Uncertainty-aware Specification and Analysis for Hardware-in-the-Loop Testing of Cyber Physical Systems

Seung Yeob Shin[a,*], Karim Chaouch[a], Shiva Nejati[b,a], Mehrdad Sabetzadeh[b,a], Lionel C. Briand[a,b], Frank Zimmer[c]

[a] *Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg*
[b] *School of Electrical Engineering and Computer Science, University of Ottawa, Canada*
[c] *SES Networks, Luxembourg*

Abstract

Hardware-in-the-loop (HiL) testing is important for developing cyber physical systems (CPS). HiL test cases manipulate hardware, are time-consuming and their behaviors are impacted by the uncertainties in the CPS environment. To mitigate the risks associated with HiL testing, engineers have to ensure that (1) test cases are well-behaved, e.g., they do not damage hardware, and (2) test cases can execute within a time budget. Leveraging the UML profile mechanism, we develop a domain-specific language, HITECS, for HiL test case specification. Using HITECS, we provide uncertainty-aware analysis methods to check the well-behavedness of HiL test cases. In addition, we provide a method to estimate the execution times of HiL test cases before the actual HiL testing. We apply HITECS to an industrial case study from the satellite domain. Our results show that: (1) HITECS helps engineers define more effective assertions to check HiL test cases, compared to the assertions defined without any systematic guidance; (2) HITECS verifies in practical time that HiL test cases are well-behaved; (3) HITECS is able to resolve uncertain parameters of HiL test cases by synthesizing conditions under which test cases are guaranteed to be well-behaved; and (4) HITECS accurately estimates HiL test case execution times.

Keywords: Test Case Specification and Analysis, Cyber Physical Systems, UML Profile, Simulation, Model Checking, Machine Learning

1. Introduction

Cyber physical systems (CPS) are increasingly ubiquitous, and include many of the critical systems used in domains such as aviation, aerospace, automotive and healthcare. CPS are subject to extensive testing. A key testing activity is *Hardware-in-the-Loop (HiL) testing*, which is aimed at testing a CPS after the integration of the system's actual software and hardware. HiL testing – not to be confused with HiL simulation, where some or all the hardware components may be simulated (Jeruchim et al., 2000) – typically takes place at the far end of the system quality assurance spectrum and as part of acceptance testing (Ammann and Offutt, 2016).

An important characteristic of HiL testing is that, due to the involvement of actual hardware, HiL test cases need to account for physical behavior of hardware. Moreover, HiL test cases have the potential to damage the system under test (SUT) or its environment. This necessitates that engineers should verify HiL test cases, before these test cases are exercised on the actual system, to ensure that the test cases are *well-behaved*. That is, the test cases must implement valid test scenarios and not pose undue risks to the SUT or its environment. An example of a potentially damaging behavior is attempting to supply a voltage to a hardware component beyond the limits that the component has been designed to support. Although such an abnormal case may be useful for robustness testing of the CPS control

*Corresponding author
Email addresses: seungyeob.shin@uni.lu (Seung Yeob Shin), karim.chaouch@uni.lu (Karim Chaouch), snejati@uottawa.ca (Shiva Nejati), m.sabetzadeh@uottawa.ca (Mehrdad Sabetzadeh), lionel.briand@uni.lu (Lionel C. Briand), frank.zimmer@ses.com (Frank Zimmer)

software, this is not the objective during HiL testing. It is, therefore, important to ensure that HiL test cases are *well-behaved* before executing them on the actual hardware.

A second important characteristic of HiL testing is that the behaviors of HiL test cases are highly impacted by environmental factors, e.g., temperature, weather conditions, or the characteristics of hardware interacting with the SUT. Exact environment conditions and hardware characteristics are only known at the actual execution time of HiL test cases. Prior to the actual HiL testing, engineers have only partial and approximative knowledge about the SUT environment and the hardware interacting with the SUT. Hence, when checking well-behavedness of HiL test cases before the actual testing, they may not be able to conclusively determine whether, or not, a test case is well-behaved, i.e., whether it may incur any hardware damage. For example, the well-behavedness of a test case supplying voltage to an external device depends on the voltage range tolerated by the device. Hence, the test case may be safe for some devices and unsafe for others. Without knowing the exact device specification, however, we cannot ascertain the well-behavedness of the HiL test case. In this situation, engineers need to identify the conditions on the environment and hardware parameters under which HiL test cases are well-behaved. If the conditions are met at the time of testing, the well-behavedness of test cases is ensured and they can safely proceed with testing.

The third important characteristic of HiL testing is that the duration of testing is often limited. While time budget constraints apply to virtually all stages of system development and testing, there is an additional major factor at play for CPS HiL testing. Since many CPS are deployed in harsh environments, the time spent on HiL testing can cut directly into the service life of a CPS. For example, once launched into orbit, a satellite has an average lifespan of 15 years. A mere two-month-long HiL testing process – not uncommon for satellites – would reduce the active service life of the satellite by more than 1%. To develop HiL test plans that can run under tight time budget constraints, engineers need to draw up accurate a-priori estimates about the *execution time* of HiL test cases. Note that similar to the test behaviors, the execution time of test cases is also impacted by the uncertainty in the SUT environment and hardware. For example, a test case may take significantly longer to run when the hardware components of the SUT need to be re-calibrated during test execution, e.g., to adapt to the system's ambient temperature.

In this article, we develop an *executable* language for specifying HiL test cases and HiL platforms. Our language aims at enabling the three tasks described above: (1) checking well-behavedness of HiL test cases, (2) identifying conditions on the uncertain environment and hardware parameters under which HiL test cases are well-behaved, and (3) estimating the execution times of HiL test cases. These three tasks are performed *before* the actual HiL testing stage and using *models* of HiL test cases and the underlying HiL platform.

The benefits of model-based analysis for CPS are widely acknowledged (Lee, 2008; Jensen et al., 2011; Nguyen et al., 2017; Thacker et al., 2010; Clarke and Zuliani, 2011; Zheng and Julien, 2015). In particular and in the area of model-based testing, approaches exist for automated generation of CPS test cases (Arrieta et al., 2017b,a; Zhang et al., 2017). The test cases produced by these approaches are nevertheless partial and abstract, thus requiring considerable manual effort before they can be used as HiL test cases (Wang et al., 2015). Industry standards such as TTCN-3 (ETSI, b) and UTP (OMG, d) support detailed specification of tests in general. These standards, however, do not specifically address CPS HiL testing and are, on their own, inadequate for our analytical needs. From a conceptual standpoint, our work is distinguished from the existing work in that it is not motivated by the analysis of a SUT, but rather the analysis of the test cases exercised against a SUT. This type of analysis, which is a necessity for CPS HiL testing and potentially beyond, has not been sufficiently explored to date.

Contributions. The contributions of this article are three-fold:

1) A modeling language for specifying CPS HiL test cases. We develop the *Hardware-In-the-loop TEst Case Specification (HITECS)* language. HITECS is a textual language defined using the *UML profile mechanism* (OMG, b). A key characteristic of HITECS is that it has an execution semantics, and it includes specific constructs to capture uncertain and physical behaviors of CPS HiL testcases. HITECS customizes the UML Testing Profile (UTP) (OMG, d) and the UML Uncertainty Profile (UUP) (Zhang et al., 2019b) to the HiL testing context. To do so, HITECS further uses the textual syntax of the Action Language for Foundational UML (Alf) (OMG, a), adopting Alf's execution semantics. To represent uncertainty in the SUT

environment and hardware, HITECS allows engineers to declare uncertain variables and to associate them with probabilistic distributions. For physical behaviors of HiL testing, which are typically captured by equations, HITECS provides mathematical constructs. HITECS is a generic HiL test case specification language which is motivated by our work experience in collaboration with several CPS industries (Abdessalem et al., 2018; Ul Haq et al., 2020; Menghi et al., 2020; Nejati et al., 2019; Liu et al., 2019; Matinnejad et al., 2019) and the existing literature on HiL testing (Asadollah et al., 2015; Ali and Yue, 2015; Abdessalem et al., 2018).

2) Analysis framework. Leveraging HITECS, we develop a framework to: (i) ensure, via formal verification, that HiL test cases properly manipulate and interact with the SUT as well as any additional instruments that provide inputs to the SUT or monitor its outputs, (ii) identify, via simulation and machine learning (ML), conditions on uncertain parameters of HiL test cases under which the test cases are well-behaved, and (iii) estimate, via simulation, the execution times of HiL test cases and thus improve HiL test planning. For verification, we provide guidelines that help engineers systematically specify assertions regarding the well-behavedness of HiL test cases. We then apply an existing model checker, JavaPathFinder (Visser et al., 2003), to HITECS test specifications in order to determine whether they satisfy their assertions. Due to the uncertainty in the SUT environment and hardware, however, assertions cannot always be verified conclusively. For the inconclusive assertions, we provide an uncertainty resolution approach. The approach samples specific values from the parameters' value ranges and executes HITECS test cases for these values to determine if the assertions are satisfied or violated. An ML classification algorithm is then used to identify conditions from the sampled data points under which HiL test case assertions are likely to hold. We then use model checking to provably ensure that the assertions hold within the identified ranges. To simulate HiL test cases, HITECS provides customizable, side-effect-free annotations and a simulation engine, allowing engineers to approximate test case execution times based on, for example, expert knowledge and historical data.

3) Industrial case study. We evaluate HITECS using an industrial case study from the satellite domain. Our evaluation results show that: (i) HITECS is applicable in practice and capable of capturing industry HiL test cases; (ii) HITECS enables engineers to define more complete and effective verification assertions than those specified based on domain expertise alone; (iii) HITECS model checking can verify several satellite HiL test cases in practical time; (iv) HITECS uncertainty resolution allows engineers to identify conditions on uncertain SUT parameters under which HiL test cases are well-behaved; and (v) HITECS simulation provides accurate estimates for the execution times of satellite HiL test cases.

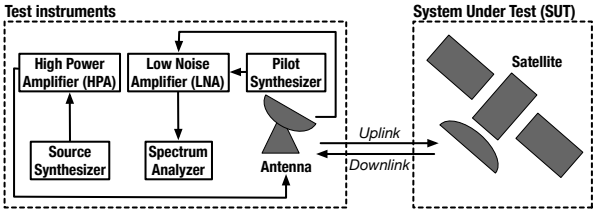


Figure 1: A simplified and partial view of the HiL test platform for a satellite after launch.

This article is an extension of a previous conference paper (Shin et al., 2018a) published at the ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018). This article offers important extensions over the previous conference paper by: (1) extending the HITECS specification language to account for the uncertain and physical behavior of CPS testing, (2) developing an uncertainty resolution approach which identifies conditions on uncertain parameters of HiL test cases ensuring that the test cases satisfy their assertions, (3) improving the evaluation of our approach accounting for the newly added uncertainty resolution method, and (4) describing a more thorough discussion and comparison of related work.

Structure. Section 2 motivates the article. Section 3 outlines our approach. Section 4 describes HITECS. Section 5 presents the HITECS analysis framework. Section 6 evaluates HITECS. Section 7 compares with related work. Section 8 concludes the article.

2. Motivating case study

We motivate our work with an industrial HiL testing case study from the satellite domain. Our case study is about *in-orbit testing* of satellite systems. In-orbit testing, which is part of the satellite HiL testing process, takes place after launching a satellite into orbit and before the satellite goes into

active service. Figure 1 shows a simplified test platform for in-orbit testing of a new satellite, which in addition to the satellite itself, involves a number of *test instruments*. Test instruments generate inputs to be fed to the SUT and monitor the SUT outputs. Specifically, the in-orbit test platform of a satellite includes, among other test instruments, an antenna for communication with the satellite and the following devices: synthesizers to generate input signals; spectrum analyzers to monitor and analyze the output signals; amplifiers to boost the power of signals being transmitted, or to filter out noise; and mechanical and electrical switches that determine the signal routing.

HiL test cases for in-orbit testing of a satellite typically include the following operations (Shin et al., 2018b): *setup*, *main*, and *teardown*. (1) The setup operation brings to a ready state the satellite as well as any test instruments used. This operation may further involve (re)calibrating the test instruments to ensure their accuracy under the environmental conditions at the time of testing. (2) The main operation exercises some satellite behavior based on the satellite's requirements. To do so, the main operation executes a sequence of steps. The following describes example steps of a main operation: First, signals (test inputs) with specific frequencies and power values are generated by the source synthesizer. The generated signals are then transferred to the antenna to be sent to the satellite. Finally, the satellite output signals are sent to the ground station and transferred to the spectrum analyzer so that they can be visualized and analyzed. (3) The teardown operation brings the satellite and test instruments to a standby state by performing cleanup operations on them. In our case study, teardown can, for example, result in reconfiguring certain parts of the satellite or the test instruments to save energy, and muting instruments to ensure that no undesirable signal is accidentally sent to the satellite.

