 5 transitions once again due to unavailability of real operational data.

Table 6. Results of the validation of BSM V3*

Model Element Type	#Belief Elements	#States	#Transitions
#Missing	0	0	0
#Incorrect	0	0	0
#Redundant	11	0	0
#Semantic Problems	0	3	2
#Model Elements	99	100	134

Table 7. Overall results of the evolution across the versions (%)

Model Element Type	% Belief Elements	%States	%Transitions	%Evolution Points
V1				
V2	37%	11%	9%	19%
V3	29%	8%	13%	11%
V3'	-11%	0%	0%	1%
M= (#V3'-#V1)/#V3'	51%	18%	21%	

* V3': Verified version of V3 with S2, M is (#V3'-#V1)/#V3', - means not applicable.

Table 7 shows the results of the percentage increase in the number of evolved model elements of BM across the three versions (V1 to V3). In addition, we also show the percentages for the verified version of V3, i.e., V3'. The last column shows the mean percentage of increase in the number of model elements in V3' as compared to V1 and is calculated as M=(#V3'-#V1)/#V3', where #V3' is the number of model elements in V3' and #V1 is the number of model elements in V1. For EP, we also show the mean percentage of increase in the evolution points in BM from V1 to V2, from V2 to V3 and from V3 to V3'.

As shown in Table 7, in the stage of V3', 51% of belief elements, 18% of states, and 21% of transitions were evolved as compared to the first version (V1). For EP, 19% of new evolution points were discovered in V2, 11% in V3, and 1% in V3'.

7.5 Effort to Build Belief Test Ready Models and Adoption of *UncerTolve*

The BMs of GS were initially built by Simula researchers (the first three authors of the paper). These models were further confirmed with the industrial partner (last author of the paper). First, the first author (second year Ph.D. candidate) created the first version of the models, which were iteratively discussed with the second (a senior scientist) and third (a chief scientist) authors. Second, two workshops (2 days each) were held to present and discuss the models with the industrial partner to check their conformance with real scenarios. Third, the Simula researchers modified the models and as a result, the final version of models was produced that was used as input of *UncerTolve* (Figure 11). Table 8 shows the rough estimate of efforts for developing the models and presenting them to the industrial partner.

We classify effort in terms of how much time it took to build the models using standard UML notations and additional effort to apply various profiles and model libraries defined in *UncerTum*. As shown in Table 8, for standard Class/Composite structure diagrams, it took 37.5 hours (about a week), whereas it took additional 3.5 hours to apply *UncerTum* profiles and model libraries. For standard UML state machines, it took 52.5 hours and an additional 12.5 hours for *UncerTum* modeling. The last column shows additional effort required with *UncerTum* as compared to standard UML, i.e., roughly

15%. The last row of Table 8 shows the effort we spent to present the models to our industrial partner.

Table 8. Efforts in terms of time (hours) to develop and present BMs

	Class/Composite Structure Diagrams		State Machine		
	Standard UML Modeling	UncerTum Modeling	Standard UML Modeling	UncerTum Modeling	% of Time
Effort to develop	37.5	3.5	52.5	12.5	15%
Effort to present	7.5		15		-

In the project [10], we have a dedicated tool vendor (Easy Global Market [3]) responsible for implementing research results including *UncerTolve* into Smartesting's commercial model-based testing tool called CertifyIt[4] and transfer of the results to the industrial partners. Such adoption of the *UncerTum*, *UncerTest*, and *UncerTolve* is on-going and will be completed by the end of the project.

7.6 Discussion and Experiences

In this section, we present discussion and our experiences of applying *UncerTolve* to the industrial case study, based on the results presented in Sections 7.1-7.4.

Based on our experience of designing drivers for model execution and evolution, we discovered that the design of a driver is highly dependent on the characteristics of a CPS. For example, in our case, we have no direct access to its testing API or internal states. In addition, GS doesn't provide feedback to its users, i.e., Bandy players. It only records the readings from the Bandy players and transmits these via radio connections to the central system, where these data are processed. Because of these two characteristics of the CPS, the driver for GS was simpler since there was less information available for GS. In addition, the feedback from the CPS to the driver was not required to be modeled in GS. This might not be the case for other CPS case studies where we may have direct access to testing APIs and there is feedback from CPS, which will consequently lead to complex driver design.

In our case study, time events were used in models to capture timing aspects. Consequently, this had an impact on designing BMs, DMs and model evolution. However, GS only sampled data after a fixed interval of time and thus the design of BMs was simpler, which may not be the case for other case studies that have much more complex timing constraints.