To illustrate, Figure 2 shows part of the main operation of a HiL test case for testing a transponder function of the satellite under test. Note that the test case described in Figure 2 is a simplified version of the original textual test-scenario description from our collaborating partner, SES Networks, by excluding some satellite-specific details. The test case first measures environment conditions (line 8) and computes the initial power level for a signal to transmit to the satellite under test based on the measured environment conditions (line 9). The test case transmits an uplink signal to the satellite under test by controlling its power level (lines 12–14). The test case then measures a downlink signal generated by the satellite in response to receiving the uplink signal (lines 15–17). The test case iterates the transmission and measurement steps based on a predefined number of times (lines 11–17). As shown in Figure 2, the test case incorporates two assertion checks (lines 13 and 16). These assertions check the well-behavedness of the test case. Specifically, they prevent the satellite from being damaged by processing an overpowered signal.

```
1  Description TestTransferCurve
2  // prev_dPLv: previous downlink power level
3  // dPLv: downlink power level
4  // uPLv: uplink power level
5  // default: predefined initial power level
6  // limit: predefined maximum power level
7  // n: predefined number of iterations
8  env = measure environment conditions
9  uPLv = compute uplink power level based on env
10 prev_dPLv = default
11 repeat n times
12 |  uPLv = increase uPLv
13 |  check assertion uPLv < limit
14 |  send signal of uPLv to the satellite under test
15 |  dPLv = measure downlink power level
16 |  check assertion prev_pLv - dPLv < 1dB
17 |  prev_dPLv = dPLv
```

Figure 2: A simplified and partial description of the main operation of the transfer curve test case for testing a transponder function of the satellite under test.

During our collaboration with SES Networks, the engineers described a pressing need for automated techniques to support the following tasks in relation to HiL test cases:

Verifying HiL test cases. Like most CPS software, HiL test cases are complex and critical, and may contain faults. Faulty HiL test cases may generate invalid test results, may damage test instruments or the SUT, or may waste energy, time and other valuable resources. For example, a satellite may be damaged if the power of the signal sent to it exceeds its limits. To address this issue, in practice, engineers monitor states of HiL test executions, e.g., signal power level (line 13 in Figure 2), at the time of actual HiL testing on a satellite. However, runtime monitoring of HiL test cases is not able to guarantee the well-behavedness of HiL test cases due to its inexhaustive and in-situ testing nature. Hence, engineers need techniques to ensure that HiL test cases are well-behaved and exercise valid test scenarios prior to executing them on the actual HiL platform. In Section 3, we precisely define the

well-behavedness requirements that HiL test cases should meet.

Resolving uncertainty in HiL testing. Model checking HiL test cases to verify their assertions is limited in two ways: First, HiL test cases may contain complex mathematics involving non-algebraic functions (e.g., exponential or logarithmic computation). Such mathematical constructs cannot be typically handled by model checkers, in particular by JavaPathFinder (Visser et al., 2003). Measuring environment conditions (line 8 in Figure 2) often requires complex equations. For instance, spreading loss, representing the amplitude loss of a signal, is measured by using a logarithmic equation – $log(4 \cdot \pi \cdot sDist^2)$ where $sDist$ refers to a distance between a ground station and the satellite under test – which cannot be handled by JavaPathFinder. Therefore, the model checker cannot verify the assertion on line 13 since `uPLv` in the assertion depends on `env` measured at line 8 using complex non-algebraic functions. Second, as we will discuss in Section 4.2, we represent uncertain parameters of HiL test cases, i.e., the parameters whose values depend on the environment conditions or the final SUT hardware, using parameters with probabilistic value ranges instead of fixing their values. This again leads to model checking being inconclusive for HiL test cases. Specifically, JavaPathFinder can neither verify nor refute the assertion at line 13 in Figure 2 because the `limit` variable has a value range of tolerable signal power which depends on actual hardware characteristics. For some values in the value range of this variable, the assertion may hold, but the assertion may not hold for the other values. Therefore, test engineers need an uncertainty resolution technique to help them identify parameter value ranges under which HiL test cases conclusively satisfy their assertions, (i.e., are well-behaved).

Estimating the execution time of HiL test cases. In-orbit testing can take several weeks during which a satellite does not provide any service or revenue. The engineers thus have to carefully plan the HiL testing process and optimize its duration, knowing that delays can be extremely costly. To enable engineers to plan HiL testing effectively and to mitigate the risk of missing deadlines, they need to be able to accurately estimate the execution times of individual HiL test cases. As discussed earlier, the execution times of HiL test cases are impacted by environmental factors. For example, the execution times of satellite HiL test cases depend on whether the antenna is already pointing to the satellite under test or not. If not, test case execution may take longer since moving the antenna requires extra time. An immediate implication here is that test case execution times should be estimated as ranges instead of exact values.

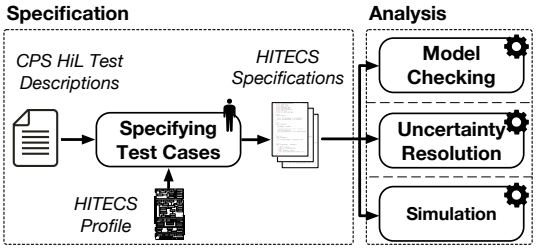


Figure 3: HITECS overview.

The analysis tasks discussed above are not specific to in-orbit testing of satellites conducted at SES Networks and are common in other CPS domains, as observed in both our work in collaboration with industry partners from other CPS domains such as IEE (Abdessalem et al., 2018; Ul Haq et al., 2020), LuxSpace (Menghi et al., 2020), QRA (Nejati et al., 2019), and Delphi (Liu et al., 2019; Matinnejad et al., 2019), as well as in the work of others (Mosterman and Zander, 2016; Lee, 2008; Zheng and Julien, 2015). In the next sections, we provide an approach for specifying and analyzing HiL test cases in such a way that the above tasks can be performed systematically and with computerized support.

3. Overview

Figure 3 shows an overview of our approach for the specification and analysis of CPS HiL test cases. The core component of our approach is a modeling language, called *Hardware-In-the-loop TEst Case Specification* (*HITECS*), defined to specify HiL test cases and to support effective automation of the three analysis tasks motivated in Section 2 and described below:

Model checking. We verify HiL test cases to ensure their well-behavedness. We define a HiL test case to be well-behaved if it satisfies the following requirements:

1. The test case properly initializes (resp. cleans up) the involved components before (resp. after) using them.
2. Before sending data to a component, the test case ensures that the component is in a state where it can process the data.

3. The test case ensures that any data sent to / received from a component is within the operating ranges of the component.

To enable the verification of HiL test cases for well-behavedness: (1) we provide guidelines for engineers to systematically specify the above requirements in terms of assertions inserted in HITECS test specifications, and (2) we apply model checking (Clarke, Jr. et al., 1999) to HITECS test specifications to determine whether the assertions hold. Due to the complex equation issues described in Section 2, HITECS model checking classifies test cases into two categories: *inapplicable* and *applicable*. We say a test case is inapplicable, if it has some behavior (i.e., complex mathematics) that cannot be handled by model checking. The applicable test cases can further be categorized as *conclusive* and *inconclusive* by HITECS model checking. We say an assertion is conclusive, if it can be verified or refuted within a given time budget. In this case, the assertion is either satisfied or violated for *all* the executions of the corresponding test case. We say an assertion is inconclusive, if neither the assertion nor its negation can be verified by the model checker because the assertion is satisfied by some test executions and is violated by some other test executions. Note that we remove the complex equations from inapplicable test cases and replace them by symbolic parameters whose values are unknown. This way the inapplicable test cases are turned into applicable test cases with uncertain parameters that may or may not be conclusive.

Uncertainty resolution. HITECS uncertainty resolution method aims to identify conditions on uncertain parameters of a test case under which the test case is well-behaved. HITECS uncertainty resolution combines the following two analysis tasks: sampling and classification. The sampling task randomly chooses values from the parameter value ranges of a test case, and then executes the test case with the sampled values. Given the execution outputs, the sampling task collects assertion results – satisfaction (true) or violation (false) – and label sampled parameter values with either true or false. The classification task then takes a labeled sampling dataset as an input and identifies (safe) conditions on parameter values such that the test case is likely to be well-behaved. We then apply the model checker to the test case augmented with the conditions on parameter values to ensure that the test cases are indeed well-behaved under those conditions.

Simulation. HITECS has an execution semantics, enabling the simulation of HiL test cases at an early stage and without the involvement of hardware. This in turn makes it possible to estimate the execution times of HiL test cases without having to exercise them against the SUT. More precisely, HITECS allows engineers to specify the execution time values for individual statements in a HiL test case based on, for example, expert judgment, historical data or analytical techniques. These values are subsequently used by the HITECS simulation engine to generate distributions that capture ranges of the actual execution times of HiL test cases.

4. Test specification

HITECS tailors the UML Testing Profile (UTP) (OMG, d), UML Uncertainty Profile (UUP) (Zhang et al., 2019b), and Action Language for Foundational UML (Alf) (OMG, a) to specify CPS HiL test cases. In Section 4.1, we provide background on UTP, UUP, and Alf, and in Section 4.2, we present HITECS.

4.1. Background on UTP, UUP, and Alf

UTP is a standard language based on UML for specifying common concepts in various testing approaches. As UTP is a profile of UML, it can be combined with other profiles and be extended or tailored to different development practices. The testing concepts in UTP are quite generic, and there is no existing work on tailoring or customizing these concepts to HiL testing. Further, UTP does not have a formal execution semantics, and cannot readily support the verification and simulation of CPS HiL test cases.

UUP is implemented based on U-Model (Zhang et al., 2016) which is a generic conceptual model for capturing uncertainties in CPS. The conceptual model of UUP, i.e., U-Model, provides a comprehensive description of uncertainties, classifies them, and can serve as a baseline for modeling uncertain behaviors of CPS. UUP enables the uncertainty-aware development of CPS and provides a set of model libraries to specify and measure various kinds of uncertainties, e.g., probability distributions.

Alf is a textual modeling language, specifying the UML modeling elements. Its primary goal is to provide an executable semantics for UML models (e.g., operations of classes). Alf specification fragments can be combined with UML models to

Table 1: HITECS contribution to UTP and UUP.

C#	Descriptions of HITECS contributions
C1	HITECS provides tailored concepts for CPS HiL testing
C2	HITECS provides quantitative means for capturing the degree of confidence about test oracles (verdicts)
C3	HITECS provides an explicit mechanism to express the uncertainties in the CPS environment
C4	HITECS provides an explicit means to specify physical behaviors in CPS testing and its environment
C5	HITECS enables model checking of HiL test cases for well-behavedness
C6	HITECS supports an uncertainty resolution method aiming to identify conditions on uncertain parameters of HiL test cases under which the test cases are well-behaved
C7	HITECS provides simulation facilities for estimating the execution time of HiL test cases

make them executable. Alternatively, Alf can be seen as a stand-alone language since, in addition to the UML behavior, it can textually represent the UML structure. The execution semantics of Alf is defined by mapping the Alf concrete syntax to the abstract syntax of the standard Foundational Subset for Executable UML Models (i.e., Foundational UML) (OMG, c).

4.2. The HITECS language

HITECS extends UTP, UUP, and uses the textual notation and executable semantics of Alf. Since UTP, UUP, and Alf are UML-based, HITECS can seamlessly combine them. The execution semantics of Alf provides a rich basis for verification and simulation. Based on our experience and feedback from practitioners, we find Alf's textual notation more suitable for HiL test cases than visual notations, since HiL test cases typically contain lengthy sequences of statements.

We identified the modeling concepts of HITECS by studying the UTP modeling elements, the uncertainty measures in UUP, the formalization of acceptance testing concepts in our earlier work (Shin et al., 2018b) and the CPS testing literature (Asadollah et al., 2015; Ali and Yue, 2015; Abdessalem et al., 2018). Table 1 outlines the main improvements and extensions that HITECS provides over UTP and UUP. Overall, HITECS provides seven new extensions that are instrumental either to specifying HiL test cases, or to analyzing them. In this section, we introduce HITECS by describing and illustrating the contributions outlined in Table 1. Figure 4 shows the HITECS profile. We use the UML profile mechanism to explicitly represent how

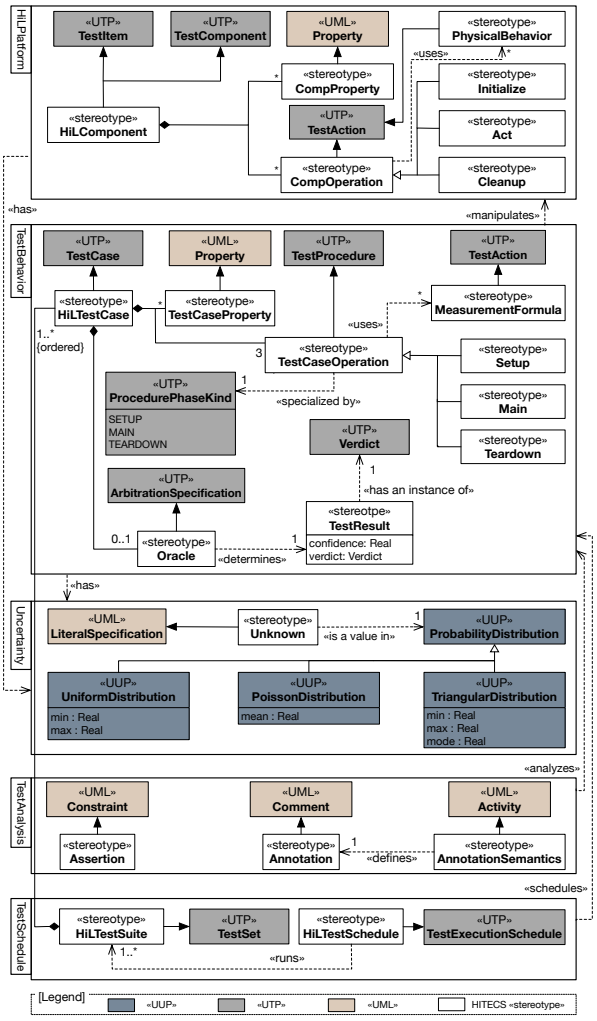


Figure 4: The HITECS profile (extension of UTP and UUP).

HITECS customizes and extends existing modeling concepts from UTP, UUP, and UML. As shown by the figure, HITECS is organized into five packages, `HiLPlatform`, `TestBehavior`, `Uncertainty`, `TestAnalysis` and `TestSchedule`, described below.

HiL Platform. A HiL platform is composed of the SUT and test instruments required to execute HiL test cases. In HITECS, these are defined by `HiLComponent` which extends UTP's `TestItem` (i.e, SUT) and `TestComponent` (i.e., test instruments). HITECS limits the visibility of `HiLComponent` to its own properties and operations due to the black-box nature of HiL testing. Specifically, HITECS guides test engineers to focus on specifying how HiL components are used by the test cases instead of capturing how they interact with one another.

Table 2: The HiLPlatform package stereotypes.

Stereotype	Description
HiLComponent	A HiL component is either the SUT or a (peripheral) test instrument required to execute a HiL test case
CompProperty	A property that characterizes a component state
CompOperation	An operation of a component that is called by a test case
Initialize	An operation to initialize a component
Act	An operation performing a main function of a component
Cleanup	An operation to cleanup a component
Physical-Behavior	A mathematical function capturing physical behavior of a component

Mapping (stereotype, specification): (HiLComponent, component specification), (CompProperty, property declaration), (CompOperation, operation specification), (Initialize, operation tagged as @Initialize), (Act, operation tagged as @Act), (Cleanup, operation tagged as @Cleanup), (PhysicalBehavior, physicalbehavior specification).

```
1  component Synthesizer {                                    HITECS
2  private frequency:Real;
3  private power:Real;
4
5  @Initialize
6  public init(in freq:Real,in power:Real){/*...*/}
7
8  @Cleanup
9  public cleanup() {/*...*/}
10
11 @Act
12 public generateSignal() {/*...*/}
13
14 @Act
15 public adjustPower(in degree:Real) {/*...*/}
16 }
```

Figure 5: HITECS specification of the synthesizer in Figure 1.

Table 2 describes the modeling concepts in the HiLPlatform package of Figure 4. The HiLComponent concept has CompProperty and CompOperation for capturing a component's attributes and operations, respectively. The operations of HiL components are categorized as Initialize, Act, and Cleanup and are respectively tagged by @Initialize, @Act, and @Cleanup annotations. These operations will be invoked by HiL test cases. Specifically, each test case specifies the operation calls in a sequence where @Initialize operations appear first, followed by @Act operations and the test case ends with @Cleanup operations. So, the order of the execution of operations is in fact defined by the order of their invocation specified in each test case. The Initialize operation of a component sets the component into a *ready* state from which it can execute its Act operations. Dually, the Cleanup operation moves the component into a *standby* state indicating that the component is not in use. For instance, Figure 5 is an example specification of a synthesizer which is used to generate input signals for the satellite under test (see Figure 1). The Synthesizer component has two attributes, frequency and power, specifying its output signals. The init() operation adjusts frequency and power to some desired values that can further be modified through generateSignal() and adjustPower() depending on the test case. The cleanup() operation turns off the synthesizer to ensure that it does not interfere with other components. In this article, we use /* ... */ (e.g., line 6 in Figure 5) in HITECS specification figures to omit irrelevant details.

```
1  component Satellite{                                       HITECS
2  /*...*/
3  @Act
4  public amplifySignal(in ul:Real):Real {
5    //minimum uplink power
6    x1:Real = Const::MIN_UL_POWER;
7    //uplink power at saturated point
8    x2:Real = Const::SAT_UL_POWER;
9    //minimum downlink power
10   y1:Real = Const::MIN_DL_POWER;
11   //downlink power at saturated point
12   y2:Real = Const::SAT_DL_POWER;
13
14   normUL:Real = (ul-x1) / (x2-x1);
15   normDL:Real = salehModel(normUL);
16   dl = normDL*(y2-y1) + y1;
17   return dl;
18 }
19 }
```

Figure 6: HITECS specification of the satellite in Figure 1.

```
1  physicalbehavior                                           HITECS
2  salehModel(in ul:Real):Real {
3    a:Real = Const::ALPHA * ul;
4    b:Real = Const::BETA * pow(ul,2);
5    dl:Real = a / (1.0+b);
6    return dl;
7  }
```

Figure 7: HITECS specification of the Saleh model (Saleh, 1981).

In HITECS, the PhysicalBehavior concept captures the behavior of a HiL component operation by mathematical formulas. For example, Figure 6 is a (partial) specification of the satellite under test which invokes the salehModel() physical behavior (line 15) specified in Figure 7. The amplifySignal() in Figure 6 specifies one of the ba-

Table 3: The TestBehavior package stereotypes.

Stereotype	Description
HiLTestCase	A test case description specifying test properties (inputs, outputs and HiL components), a set of test operations, assertions, simulation annotations and a test oracle
TestCase-Property	Inputs, outputs, and HiL components used by a test case
TestCase-Operation	An operation consisting of a sequence of statements involving calls to HiL components (i.e., SUT and test instruments)
Setup	A test operation that initializes a test
Main	A test operation that performs the main function of a test
Teardown	A test operation that cleans up a test
Oracle	A mechanism to determine whether a test passes or fails with a confidence level (e.g., pass with 100% confidence or fail with 40% confidence)
TestResult	Actual test outputs
Measurement-Formula	A formula to measure a state value of test instruments, SUT, or test environments

Mapping (stereotype, specification): (HiLTestCase, testcase specification), (TestCaseProperty, property declaration), (TestCaseOperation, operation specification), (Setup, operation tagged as @Setup), (Main, operation tagged as @Main), (Teardown, operation tagged as @Teardown), (Oracle, operation tagged as @Oracle), (TestResult, TestResult type), (MeasurementFormula, measurementformula specification).

sic functions of a satellite which amplifies the power level of a received signal, i.e., the ul parameter, to make the signal strong enough for transferring it to a station on the ground. This physical behavior of signal amplification by a satellite is typically modeled by using the Saleh model (Saleh, 1981) as specified in Figure 7. The Saleh model is a non-linear formula to capture the behavior of an amplifier.

Test behavior. The TestBehavior package in Figure 4 contains the HiL test case specification concepts. Table 3 describes these concepts. Below, we describe how the TestBehavior modeling concepts capture test cases and test oracles.

Test cases. A test case is defined by HiLTest-Case and includes TestCaseProperty, TestCase-Operation, and Oracle. TestCaseProperty captures test data such as input and output variables and HiL components used by a test case. Each test case has one Setup, one Main, and one Teardown operation tagged by @Setup, @Main, and @Teardown annotations, respectively. The Setup operation of a test case contains a sequence of statements initializing the parameters and the HiL components used by the test case. The Main operation of a test case is executed after Setup and implements the test scenario by manipulating HiL components used by the test case. The Teardown operation of a test case is executed last and cleans up the HiL components used by the test case.

```
testcase TransferCurve {                                    HITECS
private expTF:Real;
private meaTF:Real;
private frequency:Real;
private sat:Satellite;
private synth:Synthesizer;
private acu:ACU;
private sa:SpectrumAnalyzer;

@Setup
public setup() {
  /* ... */
  assert acu.satLongitude == sat.longitude;
  assert acu.satLatitude == sat.latitude;
}

@Main
public measure() {
  /* ... */
  attenuation: Uniform(0,10) = Unknown; // Real type
  /* ... */
  sl:Real = spreadingLoss(distance);
  /* ... */
  //@SimTime("synth.time.record", "uniform")
  synth.generateSignal();
  assert sa.PowerLevel()<sat.powerThreshold;
  /* ... */
}

@Teardown
public teardown() {
  /* ... */
  assert synth.RFMode() == Synthesizer::OFF;
}

@Oracle
public testOracle() : TestResult {
  /* ... */
  if (meaTF == expTF) {
    return new TestResult(PASS);
  } else {
    diff = abs(meaTF - expTF);
    return new TestResult(diff/(1+diff), FAIL);
  }
}
}
```

Figure 8: (Simplified) HITECS specification for the transfer curve test case.

Figure 8 shows an example of a HITECS specification for the transfer curve test case of a satellite. The test case aims to test a transponder function of the satellite under test. We omit satellite-specific details for testing transponder behaviors, as they are not pertinent to this article; we instead refer the interested reader to the relevant literature from the satellite communications domain (Elbert, 2008; Saleh, 1981). The expTF and meaTF variables specify the expected and the actual outputs of the test case, respectively; frequency is the test input; and

```
1  measurementformula                                    HITECS
2  spreadingLoss(in distance:Real) : Real {
3    area:Real = 4*PI*pow(distance,2);
4    return log(area);
5  }
```

Figure 9: HITECS specification of the spreading loss computation.

`sat`, `synth`, `acu`, and `sa` are the required HiL components. The `setup()` operation of the test case moves an antenna in a ground station to point to the satellite under test; the `measure()` operation performs the signal measurement procedure to assess the transfer curve function of the satellite under test; and the `teardown()` operation turns off the HiL component (e.g., synthesizer) used by the test case.

The `MeasurementFormula` concept is defined to specify formulas required to measure states of the test instruments, SUT, or test environments while executing test cases. For example, the `measure()` operation in Figure 8 invokes the `spreadingLoss()` measurement formula specified in Figure 9. Spreading loss represents a decrease in the intensity of a signal which is computed based on the distance between a station on the ground and the satellite under test (see `distance` in line 3 in Figure 9). The spreading loss is one of many (environmental) factors required to accurately measure a level of signal power transmitted (resp. received) to (resp. from) the satellite under test.

Test oracles. A test oracle determines whether a test case execution passes or fails. In HITECS, `TestResult` contains a verdict from UTP and a confidence level. A confidence level is an application-specific notion capturing the degree of confidence in test verdicts. Provided with a confidence level, the engineers will be better positioned to decide which failures they would like to inspect first. A simple way to define the confidence level for numeric values is as the deviation between the actual and expected test outputs. Specifically, `testOracle()` (simplified for exposition) in Figure 8 determines the verdict by comparing the measured translation frequency (`meaTF`) and the expected translation frequency (`expTF`). When the two values are equal, a `PASS` verdict is returned by the test oracle. Otherwise, a `FAIL` verdict is returned along with a confidence level capturing the normalized deviation value between `meaTF` and `expTF`.

Table 4: The `Uncertainty` package stereotypes.

Stereotype	Description
`Unknown`	A literal constant to represent uncertainty
`Probability-Distribution`	A probabilistic distribution of values (from UUP)

Mapping (stereotype, specification): (`Unknown`, `Unknown` literal), (`ProbabilityDistribution`, `Uniform(min, max)` | `Triangular(min, mode, max)` | `Poisson(mean)`).