In terms of the generalization of *UncerTolve*, theoretically speaking, as long as BMs of a CPS is specified in *UncerTum* [9] (a generic modeling methodology to create BMs of CPSs with subjective uncertainty) and real operational data are available, it is applicable to any case study. Our proposed modeling methodology (reported in Section 6.1) to create executable BMs is also generic and can, therefore, be tailored. In addition, our heuristics rules to update models based on the results of validation of executable models (Section 6.2), the process of calculating and abstracting objective

3 www.eglobalmark.com
4 www.smartesting.com/en/certifyit/

uncertainty and reflecting them in BMs (Section 6.2.2), and rules to evolve BMs (Section 6.3) are not specific to any case study and are thus generic. At its current state, we assessed *UncerTolve* with one CPS case study from the EU project as a proof-of-concept. In order to provide further evidence related to the generalization of *UncerTolve*, we indeed need to conduct additional case studies, which will be the focus of our near future work.

7.7 Threats to Validity

Internal validity threats in our context are due to the use of existing tools, including Daikon and IBM Rational Simulation Toolkit. Notice that Daikon has been extensively used in the literature for dynamic inference of invariants as we discussed in Section 2 and thus the chances of results being impacted by its use are minimum. IBM Rational Simulation Toolkit is a commercial product that we used for model execution and has a well-tested implementation. Therefore, it is highly unlikely that the results were impacted by its use as well. As part of an academic initiative by IBM, we were able to use fully functional version free of any cost.

Currently, we evaluated *UncerTolve* with only one industrial case study; however, to generalize the results, *UncerTolve* must be evaluated with other case studies. We plan to conduct additional industrial CPS case studies in addition to using the same case study with additional real operational data.

8 Conclusion

Given that Cyber-Physical Systems (CPSs) are tested with the assumptions on its internal behavior, its operating environment, and potential deployments, it is necessary that belief test ready models (BMs) developed to test the CPSs are continuously evolved using their real operational data including observed uncertainties. Such evolved models can be used to generate additional test cases to be executed on the current and future deployments of the same CPS. To this end, we proposed a test ready model evolution framework called *UncerTolve*. The framework was specially designed and developed to evolve BMs of CPSs with explicitly captured subjective uncertainty. Our aim is to not only improve the quality of BMs and evolve captured uncertainty, but also potentially discover unspecified uncertainty.

UncerTolve used several methods to evolve the models and uncertainty measurements including validation and evolution using model execution with real operational data collected from the application of a CPS and evolving constraints with a machine learning technique implemented in Daikon—dynamic invariant detection tool based on real operational data. *UncerTolve* was evaluated as a proof-of-concept with one industrial CPS case study from the healthcare domain, where we managed to evolve 51% of belief elements, 18% of states, and 21% of transitions. In the future, we are planning to use the evolved models to generate additional test cases by defining new test strategies

focusing on the evolved parts of BM. Such test strategies will be implemented in our uncertainty-based test case generation and minimization framework called *UncerTest*. In addition, we are planning to conduct further case studies to evaluate *UncerTolve*.

9 ACKNOWLEDGMENT

This research was supported by the EU Horizon 2020 funded project (Testing Cyber-Physical Systems under Uncertainty). Tao Yue and Shaukat Ali are also supported by RCN funded Zen-Configurator project, RFF Hovedstaden funded MBE-CR project, RCN funded MBT4CPS project, RCN funded Certus SFI and the EU COST action MPM4CPS.