Uncertainty. The `Uncertainty` package in Figure 4 contains the modeling concepts for representing uncertainty in CPS HiL testing. Table 4 describes the concepts in this package. HITECS introduces a special `Unknown` literal to represent a value that can be determined only at the time of HiL testing and is unknown at the time of test specification. HITECS further allows test engineers to associate a probability distribution to an unknown value (see `ProbabilityDistribution` in Figure 4). Test engineers typically use `Unknown` for values that depend on uncertain environmental factors that are not a-priori-known such as temperature. For instance, line 20 in Figure 8 shows an example of using `Unknown`. The statement declares the `attenuation` variable of a uniform-distribution data type from UUP (see Figure 4) and initializes the variable to an unknown value within the interval of [0, 10], indicating that attenuation may get any value in [0, 10] with an equal probability. We note that the intention of `Unknown` in the statement (line 20) is to explicitly indicate that the value of the `attenuation` variable is unknown. Test engineers can omit `Unknown` in such variable declarations if they favor excluding an auxiliary keyword, i.e., Unknown, as doing so does not change the semantics. The `attenuation` variable could be specified using other distributions based on an engineer's approximative knowledge as follows: `Triangular(min,mode,max)` or `Poisson(mean)` (see the `Uncertainty` package in Figure 4). Attenuation represents the reduction of the amplitude of signals before they reach a satellite. Knowing the attenuation coefficient is necessary for calculating an appropriate level of signal power. The exact value of the attenuation nevertheless depends on environmental factors.

For the model checking and simulation analysis tasks (discussed in Section 5), the `Unknown` literals are, respectively, replaced with symbolic variables and random-number generators. Specifically, for model checking, additional constraints are added

Table 5: The `TestAnalysis` package stereotypes.

Stereotype	Description
`Assertion`	A predicate used to verify a test case
`Annotation`	An annotation attached to a statement and used by the HITECS simulator (e.g., to estimate the execution time of a test case)
`Annotation-Semantics`	An (operational) semantics of a simulation annotation

Mapping (stereotype, specification): (`Assertion`, `assert` statement), (`Annotation`, `//@identifer(arguments)` annotation), (`AnnotationSemantics`, `annotationsemantics` specification).

Table 6: Well-behavedness requirements for HiL testing.

R#	Description of well-behavedness requirement
R1	A HiL component should be correctly configured during its initialization
R2	A HiL component should be in a state where they can properly process the data that it receives from a test
R3	A HiL component should be cleaned up after finishing a test
R4	Inputs of a HiL component operation should be within valid ranges
R5	Outputs of a HiL component operation should be within valid ranges

along with symbolic variables to exclude out-of-range values when [min,max] ranges are explicitly specified, e.g., [min,max] for a uniform distribution. For example, the `attenuation` variable (line 20 in Figure 8) is replaced with a symbolic variable, e.g., SV_u, and a feasible condition of the variable is constrained by $0 \leq SV_u \leq 10$ as the `attenuation` range is explicitly specified by `Uniform(0,10)`. Note that when a Poisson distribution is associated with the `Unknown` literal, the literal is replaced with a symbolic variable, e.g., SV_p, and a feasible condition of the variable is constrained by $0 \leq SV_p$ as the minimum of a Poisson distribution is always 0 whereas its maximum value is not defined, regardless of the mean parameter of a Poisson distribution (see Figure 4). For simulation, the random-number generators, replacing `Unknown` literals, yield random numbers based on their corresponding distributions, e.g., uniform, triangular, or Poisson. For instance, the `attenuation` variable is replaced with a random number generator that returns any value in [0,10] with an equal probability.

Test analysis. The `TestAnalysis` package in Figure 4 contains the modeling concepts used for model checking, uncertaity resolution, and simulation (Section 5). Table 5 describes the concepts in this package. Among these concepts, `Assertion` and `Annotation` appear inside a test case specification, whereas `AnnotationSemantics` needs to be provided as a separate routine. The `TransferCurve` test specification in Figure 8 exemplifies `Assertion` and `Annotation`. As for `AnnotationSemantics`, an example is provided in Figure 10 (discussed later). Below, we elaborate the `TestAnalysis` package.

Assertions. HITECS defines the `Assertion` stereotype to specify the well-behavedness requirements of HiL test cases (see Section 3). In Table 6, we use the HITECS terminology to restate the well-behavedness requirements originally described in Section 3. Assertions capturing these requirements can be added to any of the test case operations (`Setup`, `Main`, and `Teardown`). For instance, the assertions on lines 13-14 in Figure 8 are related to R1 in Table 6 and specify that an antenna must point to the satellite under test after executing `setup()` in Figure 8. The assertion on line 26 in Figure 8 is related to R5 in Table 6 and specifies that the power of the signals sent to a satellite must be less than a threshold to avoid any damage to the satellite. Finally, the assertion on line 33 in Figure 8 is related to R3 in Table 6 and describes that the synthesizer must be turned off after the execution of the `teardown()` operation.

In general, one difficulty of applying verification techniques (e.g., model checking) is that the formal properties (e.g., assertions) are not available, and engineers may not know how to produce them. In the context of HITECS, engineers should transform the well-behavedness requirements in Table 6 into formal assertions defined based on HiL components' operations and properties, and HiL test case properties. These assertions should then be inserted into proper locations in HiL test case operations. To support engineers in developing assertions, in Section 5.1, we provide guidelines on how to systematically write assertions based on the well-behavedness requirements in Table 6 for HITECS test specifications.

Simulation annotations. HITECS simulation annotations aim to specify information about the cost and performance of test case statements in a way that the information can be interpreted by our simulation engine (see Section 5.3). In particular, in our case study, we use HITECS simulation annotations to specify the execution time of calls to HiL component operations. Our annotations are nevertheless flexible and can be used for other purposes too. The syntax of HITECS simulation annotations is rep-

```
1  annotationsemantics                                    HITECS
2  SimTime(in record:String, in type:String):Real {
3    t: Real = 0;
4    /*@inline('Java')
5    //... omitted
6    //list: contains time values in the record
7    if (type.equals("uniform")) {
8      Random r = new Random();
9      int size = list.size();
10     t = list.get(r.nextInt(size));
11   } else {
12     //t is determined by record and type
13     //e.g., triangular distribution
14   }
15   */
16   return t;
17 }
```

Figure 10: HITECS specification of `@SimTime` semantics.

```
1  scheduler ScheduleInOrbitTest() {                      HITECS
2    suite:TestSuite = new InOrbitSatTest();
3    for (tc in suite) { //tc: test case
4      tc.run();
5    }
6  }
```

Figure 11: HITECS specification of a test scheduler.

Table 7: The `TestSchedule` package stereotypes.

Stereotype	Description
`HiLTestSuite`	An ordered list of test cases
`HiLTestSchedule`	A procedure that defines the execution order of a test suite

Mapping (stereotype, specification): (`HiLTestSuite`, `TestSuite` container), (`HiLTestSchedule`, `scheduler` specification).

resented as a form of `//@identifier(arguments)` where `identifier` and `arguments` denote the name and an optional list of arguments for the annotation. Each annotation provides information about the statement that immediately follows it. We refer to the statement following an annotation as the *annotated statement*. For example, `@SimTime("synth.time.record","uniform")` on line 24 in Figure 8 is an annotation providing information about the execution times of its next statement, i.e., line 25. This annotation has `SimTime` and `("synth.time.record","uniform")` as its `identifier` and `arguments`, respectively.

To make the annotations interpretable by our simulator, we require that test engineers should provide the (operational) semantics of each annotation using Alf or Java routines. The routine specifying the semantics of an annotation `//@identifier(arguments)` must be named `identifier(arguments)`. For example, Figure 10 illustrates the semantic routine related to the `@SimTime` annotation in Figure 8. This routine is specified in Java since `@SimTime`'s semantics relies on Java libraries for statistical analysis. In this routine, the block between lines 4–15 is nested by the Alf statement `/*@inline('Java') ... */`, indicating that the block is specified in Java. `@SimTime` has two arguments: *record* which is a list of execution time values of the annotated statement, and *type* which defines how a distribution can be built based on the values in *record*. According to the `@SimTime` routine in Figure 10 (lines 7–10), the `@SimTime` annotation in Figure 8 indicates that the execution time of the statement `synth.generateSignal()` can take, with an equal probability, any value from `synth.time.record`. In Section 5.3, we will discuss how the annotation semantics are used by our simulator. Note that, as we discuss in Section 5.3, HITECS annotations are side-effect-free. This is in contrast to Alf annotations in general, which are not necessarily side-effect-free and can modify the behavior of the annotated statements.

TestSchedule. `TestSchedule` enables engineers to execute test cases in a particular order. Table 7 describes the stereotypes in `TestSchedule` of HITECS. For instance, `ScheduleInOrbitTest` in Figure 11 runs the test cases in the `InOrbitSatTest` test suite based on the order specified in `suite`. Line 4 in Figure 11 runs each test case `tc` in `suite` by sequentially executing the `@Setup`, `@Main`, and `@Teardown` operations of `tc`. Note that test oracle operations are optional in HITECS (see Figure 4); hence, they may or may not be invoked by test schedules.

5. Specification analysis

In this section, we describe how HITECS enables model checking, uncertainty resolution, and simulation of HiL test cases. Figure 12 shows the analysis component of HITECS. Specifically, HITECS model checking contains the following three steps: "convert HITECS assertions", "translate HITECS into Java" and "run JavaPathFinder"; HITECS uncertainty resolution contains the following four steps: "translate HITECS into Java", "simulate for sampled values", "learn conditions using decision trees" and "run JavaPathFinder with conditions"; and HITECS simulation contains the following four steps: "convert HITECS Annotations", "translate HITECS into

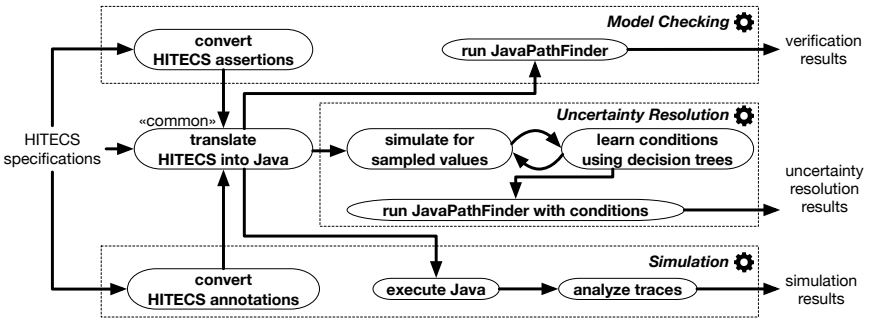


Figure 12: Overview of the analysis component in Figure 3 performing model checking, uncertainty resolution, and simulation.

Java", "execute Java", and "analyze traces". The three analysis tasks translate HITECS specifications into Java (see the common step "translate HITECS into Java" in Figure 12). As discussed in Section 4, HITECS adopts the formal, operational semantics of Alf. The translation of HITECS into Java relies on the Alf semantics and prior translations of Alf into object-oriented programming languages such as Java and C++. We omit the technical details of the translation, which are not part of our contributions, and refer the interested reader to existing work (Buchmann and Rimer, 2016; Ciccozzi, 2016). Below, we explain the steps of model checking, uncertainty resolution, and simulation in HITECS.

5.1. Model checking

The goal of HITECS model checking is to show the well-behavedness of HITECS specifications based on the requirements in Table 6. To do so, the requirements must first be specified in terms of assertions. The effectiveness of model checking highly depends on the precision and quality of the underlying assertions. However, developing assertions requires a lot of manual effort and poses a challenge to test engineers who are typically experts in some CPS application domain (e.g., automotive or space engineering), but not necessarily in software engineering. To address this difficulty, we provide guidelines to help test engineers specify precise well-behavedness assertions for HiL test cases and place these assertions in appropriate locations within HITECS specifications.

Table 8 presents our guidelines for specifying the assertions induced by the requirements in Table 6. The guidelines specify the content of assertions and their expected locations in HITECS specifications. For example, the guideline in the first row of Table 8 (which prescribes assertions for checking whether the `HiLComponent` attributes are set correctly after the test case setup operations) aims to capture requirement R1 in Table 6. The two assertions on lines 13 and 14 in Figure 8 are written based on this guideline. Similarly, the assertion on line 26 in Figure 8 is written based on the guideline on the second row of Table 8. Finally, the assertion on line 33 in Figure 8 follows the guideline on the last row of Table 8.

Table 8: Guidelines prescribing assertions to be inserted into HITECS specifications to verify the requirements in Table 6.