10 REFERENCES

[1] M. Boshernitsan, R. Doong, and A. Savoia, "From daikon to agitator: lessons and challenges in building a commercial tool for developer testing," in Proceedings of the 2006 international symposium on Software testing and analysis, Portland, Maine, USA, 2006, pp. 169-180.
[2] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "IODINE: a tool to automatically infer dynamic invariants for hardware designs," in Proceedings of the 42nd annual Design Automation Conference, Anaheim, California, USA, 2005, pp. 775-778.
[3] C. Ackermann, R. Cleaveland, S. Huang, A. Ray, C. Shelton, and E. Latronico, "Automatic Requirement Extraction from Test Cases," *Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky and N. Tillmann, eds., pp. 1-15, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.
[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99-123, 2001.
[5] D. B. Rawat, J. J. Rodrigues, and I. Stojmenovic, *Cyber-physical systems: from theory to practice*: CRC Press, 2015.
[6] S. Sunder, "Foundations for Innovation in Cyber-Physical Systems," in Proceedings of the NIST CPS Workshop, Chicago, IL, USA.
[7] E. Geisberger, and M. Broy, *Living in a networked world: Integrated research agenda Cyber-Physical Systems (agendaCPS)*: Herbert Utz Verlag, 2015.
[8] S. Hangal, and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. pp. 291-301.
[9] M. Zhang, S. Ali, T. Yue, and R. Norgre, *An Integrated Modeling Framework to Facilitate Model-Based Testing of Cyber-Physical Systems under Uncertainty,* Technical report 2016-02, Simula Research Laboratory, 2016; https://www.simula.no/publications/integrated-modeling-framework-facilitate-model-based-testing-cyber-physical-systems.
[10] S. Ali, and T. Yue, "U-Test: Evolving, Modelling and Testing Realistic Uncertain Behaviours of Cyber-Physical Systems," in Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). pp. 1-2.
[11] M. Zhang, S. Ali, T. Yue, and M. Hedman, *Uncertainty-based Test Case Generation and Minimization for Cyber-Physical Systems: A Multi-Objective Search-based Approach,* Technical report 2016-13, Simula Research Laborabory, 2016; https://www.simula.no/publications/uncertainty-based-test-case-generation-and-minimization-cyber-physical-systems-multi.
[12] M. Daun, J. Brings, T. Bandyszak, P. Bohn, and T. Weyer, "Collaborating multiple system instances of smart cyber-physical systems: a problem situation, solution idea, and remaining research challenges," in Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems, Florence, Italy, 2015, pp. 48-51.
[13] OMG, "Unified Modeling Language (UML)," June 2015.
[14] O. M. Group, "Object Constraint Language (OCL)," February 2014.
[15] M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren, "Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model," in Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA). pp. 247-264.
[16] OMG, "UML Testing Profile," April, 2013.
[17] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: dynamic symbolic execution for invariant inference," in Proceedings of the 30th international conference on Software engineering, Leipzig, Germany, 2008, pp. 281-290.
[18] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic, "Using dynamic execution traces and program invariants to enhance behavioral model inference," in 2010 ACM/IEEE 32nd International Conference on Software Engineering. pp. 179-182.
[19] T. Berg, B. Jonsson, and H. Raffelt, "Regular Inference for State Machines Using Domains with Equality Tests," *Fundamental Approaches to Software Engineering: 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, J. L. Fiadeiro and P. Inverardi, eds., pp. 317-331, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.
[20] L. David, M. Shahar, and K. Siau-Cheng, "Mining modal scenario-based specifications from execution traces of reactive systems," in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA, 2007.
[21] D. Lo, and S. Maoz, "Scenario-based and value-based specification mining: better together," *Automated Software Engineering,* vol. 19, no. 4, pp. 423-458, 2012.
[22] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Using Declarative Specification to Improve the Understanding, Extensibility, and Comparison of Model-Inference Algorithms," *IEEE Transactions on Software Engineering,* vol. 41, no. 4, pp. 408-428, 2015.

[23] G. Carlo, M. Andrea, and M. Mattia, "Synthesizing intensional behavior models by graph transformation," in Proceedings of the 31st International Conference on Software Engineering, 2009.
[24] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 178-189.
[25] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 345-354.
[26] L. Davide, M. Leonardo, and P. Mauro, "Inferring state-based behavior models," in Proceedings of the 2006 international workshop on Dynamic systems analysis, Shanghai, China, 2006.
[27] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in Proceedings of the 30th international conference on Software engineering. pp. 501-510.
[28] P. Tonella, C. D. Nguyen, A. Marchetto, K. Lakhotia, and M. Harman, "Automated generation of state abstraction functions using data invariant inference," in Automation of Software Test (AST), 2013 8th International Workshop on. pp. 75-81.
[29] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," in Proceedings of the 36th International Conference on Software Engineering. pp. 468-479.
[30] T. Berg, B. Jonsson, and H. Raffelt, "Regular Inference for State Machines with Parameters," *Fundamental Approaches to Software Engineering: 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings*, L. Baresi and R. Heckel, eds., pp. 107-121, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.
[31] O. Tony, H. Michael, F. Sebastian, H. Armand, P. Marc, B. Ivan, and B. Yuriy, "Behavioral resource-aware model inference," in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, Vasteras, Sweden, 2014.
[32] G. Mark, and S. Zhendong, "Javert: fully automatic mining of general temporal properties from dynamic traces," in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Atlanta, Georgia, 2008.
[33] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Software Engineering*, pp. 1-43, 2015.
[34] I. H. Witten, and E. Frank, *Data Mining: Practical machine learning tools and techniques*: Morgan Kaufmann, 2005.
[35] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. pp. 267-277.
[36] O. Raz, P. Koopman, and M. Shaw, "Semantic anomaly detection in online data sources," in proceedings of the 24th International Conference on Software Engineering. pp. 302-312.
[37] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in Proceedings of the 28th international conference on Software engineering. pp. 282-291.
[38] S. Hangal, and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in Proceedings of the 24th international conference on Software engineering. pp. 291-301.
[39] "U-RUCM: Specifying Uncertainty in Use Case Models," accessed 2017; http://zen-tools.com/rucm/U_RUCM.html.
[40] OMG, "UML Profile For MARTE: Modeling And Analysis Of Real-Time Embeded Systems," June, 2011.
[41] "JGrapht," accessed 2016; http://jgrapht.org/.
[42] B. Liu, *Uncertainty theory*: Springer, 2015.
[43] "Eclipse OCL," accessed 2016; http://www.eclipse.org/modeling/mdt/?project=ocl - ocl.
[44] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating Test Data from OCL Constraints with Search Techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376-1402, 2013.
[45] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A Search-Based OCL Constraint Solver for Model-Based Test Data Generation," in 2011 11th International Conference on Quality Software. pp. 41-50.
[46] "IBM RSA Simulation Toolkit," accessed 2016; http://www-03.ibm.com/software/products/en/ratisoftarchsimutool.