R#	Assertion guideline related to requirement R#
R1	At the end of the `Setup` operation of a `HiLTestCase`, an assertion may check if each `CompProperty` of each `HiLComponent` is properly initialized
R2 R5	After each `CompOperation` invocation by a `HiLTestCase`, an assertion may check if the output of `CompOperation` is within its valid ranges; further, an assertion may check if each `CompProperty` of each `HiLComponent` is set correctly
R2 R4	Before each `CompOperation` invocation by a `HiLTestCase`, assertions may check if the input parameters of `CompOperation` are within their valid ranges; further, an assertion may check if `HiLComponent` is in state where `CompOperation` can be invoked
R3	At the end of the `Teardown` operation of a `HiLTestCase`, an assertion may check if each `CompProperty` of each `HiLComponent` is properly cleaned up

Regarding inapplicable HITECS specifications (described in Section 3), HITECS model checking replaces complex equations that cannot be analyzed with (unconstrained) symbolic variables. As this replacement provides conservative abstractions for those complex equations, HITECS model checking results are conservative as well. Specifically, HITECS model checking returns a conclusive result for an assertion verification only when the assertion is satisfied or violated for *all* possible executions

```
1  testcase TransferCurve {                                    HITECS
2  /* ... */
3
4  @Setup
5  public setup() {
6    /* ... */
7    assert acu.satLongitude == sat.longitude;
8    assert !(acu.satLongitude == sat.longitude);
9    assert acu.satLatitude == sat.latitude;
10   assert !(acu.satLatitude == sat.latitude);
11 }
12
13 @Main
14 public measure() {
15   /* ... */
16   assert sa.PowerLevel()<sat.powerThreshold;
17   assert !(sa.PowerLevel()<sat.powerThreshold);
18   /* ... */
19 }
20
21 @Teardown
22 public teardown() {
23   /* ... */
24   assert synth.RFMode() == Synthesizer::OFF;
25   assert !(synth.RFMode() == Synthesizer::OFF);
26 }
27 }
```

Figure 13: Negated assertions for verifying the HITECS specification of Figure 8.

of the corresponding HITECS specification. Due to uncertainty in CPS HiL testing, HITECS model checking also returns an inconclusive result for an assertion when *some* executions of the corresponding HITECS specification violate the assertion. To determine inconclusiveness of verification results, HITECS model checking verifies an assertion and its negated assertion together. HITECS model checking returns inconclusive for an assertion when model checking concludes that both the assertion and its negation hold for *some* executions.

Having defined our guidelines for assertion specification, we now describe the steps of HITECS model checking in Figure 12. To determine if a given assertion can be conclusively verified, the "convert HITECS assertions" step first inserts negated assertions below the original assertions and converts both the original and negated HITECS assertions into Java assertions. Each assertion then can be conclusively verified if either the assertion or its negation is proven by the model checker. The former means that the assertion is conclusively correct and the latter means the assertion is conclusively false. If neither the assertion nor its negation can be verified by the model checker then we can conclude that the assertion is inconclusive (i.e., its correctness or lack there-of is unknown). For instance, Figure 13 shows a modified HITECS specification after which the "convert HITECS assertions" step added the negated assertions (lines 8, 10, 17, and 25 in Figure 13) of the respective original assertions in the HITECS specification of Figure 8. We note that such intermediate modified HITECS specifications are maintained by the HITECS model checking component to keep track of the original and negated assertions. Similarly, the "translate HITECS into Java" step produces the Java translations of HITECS specifications. In this step, the `Unknown` literals in the HITECS specifications are replaced with symbolic variables, as explained in Section 4.2.

The "run JavaPathFinder" step applies JavaPathFinder (Visser et al., 2003) (using Z3 (de Moura and Bjørner, 2008)) – a well-known and widely-used model checking tool – to the generated Java translations. For each assertion, applying JavaPathFinder leads to one of the following situations: (1) JavaPathFinder terminates and computes inputs violating either the assertion or its negation, (2) JavaPathFinder terminates and reports that either the assertion or its negation satisfies all the inputs, (3) JavaPathFinder terminates and reports some failure messages for handling complex equations, or (4) JavaPathFinder fails to terminate within the time allotted. For case (1), we say the assertion is *inconclusive*. For case (2), we say the assertion is *conclusive*. For case (3), JavaPathFinder is inapplicable of handling that assertion. Given the failure messages containing the information of unhandled complex equations, the "run JavaPathFinder" step replaces the complex equations with symbolic variables, and then applies JavaPathFinder again. For case (4), HITECS model checking is not able to verify the Java translation.

We chose to translate HITECS specifications into Java since Alf constructs can be easily mapped to Java. Alternatively, we could have translated HITECS into other programming languages (e.g., C++) and used other model checkers (e.g., CBMC (Clarke et al., 2004)).

5.2. Uncertainty resolution

The HITECS uncertainty resolution method aims to identify conditions on unknown parameters of a HITECS test case specification, containing inconclusive assertions, under which the specification is well-behaved. To do so, the HITECS uncertainty resolution method combines the following three analysis tools: (1) a simulator to check assertions in the HITECS specification for sampled parameter values,

```
1  Algorithm simulate for sampled values
2  Input tc: HITECS test case specification
3  Input ns: Number of samples
4  Input V⃗: Vector of [min,max] ranges for unknown vars
5  Output D⃗: Vector of (values, true/false) samples
6
7  D⃗ ← () // empty vector
8  for ns times do
9  |  v⃗ ← sample(V⃗) // vector of values
10 |  trace ← run(tc, v⃗) // execution trace
11 |  b ← check(trace) // true if all assertions hold
12 |  s⃗ ← ⟨v⃗, b⟩ // label: b (true or false)
13 |  D⃗ ← add(D⃗, s⃗)
14 return D⃗
```

Figure 14: A simulation algorithm for creating labeled samples consisting: values of unknown parameters in a HITECS specification and an assertion result of the specification determined by the parameter values.

```
1  Algorithm learn conditions using decision trees
2  Input tc: HITECS test case specification
3  Input ns: Number of samples
4  Input V⃗: Vector of [min,max] ranges for unknown vars
5  Input nd: Number of decision trees to build // budget
6  Output T: Set of vectors of [min,max] value ranges
7
8  T ← {} // empty set
9  R ← {V⃗}
10 cnt ← 0
11 while cnt < nd and R ≠ {} do
12 |  L⃗ ← findLargestRegion(R) // value ranges
13 |  R ← R \ {L⃗}
14 |  D⃗ ← simulate tc with ns and L⃗ // see Figure 14
15 |  if for all ⟨v⃗, b⟩ in D⃗: b = true
16 |  |  T ← T ∪ {L⃗}
17 |  else
18 |  |  t ← createTree(D⃗) // learning decision tree
19 |  |  N ← narrowRanges(t, L⃗) // incl. true samples
20 |  |  R ← R ∪ N
21 |  |  cnt ← cnt + 1
22 return T
```

Figure 15: A classification algorithm for learning value ranges, i.e., conditions, of unknown parameters in a HITECS specification under which the specification is likely well-behaved.

(2) an ML-based decision tree learner to infer conditions, i.e., value ranges, on unknown parameters from the sampled values and simulated assertion results under which the HITECS specification is likely well-behaved, and (3) a model checker to find a subset of the conditions learned from decision trees under which the specification is conclusively well-behaved. As shown in Figure 12, the HITECS uncertainty resolution method analyzes a Java translation of the HITECS specification produced by the "translate HITECS into Java" step. The remaining steps of the HITECS uncertainty resolution method are the following steps: "simulate for sampled values", "learn conditions using decision trees", and "run JavaPathFinder with conditions".

The "simulate for sampled values" step in Figure 12 executes a Java translation of a HITECS specification *tc* to create a labeled dataset containing tuples $\langle \vec{v}, b \rangle$ where $\vec{v}$ is a value assignment to uncertain parameters of *tc*, and b is a binary label indicating whether or not *tc* executed with $\vec{v}$ satisfies its assertions. Figure 14 describes an algorithm executed at the "simulate for sampled values" step. Given the vector $\vec{V}$ of the initial value ranges for n number of unknown parameters in a HITECS test case specification *tc*, let V_i be a range of ith unknown parameter in *tc* where $i = 1, 2, ..., n$. We denote by $min(V_i)$ and $max(V_i)$, respectively, the minimum and maximum values in the V_i range. When the minimum or maximum of V_i is not defined, $min(V_i)$ or $max(V_i)$ returns, respectively, negative or positive infinity. Note that the implementation of the HITECS tool – a Java program – is not capable of supporting true infinity; hence, predefined MIN_VALUE and MAX_VALUE constants in Java are used accordingly.

The algorithm in Figure 14 first creates value assignments $\vec{v}$ for $\vec{V}$ such that $min(V_i) \leq v_i \leq max(V_i)$ for all value v_i in $\vec{v}$ and V_i in $\vec{V}$ (line 9). We use an adaptive random search technique (Luke, 2013) to sample values within ranges. The adaptive random search extends the naive random search by maximizing the Euclidean distance between the sampled points. The algorithm then executes the java translation of *tc* with the $\vec{v}$ vector of sampled values (line 10) and determines whether or not all the assertions of *tc* hold under the sampled values (line 11). An assertion result b is *true* only if all the assertions of *tc* are satisfied; otherwise, the sampled values are labeled with *false*. We refer to a tuple of $\langle \vec{v}, true \rangle$ as a true-labeled sample and $\langle \vec{v}, false \rangle$ as a false-labeled sample. The algorithm records the sampled values and the assertion result (lines 12–13). Last, the algorithm repeats the above process to collect *ns* number of labeled samples in the $\vec{D}$ dataset (lines 8–13).

Given a dataset $\vec{D}$ obtained by executing the algorithm in Figure 14, the "learn conditions using decision trees" step in Figure 12 builds a set of decision trees and identifies conditions on unknown parameters under which the HITECS specification *tc* likely satisfies all its assertions. Figure 15 shows an algorithm that describes the (iterative) interactions between the "learn conditions using decision

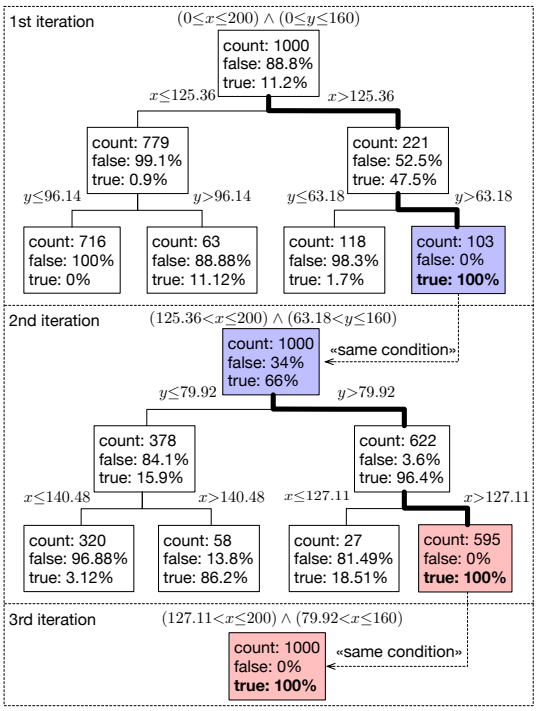


Figure 16: Example decision trees generated by the learning algorithm described in Figure 15.

trees" and "simulate for sampled values" steps. The algorithm first simulates the *tc* specification with a vector $\vec{L}$ of parameter ranges to obtain a dataset $\vec{D}$ containing *ns* number of labeled samples (line 14). At the first iteration of the algorithm, $\vec{L}$ is equal to the input vector $\vec{V}$ of the initial value ranges of unknown parameters of the *tc* specification. If the obtained dataset $\vec{D}$ contains only true-labeled samples, the algorithm record $\vec{L}$ to be returned since the *tc* specification holds all the assertions under $\vec{L}$ (lines 15–16). In the other case, the algorithm builds a decision tree based on the $\vec{D}$ dataset (line 18) and then uses the tree to restrict the value regions, i.e., strengthen its condition, which would likely satisfy all the assertions in the next iteration (lines 19–21). The set R of candidate value regions obtained at lines 19–20 are decided by selecting leaves in the t tree (line 18). Engineers set the rules to choose such leaves to be analyzed in the next iteration, based on a set minimum percentage of true-labeled samples (e.g., 100%, 95%) and then by prioritizing candidate regions based on their size, i.e., analyzing the ones with the largest region size first (line 12). As we discuss below, Figure 16 illustrates the manipulation of decision trees by the algorithm in Figure 15.