Appendix A. TOLVER-E

ocl.evaluate is a function that is used to evaluate the constraint specified in OCL. The result of the evaluation is either *true* or *false*. The **catch TriggerException** represents the exception in case none of the specified triggers occur. For example, $S1 \xrightarrow{Tran1} S1$, the event of the trigger of Tran1 is kind of TimeEvent and the effect of Tran1 is *"throw new TriggerException(S1.name)"*.

#	Definition in OCL	Suggested Action
R1	**context** State not **ocl.evaluate**(**self**.stateInvariant)	One of R1.1, R1.2, or R1.3 will be selected.
R1.1	State.allInstance->excludes(**self**)->select(s:State|**ocl.evaluate**(s.stateInvariant))->size()=0	Modify this State or Add an unknown State with applied «BeliefElement»
R1.2	State.allInstance-> excludes(**self**)->select(s:State|**ocl.evaluate**(s.stateInvariant))->size()=1	Add a new Transition with the same Trigger with applied «BeliefElement»
R1.3	State.allInstance-> excludes(**self**)->select(s:State|**ocl.evaluate**(s.stateInvariant))->size()>1	Check the redundant problem, and add the transitions with the same Trigger with applied «BeliefElement» to these states if they are correct
R2	**context** State **catch TriggerException** and **ocl.evaluate**(**self**.stateInvariant)	One of R2.1, R2.2, or R2.3 will be selected.
R2.1	**context** State self.outgoings->exists(t:Transition| t.triggers->exists(t:Trigger|t.event.**ockIsKindOf**(CallEvent)))	Check invocation of operation
R2.2	**context** State self.outgoings->exists(t:Transition| t.triggers->exists(t:Trigger|t.event.**ockIsKindOf**(SignalEvent)))	Check composite structure diagram and state machine of driver
R2.3	**context** State self.outgoings->exists(t:Transition| t.triggers->exists(t:Trigger|t.event.**ockIsKindOf**(TimeEvent)))	Check the TimeExpression
R2.4	**context** State self.outgoings->exists(t:Transition| t.triggers->exists(t:Trigger|t.event.**ockIsKindOf**(ChangeEvent)))	Check the ChangeExpression
R3	**context** State **catch TriggerException** and not **ocl.evaluate**(**self**.stateInvariant)	One of R3.1, R3.2, or R3.3 will be selected.
R3.1	Refer to R1.1	Add an unknown transition to an unknown state with applied «BeliefElement»
R3.2	Refer to R1.2	Add an unknown transition to a known state with applied «BeliefElement»
R3.3	Refer to R1.3	Check the redundant problem, and add the unknown transitions to these states if they are correct.
R4	**context** Transition not **self**.guards->forAll(c:Constraint| **ocl.evaluate**(c))	One of R4.1 or R4.2 will be selected.
R4.1	**context** Transition **self**.triggers-> exists(t:Trigger|t.event.**ockIsKindOf**(CallEvent))	Modify the guard of call event / Add new transition with applied «BeliefElement»
R4.2	**context** Transition **self**.triggers-> exists(t:Trigger|t.event.**ockIsKindOf**(SignalEvent))	Modify the guard of signal event/ Add new transition with applied «BeliefElement»/ Check the signal from DM

Appendix B. TOLVER-D

The value ranges to make constraint *true* is represented as below,

$$C(x_0, \dots x_n) = C_0(x_0) \cap \dots \cap C_n(x_n)$$

The possible situations whereby the invariant needs to be modified are described as follows (C^{org} represents the original constraint, and C^{dai} represents the invariant from daikon). Note that any of them should apply «BeliefElement» by default.