The algorithm in Figure 15 relies on decision tree learning which is a supervised learning technique, using a labeled dataset (Witten et al., 2011). In the algorithm, the labeled dataset $\vec{D}$ is created by executing the simulation algorithm presented in Figure 14 (see lines 14 and 18 in Figure 15). The learning algorithm in Figure 15 uses an ML-based binary decision tree learner to classify the two (true or false) labels in the $\vec{D}$ dataset. A decision tree is composed of decision nodes, leaves, and edges. A decision node is linked via edges to its child nodes. An edge represents a condition on a parameter to be evaluated which determines a decision path to descend from a parent node to an appropriate child node. Note that a decision node has only disjoint (deterministic) conditions associated with its edges. A leaf node represents a true or false assertion result determined by a conjunction of edge predicates from the root to the leaf.

For example, Figure 16 shows three decision trees generated by executing a decision tree learner (line 18) in Figure 15 for an artificial HITECS test case specification *tc* with two unknown parameters x and y. We set initial value ranges of the two parameters x and y to [0,200] and [0,160], respectively. As shown in the root node of the "1st iteration" part in Figure 16, the tree is constructed by the dataset containing 1000 samples – 88.8% of the samples violate some assertions in the HITECS specification, whereas 11.2% of them satisfy all the assertions in the specification. Given the decision tree, the algorithm identifies a set of conditions under which they likely satisfy all the assertions in the specification (line 19 in Figure 15). At the "1st iteration" part in Figure 16, the algorithm identifies that the specification likely holds its assertions under the condition $c_1 = (125.36 < x \leq 200) \wedge (63.18 < y \leq 160)$ as all the 103 out of 1000 samples satisfying c_1 are true-labeled. In this example, the algorithm chose a condition yielding 100% true-labeled samples. However, an engineer can easily relax this and allow some small percentages of false-labeled samples, e.g., false: 5% and true: 95%. At the "2nd iteration" part in Figure 16, the algorithm further restricts the condition c_1 as 34% of the newly created 1000 samples satisfying c_1 are false-labeled. At the "3rd iteration", the algorithm returns the condition $c_2 = (127.11 < x \leq 200) \wedge (79.92 < y \leq 160)$ since all the new 1000 samples are true. Note that in this example, we happen to have only one condition yielding 100% true-labeled samples. Otherwise, if

there are multiple such conditions, the algorithm attempts to strengthen all of them by constructing and analyzing the corresponding decision trees.

Note that the algorithm in Figure 15 analyzes and restricts all the candidate value regions, i.e., conditions, in priority order of their region sizes, i.e., analyzing the largest one first (line 12), unless it runs out of its search budget *nd*. Let $\vec{V}$ be the input space containing value ranges of n number of unknown parameters in a HITECS specification. Let R be the set of candidate regions obtained at lines 9, 13, and 20 in Figure 15. Let $\vec{W}_j$ be an element in R, and W_i^j be an element range in $\vec{W}_j$ such that V_i in $\vec{V}$ and W_i^j in $\vec{W}_j$ are different ranges for the same parameter. We then define the size of $\vec{W}_j$ in R as follows:

$$size(\vec{W}_j, \vec{V}) = \prod_{i=1}^{n} \frac{max(W_i^j) - min(W_i^j)}{max(V_i) - min(V_i)}$$

We note that unknown parameters in a HITECS specification may have different scales. Hence, in the $size(\vec{W}_j, \vec{V})$ function, for all W_i^j in $\vec{W}_j$ and V_i in $\vec{V}$, the range distance of $max(W_i^j) - min(W_i^j)$ is divided by the original input range distance of $max(V_i) - min(V_i)$ to normalize the differences.

Last, the "run JavaPathFinder with conditions" step in Figure 12 uses model checking to ensure a given HITECS specification constrained by the conditions synthesized by the algorithm in Figure 15 conclusively satisfies all its assertions. Specifically, for each HITECS specification *tc* and each condition *c* computed by the algorithm in Figure 15, we check whether *tc* satisfies all its assertions after its uncertain parameters being restricted based on *c*. The HITECS uncertainty resolution approach reports conditions *c* only if they lead to conclusive satisfaction of the assertions as determined by an exhaustive model checker. Note that if a HITECS specification is inapplicable, the "run JavaPathFinder with conditions" step replaces complex equations with symbolic variables in a conservative manner as the HITECS model checking method does.

5.3. Simulation

In Section 4.2, we described the general syntax of HITECS simulation annotations and how engineers can specify their semantic routines. In this section, we describe the steps of the HITECS simulation in Figure 12. We further show how the simulation annotations can be used for estimating HITECS specification execution times.

```
1  Function Sum
2  Input traces: execution traces
3  Input id: annotation's identifier of interest
4  Output t⃗: vector of values
5
6  t⃗ ← () //empty vector
7  for each trace in traces do
8  |  tmp ← 0;
9  |  call_list ← grep id in trace
10 |  //call_list: call stmts to the id semantic routine
11 |  for each call_stmt in call_list
12 |  |  ret ← execute call_stmt
13 |  |  tmp ← tmp + ret
14 |  t⃗ ← add(t⃗, tmp)
```

Figure 17: A pre-defined aggregator function of the "analyze traces" step in Figure 12.

The "translate HITECS into Java" step produces Java translations of HITECS specifications, excluding their simulation annotations. The simulation annotations are handled separately by the "convert HITECS annotations" step, which creates a log statement corresponding to each annotation and inserts the statement into the Java translations. Every time we execute a Java translation of a HITECS specification (using the "execute Java" step), each log statement corresponding to the `//@identifier(arguments)` annotation inserts into the output trace an invocation to the `identifier(arguments)` semantic routine.

The last step, "analyze traces", computes the simulation results based on the output traces generated by the "execute Java" step. This last step scans all the traces and executes the semantic routine for each annotation whenever it encounters a call to that routine in the traces. The outputs obtained from individual semantic routines should be aggregated to generate simulation results. To do so, the "analyze traces" step provides some pre-defined functions aggregating these outputs. Specifically, for each annotation in the input HITECS specifications, engineers need to either select an aggregator function from the pre-defined ones or define their own aggregator function. Figure 17 shows (in pseudo-code form) an example aggregator function, pre-defined by the "analyze traces" step. This function computes a vector $\vec{t}$ such that every element of $\vec{t}$ is related to one trace in `traces` and is the sum of the outputs of the semantic routine of the `id` annotation appearing on that trace.

For example, Figure 18 shows how the HITECS simulation is used to compute the execution time estimations for HiL test cases. Recall that for this purpose engineers need to annotate statements using

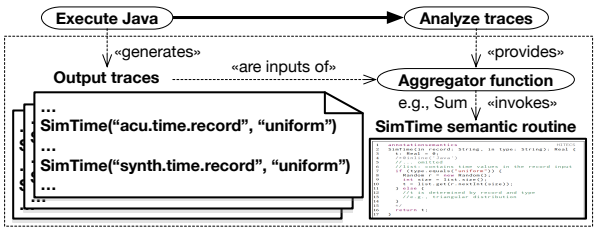


Figure 18: Estimating the execution times of HiL test cases using the HITECS simulation in Figure 12.

`@SimTime` and provide a `SimTime` semantic routine (e.g., Figure 10). We use the `Sum` aggregator function in Figure 17 to combine the execution times of individual statements. The HITECS simulation first simulates the HiL test case under analysis a number of times based on different inputs and randomly-generated numbers conforming to associated probabilistic distributions for the `Unknown` literals. This accounts for the randomness of the `Unknown` literal and generates a set of traces corresponding to different inputs. As shown in Figure 18, the output traces, which contain calls to the `SimTime` routine, are passed to the `Sum` function. For each trace, the `Sum` function computes the sum of the execution times generated by calls to the `SimTime` routine and stores the sum in the vector $\vec{t}$. At the end, this vector represents a distribution of the execution time values of the HiL test case under analysis obtained based on several runs of the test case.

Note that as mentioned in Section 4.2, our annotations are side-effect-free because our simulator executes the annotation routines after executing the HITECS specifications and only to interpret their output traces.

6. Evaluation

This section describes our evaluation of the HITECS specification and analysis framework through an industrial case study from the satellite domain. Our (sanitized) case study data is available online (Shin et al., 2019).

6.1. Research questions (RQs)

RQ1 (assertion guidelines): *Are our guidelines for defining well-behavedness assertions useful?* HITECS model checking relies on the guidelines that we provide to assist test engineers with defining well-behavedness assertions (see Section 5.1). In RQ1, we investigate whether our guidelines lead to more effective and complete well-behavedness assertions for HiL test cases, compared to when these assertions are defined without systematic guidance.

RQ2 (model checking): *Can HITECS verify HiL test case assertions in practical time?* HITECS uses JavaPathFinder to verify the assertions in HiL test cases. Although JavaPathFinder has been successfully applied in some application domains (Visser et al., 2004; van der Merwe et al., 2012; Lindstrom et al., 2005), it has not been previously evaluated for CPS test cases. In RQ2, we investigate whether JavaPathFinder is able to verify well-behavedness assertions of industry HiL test cases in practical time (see the definition of conclusiveness in Section 5.1).

RQ3 (uncertainty resolution): *Can HITECS identify conditions on unknown parameters of HiL test cases under which HiL test cases are conclusively well-behaved?* The HITECS uncertainty resolution combines the following three compute-intensive techniques: simulation, learning decision trees, and model checking. In RQ3, we investigate whether the HITECS uncertainty resolution approach identifies, in practical time, conditions on unknown parameters of industry HiL test cases, under which they are conclusively well-behaved. We further examine the impact on the identification of such conditions by varying ML-based decision tree learners (i.e., J48 (Quinlan, 1993), SimpleCart (Breiman et al., 1984), and REPTree (Quinlan, 1987)) which have been applied in many studies (Abdessalem et al., 2018; Bettaieb et al., 2019; Safavian and Landgrebe, 1991).

RQ4 (simulation): *Can HITECS accurately estimate the execution times of HiL test cases via simulation?* The HITECS simulation generates in a randomized way a large number of HiL test case traces and analyzes them based on the `@SimTime` annotation semantics (see Section 5.3). To answer RQ3, we evaluate whether the randomized HITECS simulation is able to accurately estimate the execution times of industry HiL test cases.

6.2. Industrial study subjects

We have evaluated our approach by applying it to a real in-orbit-testing case study from the satellite domain. The case study context was described earlier in Section 2. Our evaluation is based on seven representative HiL test scenarios from SES Networks. Using textual documents describing in-orbit testing procedures, we created HITECS specifications for these seven scenarios. The resulting

HITECS test case specifications contain between 821 to 1123 statements each. In total, these test case specifications use 16 different HiL components. Each component has between zero to 25 attributes, between two to 27 operations, and between two to nine physical behaviors. Each test case specification has between eight to ten known parameters, between seven to eleven unknown parameters, and between one and two measurement formulas and interacts with between 13 to 15 components. The initial value ranges of unknown parameters are also provided by SES Networks.

The textual descriptions from SES Networks envisaged a number of well-behavedness checks for each of the test scenarios in our study. These checks were placed into the test scenarios based solely on the domain knowledge of the engineers, and without following a systematic process. We converted these checks into HITECS assertions and inserted them into our HITECS specifications. On average, we had 53.4 assertions per specification. As we will discuss in Section 6.3, to answer RQ1, we compare these assertions, which are rather ad-hoc and defined without systematic guidance, with the well-behavedness assertions that we derive based on our guidelines in Table 8.

In addition, for each of the seven test cases in our study, we obtained historical data files from real-world executions of the tests in previous in-orbit testing campaigns performed on satellites and HiL platforms similar to ours. Specifically, the data files were obtained based on components that were identical or near-identical to our case study components, the same satellite orbital characteristics, and the same ground station for communicating with the satellite. In general, such usable historical data is obtainable for many CPS, since these systems often share a lot of common components with previous systems. Furthermore, new CPS components typically come with detailed technical specifications and performance data from the manufacturers. In our case study, we extracted from the available historical data execution time values for the component operation calls as well as the whole HiL test cases. We use these values to answer RQ4.

6.3. Experiment design

To answer RQ1 and RQ2, we rely on mutation analysis (Jia and Harman, 2011) of test cases. Specifically, we created faulty test cases using an automated fault injection method. To do so, we designed a number of mutation operators to capture common faults in this domain. The operators were designed based on our discussions with domain experts as well as our analysis of the in-orbit test scenario documents. We implemented three mutation operators: (1) deleting an operation call, (2) modifying the return value of an operation and (3) modifying the input parameter value of an operation. For mutation operators (2) and (3), we negate the value if it is boolean, replace it with the next/previous value if it is from an enumeration, add a constant to it if it is numeric, and replace it with null if it is a string.

Our fault seeding program generated 781 *candidate* mutants based on our seven HITECS specifications. Each mutant contained one fault seeded by one mutation operator. Some of these mutants were not faulty as they were behaviorally equivalent to the unmutated HITECS specifications (i.e., equivalent mutants). For example, in our context, equivalent mutants were created because there were some duplicated component operation calls in the original test scenarios that carried over to our HITECS specifications. Removing the redundant statements does not introduce a fault. Following the procedure proposed by Yao et al. (2014), we identified the equivalent mutants by manually inspecting all the candidates. In our study, 172 out of the 781 generated mutants turned out to be equivalent. We used the remaining 609 non-equivalent mutants in our experiment.

To answer RQ1, we added to our seven HITECS specifications the assertions prescribed by our guidelines in Table 8. On average, per specification, we had 110.7 assertions prescribed by our guidelines. Recall from Section 6.2 that our HITECS specifications also include some assertions based on the ad-hoc checks in the textual test scenario descriptions. We put the ad-hoc assertions and the ones based on guidelines in separate copies so as to compare them. Note that the mutation operators do not change the assertions.

We consider two metrics to answer RQ1 and RQ2: (1) *mutation coverage* and (2) *execution time*. We say a HITECS mutant is killed if JavaPathFinder reports that the mutant violates at least one of its assertions. For each test case tc, we compute the *mutation coverage* $cov(tc)$ as the proportion of the number of killed mutants of tc over the total number of non-equivalent mutants of tc. For the second metric, we measure the execution time of each run of JavaPathFinder on each mutant. We note that inconclusive assertions are excluded in the experiments for RQ1 and RQ2. This is because inconclusive assertions cannot be used to detect faults

injected in mutants. We address the issues regarding inconclusive assertions by the HITECS uncertainty resolution method and evaluate the method by answering RQ3.

To answer RQ3, we apply the HITECS uncertainty resolution method to the seven HITECS specifications with the guideline-based assertions and consider two metrics: (1) *execution time* and (2) *region size*. For the first metric, we measure the execution time of the HITECS uncertainty resolution method. In addition, as inferred conditions depend on a decision tree learning techniques (e.g., J48, SimpleCart, or REPTree), we examine different hyper-parameter configurations of these techniques, which have been successfully applied in many applications (Abdessalem et al., 2018; Bettaieb et al., 2019; Safavian and Landgrebe, 1991). To compare the performance of those combinations, we measure the region size (see *size*() equation in Section 5.2) determined by the conditions on unknown parameters of a HiL test case returned by the uncertainty resolution method. The larger the region size, the more options to choose parameter values. Test engineers typically prefer to have a higher number of options regarding unknown parameter values, assuming all the options guarantee the well-behavedness of HiL testing.

We conducted initial experiments (1) to confirm the use of the adaptive random search instead of the naive random search for sampling values of unknown parameters and (2) to identify, among all hyper-parameters of ML-based decision tree learners, which ones have a large impact on the resulting region sizes. The initial experiments confirmed that the adaptive random search produces a larger region size than the naive random search. Regarding hyper-parameters, we found that tuning the hyper-parameters which control tree pruning and node splitting has a significant impact on the region sizes, while tuning the other parameters does not lead to notable differences. Therefore, we compare the three decision tree learners, i.e., J48, SimpleCart, and REPTree, by using optimal values of the tree pruning and node splitting parameters.

To answer RQ4, for each component operation call statement in our HITECS specifications, we inserted a `@SimTime` to annotate that statement. Recall from Section 4.2 that `@SimTime` has two parameters `record` and `type`. For the `record` parameter, we analyzed historical data files from past real-world in-orbit testing campaigns as described in Section 6.2 and obtained a vector of execution time values for each component operation call. We specified the `type` parameter as `uniform` (see the example in Figure 8).

We ran the experiments on a computer equipped with a 2.8 GHz Intel Core i7 CPU and 16 GB of memory.

Table 9: Mutation analysis results for the seven HITECS specifications.

tc	# non-equivalent mutants	# killed mutants		*cov*(*tc*)	
		ad-hoc	guideline	ad-hoc	guideline
tc1	90	56	90	0.62	1.00
tc2	63	51	63	0.81	1.00
tc3	57	49	57	0.86	1.00
tc4	99	57	99	0.58	1.00
tc5	97	56	97	0.58	1.00
tc6	91	57	91	0.63	1.00
tc7	112	56	112	0.50	1.00

6.4. Results

RQ1 (assertion guidelines). We applied JavaPathFinder to the 609 mutants containing ad-hoc assertions and to the 609 mutants containing assertions prescribed by our guidelines. Table 9 shows the mutation coverage values, *cov*, for ad-hoc and guideline-based assertions for each test case in our study. As shown in the table, the number of killed mutants containing ad-hoc assertions is less than the number of killed mutants containing assertions based on our guidelines. Specifically, for all the test cases, while all of the mutants with guideline-based assertions are killed by JavaPathFinder, only 50% to 86% of the mutants with ad-hoc assertions are killed by JavaPathFinder.

We note that the results are valid under our experiment design described in Section 6.3. Specifically, mutants created by the three mutation operators (see Section 6.3) are subject to be killed by the ad-hoc and guideline-based assertions in our experiments. To create realistic and representative fault-seeded mutants, we relied on our collaborating partner's inputs, i.e., discussions and real in-orbit test scenarios. Systematically creating effective mutants is still challenging (Andrews et al., 2005; Petrovic et al., 2018), but lies outside the scope of this article.

The answer to **RQ1** *is that* our guidelines help engineers develop more effective and complete well-behavedness assertions for HiL test cases compared to when engineers develop assertions without any systematic guidance.

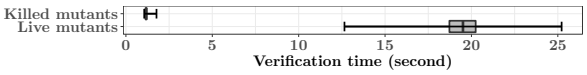


Figure 19: HITECS verification time for the live mutants and killed mutants. Box plot: Min-25%-50%-75%-Max.

Table 10: Number of guideline-based assertions for each HiL test case in our case study (see the definitions of conclusive and inconclusive assertions described in Section 5.1).

type	# assertions						
	tc1	*tc2*	*tc3*	*tc4*	*tc5*	*tc6*	*tc7*
conclusive	110	67	60	119	123	126	149
inconclusive	4	1	2	4	4	2	4

RQ2 (model checking). JavaPathFinder was able to verify all the mutants in our experiments by terminating in less than 25.2s, and either reporting assertion violations or concluding that no assertion is violated. Figure 19 shows the execution times of JavaPathFinder for the killed and live mutants separately. On average, it took JavaPathFinder 1s to show assertion violations for killed mutants, and 19s to conclude no assertion is violated for live mutants. Further, it took JavaPathFinder 1.35h and 0.20h to verify the mutants containing ad-hoc and guideline-based assertions, respectively. The mutants with guideline-based assertions required significantly less verification time compared to those with ad-hoc assertions because they included significantly more killed mutants.

We note that the practical efficiency of JavaPathFinder in our context is partly due to the simple structure of HiL test cases. In particular, HiL test cases are mainly sequential, typically contain few branches and their loops are often bounded with constant values. Otherwise, the performance of model checkers (such as JavaPathFinder) may diminish both in terms of execution time and memory usage when they are applied to concurrent code with unbounded loops and highly branching structures.

The answer to **RQ2** *is that*, for any one of the HiL test cases in our study, JavaPathFinder verified all the well-behavedness assertions in less than 25.2s.

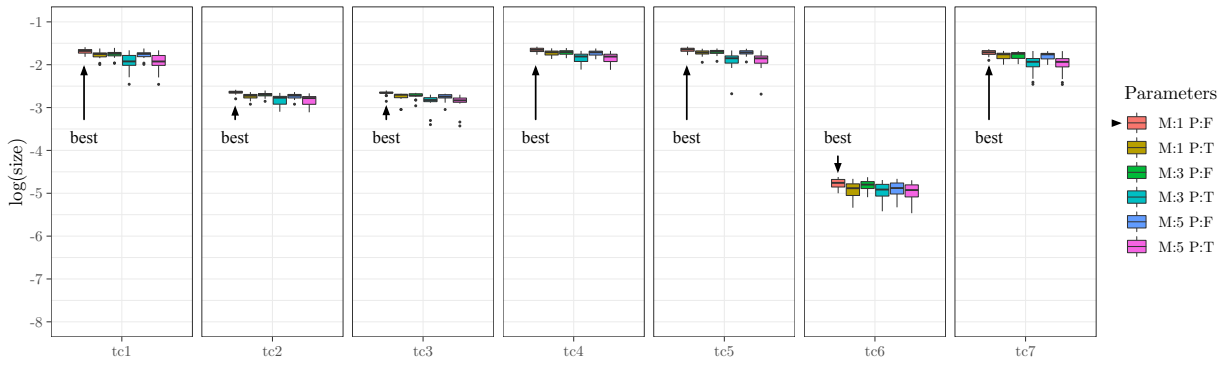
RQ3 (uncertainty resolution). Table 10 shows the number of guideline-based assertions that can be either conclusively or inconclusively verified by the HITECS model checking method. Note that all the seven HITECS test specifications have complex equations, e.g., containing logarithmic functions, that cannot be handled by model checking. Therefore, the numbers of conclusive and inconclusive assertions presented in the table are computed after replacing the complex equations with symbolic variables (see Section 5.1). Recall from Section 5.2 that the HITECS uncertainty resolution method takes an original HITECS specification as an input. For the inconclusive assertions, the uncertainty resolution method synthesizes conditions on the unknown parameters of the seven HiL test cases in our study under which they are guaranteed to be well-behaved.

Table 11: The number of identified conditions via the HITECS uncertainty resolution method and the measured execution time for identifying the conditions. The uncertainty resolution algorithm in Figure 15 is configured as follows: $ns = 10000$ and $nd = 10000$, using J48 with *the minimum number of instances per leaf = one* and *no pruning*.

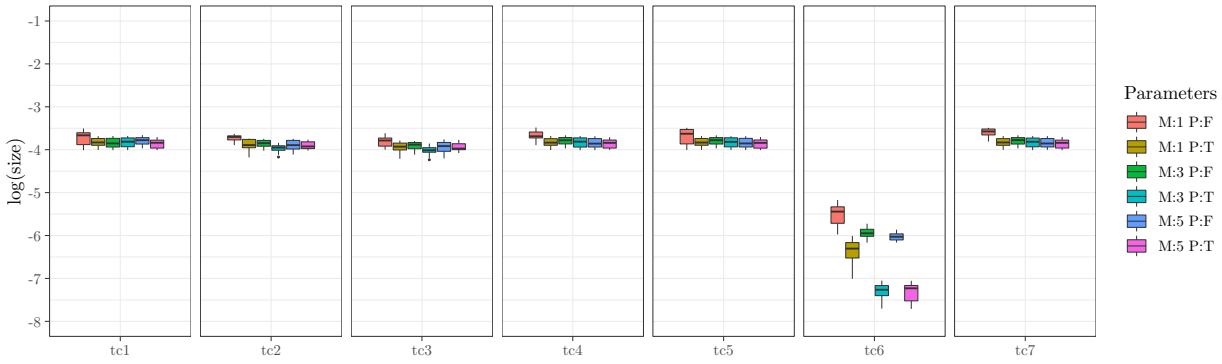
tc	# conds.	execution time			
		simulation	learning	model checking	total
tc1	6167	2.11h	3.02h	1.48h	6.61h
tc2	5009	1.95h	2.84h	1.35h	6.14h
tc3	5375	2.05h	2.82h	1.42h	6.29h
tc4	6050	2.17h	3.05h	1.53h	6.75h
tc5	6163	2.03h	3.03h	1.42h	6.48h
tc6	5117	1.78h	2.91h	1.32h	6.01h
tc7	5954	1.95h	3.05h	1.45h	6.45h
mean	5691	2.01h	2.96h	1.42h	6.39h

For each HITECS test specification, Table 11 shows the number of synthesized conditions and the execution time of the uncertainty resolution method. A condition in the table means a conjunction of predicates with the following form: $(min < var \leq max)$ where *var* is an unknown parameter (see example conditions in decision trees of Figure 16). Specifically, the number of synthesized conditions for *tc1* is 6167, meaning that the HITECS uncertainty resolution method generated 6167 distinct decision tree leaf nodes. These conditions are defined over the 11 unknown parameters of *tc1*, and *tc1* is guaranteed to be well-behaved under each of these conditions.