#	Description	
D1	$C^{org} \supset C^{dai}$, we suggest	
	1	The variables in both constraints are the same. For example, S1 $\xrightarrow{Tran1}$ S2, the state invariant of S2 is {x > 1}, then the invariant from Daikon is {x > 2}, so 1) Split S2 into two states with the same trigger, S1 $\xrightarrow{Tran1}$ S2.1 {x > 2} and S1 $\xrightarrow{Tran1}$ S2.2 {x > 1 and x ≤ 2}; 2) Modify the state invariant of S2 to {x > 2}; 3) No change
	2	The number of variables is more than the original constraints. For example, S1 $\xrightarrow{Tran1}$ S2, the state invariant of S2 is {x > 1}, then the invariant from Daikon is {x > 1 and y = 0}, so 1) Split S2 to two states with the same trigger, S1 $\xrightarrow{Tran1}$ S2.1 {x > 1 and y = 0} and S1 $\xrightarrow{Tran1}$ S2.2 {x > 1 and y ≠ 0}, 2) Modify the state invariant of S2 to {x > 1 and y = 0}; 3) No change
D2	$C^{org} \subset C^{dai}$, we suggest	
	1	The variables are the same. For example, S1 $\xrightarrow{Tran1}$ S2, the state invariant of S2 is {x > 1}, then the invariant from Daikon is {x > 0}, so 1) Merge relevant states which may reach (e.g. S2 $\xrightarrow{Tran2}$ S3 {x > 0 and x < 1}) to a composite state S1 $\xrightarrow{Tran1}$ {S2 $\xrightarrow{Tran2}$ S3}, 2) Modify the state invariant of S2 to {x > 0}; 3) No change
	2	The number of variables is less than in the original constraints. For example, S1 $\xrightarrow{Tran1}$ S2, the state invariant of S2 is {x > 1 and y = 0}, then the invariant from Daikon is {x > 1}, so 1) Merge relevant states which may reach (S2 $\xrightarrow{Tran2}$ S3 {x > 1 and z > 2}, S2 $\xrightarrow{Tran3}$ S3 {x > 1 and t = 0}) to a composite state S1 $\xrightarrow{Tran1}$ {S2 $\xrightarrow{Tran2}$ S3, S2 $\xrightarrow{Tran3}$ S3}, 2) Modify the state invariant of S2 to {x > 1}; 3) No change
D3	$C^{org} \cap C^{dai} \neq \emptyset$ and $C^{org} \not\subset C^{dai}$ and $C^{org} \not\supset C^{dai}$, we suggest	
	1	The variables are the same. For example, S1 $\xrightarrow{Tran1}$ S2, the state invariant of S2 is {x > 1}, then the invariant from Daikon is {x < 2}, so 1) Make intersections of C^{org} and C^{dai}, then S1 $\xrightarrow{Tran1}$ S2{x > 1 and x < 2}, 2) Use constraint from Daikon S1 $\xrightarrow{Tran1}$ S2{x < 2}; 3) No change
	2	The variables are different. For example, S1 $\xrightarrow{Tran1}$ S2, the state invariant of S2 is {x > 1 and y = 0}, then the invariant from Daikon is {y = 0 and z > 2}, so 1) Make intersections of C^{org} and C^{dai}, then S1 $\xrightarrow{Tran1}$ S2{x > 1 and y = 0 and z > 2}, 2) Use the constraint from Daikon S1 $\xrightarrow{Tran1}$ S2{y = 0 and z > 2}; 3) No change
D4	$C^{org} \cap C^{dai} = \emptyset$, we suggest	
	1	The variables are the same. For example, S1 $\xrightarrow{Tran1}$ S2, the state invariant of S2 is {x > 1}, then the invariant from Daikon is {x < 0}, so 1) the first solution is to make unions of C^{org} and C^{dai}, then S1 $\xrightarrow{Tran1}$ S2 ¬{x ≤ 1 and x ≥ 0}, 2) use constraint from Daikon S1 $\xrightarrow{Tran1}$ S2{x < 0}; 3) unchanged
	2	The variables are different. For example, S1 $\xrightarrow{Tran1}$ S2, the state invariant of S2 is {x > 1}, then the invariant from Daikon is {y = 0}, so 1) Make unions of C^{org} and C^{dai}, then S1 $\xrightarrow{Tran1}$ S2 ¬{x ≤ 1 and y ≠ 0}, 2) Use constraint from Daikon S1 $\xrightarrow{Tran1}$ S2{y = 0}; 3) No change