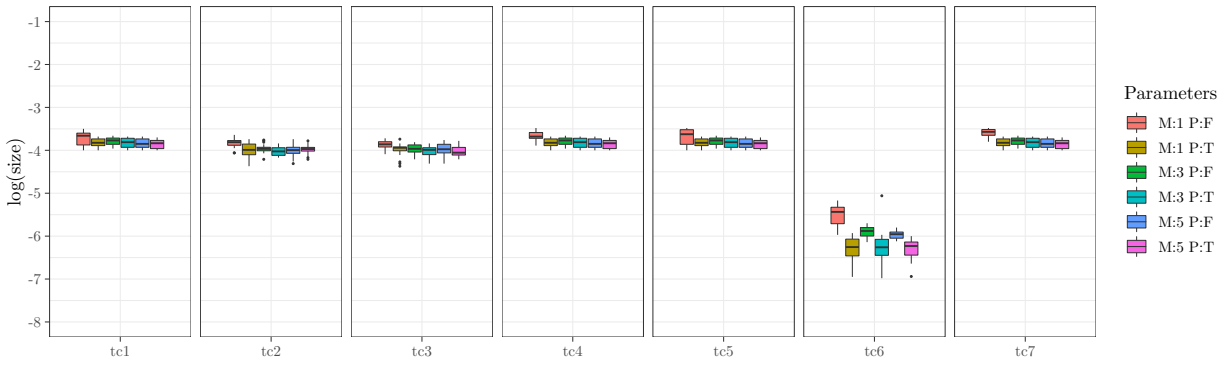
Note that test engineers in practice do not need to inspect individual conditions. At the actual time of HiL testing, they often have narrowed down the value ranges of unknown parameters into more restricted ranges compared to the initial and widely-estimated ranges they typically provide at the design-time testing. If the narrowed-down ranges are subsumed by some of the synthesized conditions, they can conclude that their HiL testing is going to



(a) J48



(b) SimpleCart



(c) REPTree

Figure 20: Region size distributions resulting from hyper parameter optimization for J48, SimpleCart, and REPTree. M: Minimal number of instances per leaf (1,3, or 5) and P: Pruned tree (T-pruned or F-unpruned).

be safe. Otherwise, they either need to find ways to further narrow down the ranges (e.g., by modifying hardware or by further restricting operating ranges of some devices) or they do not proceed with HiL testing due to high chances of hardware damage.

In addition, we measured the execution times of the HITECS uncertainty resolution procedure (see Figure 12). In this experiment, we set $ns = 10000$ and $nd = 10000$ in Figure 15. As shown in Table 11, one run of the HITECS uncertainty resolution method took, on average, 6.39h – 2.01h for simulation, 2.96h for decision tree learning, and 1.42h for model checking. The execution time is acceptable in practice as the HITECS uncertainty resolution method can be executed offline at design-time.

We further compared the impacts of using different ML-based decision tree learners on the HITECS uncertainty resolution method with different hyper-parameter values. Due to random variation in the uncertainty resolution method, we repeated our experiments 20 times. Figure 20 shows boxplots that compare J48, SimpleCart, and REPTree with different configurations of the minimum number of instances per leaf (M) and the boolean tree pruning flag (P). As shown in Figure 20, the HITECS uncertainty resolution method with J48, by setting $M = 1$ and $P = \mathit{false}$, produces the largest region size yielded by the synthesized conditions. We omit the corresponding statistical testing results of Figure 20 as the boxplots show visually significant differences between the best J48 configuration and the other configurations.

We note that the larger the region size, the more options to choose values for unknown parameters of HiL test cases. A small region size indicates that the test case is well-behaved only under restricted environment conditions and parameters. Engineers may choose to not execute test cases within such restrictions when they do not match their usual standards and normal situations. Alternatively, such test cases may be scheduled to run during hours when more test operators are available to fully observe test executions and act quickly if any risk of damage is present.

The answer to **RQ3** *is that* the HITECS uncertainty resolution method provides conditions on unknown parameters of HiL test cases under which the test cases are guaranteed to be well-behaved. Furthermore, we found the optimal configuration that leads to largest region sizes (hence, yielding most relaxed conditions on unknown parameters of HiL test cases). Using the optimal configuration, the uncertainty resolution method took, on average, 6.39h which is acceptable in general since our approach is an offline analysis that can be performed at design time and long before starting the actual HiL testing.

RQ4 (simulation). To estimate the execution time values of HiL test cases in our study, we ran each HITECS specification 3000 times, and created 3000 `traces` to be used by our simulation algorithm (see Section 5.3). We selected the number of simulation traces to be 3000 for two reasons: (1) The shapes and the ranges of execution-time distributions for all of our HITECS specifications started to stabilize when we used about 3000 simulation runs, and (2) the 95% confidence interval (CI) (Fisher, 1959) of the estimated execution time distributions obtained based on 3000 simulation runs is very small, i.e., less than ±1.5% of the mean estimated time, for all of our HITECS specifications. For example, the estimated execution time distribution of test case 4 in our study is as follows: the mean is 1924.96s, the standard deviation is 289.18, the minimum is 1037s, the 1st quartile is 1766.5s, the median is 2037s, the 3rd quartile is 2089s, and the maximum is 2718s. The 95% CI of test case 4 is thus 1924.96 (mean) ±10.35. The interval of 10.35 is computed by $1.96 \cdot 289.18/\sqrt{3000}$, where 1.96 is the Z-value for 95% confidence, 289.18 is the standard deviation, and 3000 is the number of simulation traces (Fisher, 1959).

Figure 21 compares the estimated execution time distributions computed by the HITECS simulation framework with some actual execution time samples for each HITECS specification in our study. Note that, in the figure, the actual execution time samples are shown as (red) dots around each distribution. Recall from Section 6.2 that the sample execution times are extracted from historical data files provided by SES Networks. We note that due to the confidentiality of most satellite operation information, we were provided with only a few historical data files from which at most seven sample execution times could be extracted for each test case. As shown in Figure 21, the actual execution time values are within the min-max ranges of their corresponding estimated distributions. Further, our domain experts validated the estimated distributions for each test case specification in our study.

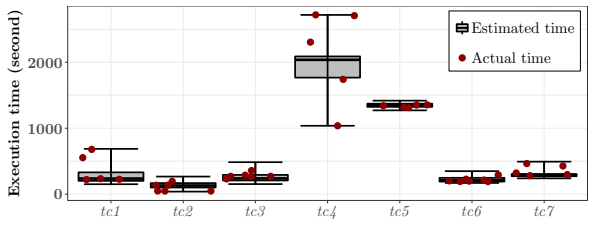


Figure 21: Comparing the estimated execution time distributions and the actual execution time samples of the seven HITECS specifications. Box plot: Min-25%-50%-75%-Max.

The answer to **RQ4** *is that* the HITECS simulation framework provides accurate execution time estimates for HiL test cases. Specifically, all the actual execution time samples of the HiL test cases in our study are within the min-max ranges computed by our simulation approach.

6.5. Threats to validity

This section discusses the validity considerations which are most relevant to our work, i.e., internal and external validity.

Internal validity. We mitigate potential biases and errors in our experiment data by using actual CPS HiL test data from SES Networks. First of all, we specified real satellite test procedures using HITECS. For HITECS model checking, HITECS assertion guideline was evaluated by comparing the effectiveness of two sets of assertions (1) prescribed by the guideline and (2) described in the real test procedures. For HITECS uncertainty resolution, initial value ranges of unknown parameters are determined by SES Networks. We also compared different ML-algorithms by controlling their hyper-parameters. For HITECS simulation, we used real historical data from SES Networks to estimate execution times of HiL test cases.

External validity. HITECS tailors existing generally applicable UML profiles, i.e., UTP, UUP, and Alf, for specifying and analyzing HiL testing in the context of CPS and is evaluated by applying it to a single industry case study in the satellite domain. Even though the case study system is a representative HiL testing of CPS in an actual industry setting, additional case studies are essential to validate our approach in different domains. In particular, our experiment results show that both HITECS model checking and HITECS uncertainty resolution spent practically acceptable time to execute. However, this finding requires further investigation as the performance of the underlying model checking technique of HITECS highly depends on system characteristics.

7. Related work

This section compares HITECS with different strands of related work in the area of (1) standards, (2) model checking and simulation in the context of verifying test cases and estimating their execution time, respectively, and (3) uncertainty-aware specification and analysis.

The standards that HITECS builds on, namely UTP and Alf, have been used in many research strands (Baker et al., 2007; Schieferdecker et al., 2003; Iber et al., 2015; Bagnato et al., 2013; Elaasar and Badreddin, 2016; Ciccozzi, 2016; Buchmann and Rimer, 2016; Seidewitz and Tatibouet, 2015; Seidewitz, 2017). For instance, UTP has been used as a base language to specify tests (e.g., Ubtl (Iber et al., 2015)), and Alf has been integrated into mainstream MDE tools (e.g., Papyrus (Seidewitz and Tatibouet, 2015) and MagicDraw (Seidewitz, 2017)) as an action language. To the best of our knowledge, HITECS is the first attempt at tailoring and extending UTP and Alf for specifying and analyzing CPS HiL test cases. More specifically, the extensions and improvements that HITECS offers over UTP (see Table 1) have not appeared in any prior work. The same can be said about how we utilize Alf for creating executable HiL test case specifications.

The European Telecommunications Standards Institute (ETSI) is responsible for developing standard languages for test specification. For example, the Testing and Test Control Notation (TTCN-3) (ETSI, b) and the Test Description Language (TDL) (ETSI, a) are standard languages developed by ETSI. TTCN-3 is a test specification language that has been applied in a variety of application domains such as telecommunication, transportation, and automotive. TDL is a language for describing test scenarios to fill the gap between informally-described test purposes and formally-defined test case specifications. UTP, which is the basis for HITECS, has been influenced by the concepts in ETSI standards, particularly those in TTCN-3. TTCN-3 and TDL, while being industry standards, are both generic test specification languages. In contrast, we have designed HITECS by following the paradigm of domain-specific modeling, with a specialized focus on CPS HiL testing.

Model checking and simulation have been widely used in a variety of application domains (Biere et al., 2003; Lindstrom et al., 2005; Grumberg and Veith, 2008; Enoiu et al., 2016; Adiego et al., 2015). However, verifying CPS HiL test cases and estimating their execution times have not been studied much in the existing work. Naik and Sarikaya (Naik and Sarikaya, 1993) use model checking to verify test cases developed for testing protocols. In their work, test case behaviors are expressed using extended state machines, and verified against safety and liveness properties formalized in temporal logic. Our work, in contrast, focuses on ensuring test case well-behavedness (see Table 6). To this end, we provide systematic guidelines to help engineers insert well-behavedness assertions into their test case specifications. Further, unlike Naik and Sarikaya, we demonstrate the effectiveness of our approach by empirically evaluating it on an industrial case study. Aranha and Borba (Aranha and Borba, 2007) propose an estimation model for test execution times. Their approach specifies test cases using a controlled natural language (CNL (Schwitter, 2002)), and estimates the execution times of test cases based on the size of CNL test case specifications and historical test execution data. As we argued earlier, estimating test execution times in the context of HiL testing involves uncertainty due to environmental factors. Hence, unlike the work of Aranha and Borba, we estimate test execution times as distributions (rather than point values) in order to account for such uncertainty. Further, our simulation annotations are flexible and can be used for estimating measures other than test execution times, e.g., the hardware wearout that may result from HiL testing. Finally, none of the above approaches provides a language to make test case specifications amenable to verification and simulation analysis.

Uncertainty has drawn a lot of attention in the software engineering research for supporting a variety of tasks, e.g., uncertainty modeling (Whittle et al., 2010; Zhang et al., 2016, 2019b), model checking in the presence of uncertainty (Bruns and Godefroid, 1999; Legay et al., 2010; Kwiatkowska et al., 2002), uncertainty-aware testing (Zhang et al., 2019a), and analyzing probabilistic assertions (Sampson et al., 2014). Whittle et al. (2010) propose RELAX, a requirements language for self-adaptive systems. RELAX provides a declarative language for specifying uncertainty in the context of adaptive systems and a formal semantics based on temporal fuzzy logic. Zhang et al. (2016) present U-Model, a conceptual model to understand uncertainty in the context of CPS. U-Model provides a systematic approach to identify, classify, and specify uncertainty at various development phases of CPS. Recently, Zhang et al. (2019b) introduce the UML Uncertainty Profile (UUP) which is based on the conceptual uncertainty model for CPS, i.e., U-Model. HITECS tailors UUP to capture uncertainty in the context of CPS testing. These prior modeling research strands aim at accounting for uncertainty concerns faced in various system development phases. In contrast to the above work, HITECS is specifically designed to capture the uncertainty in the context of CPS testing. HITECS specifications can be further analyzed through the following three uncertainty-aware analysis tasks: model checking, uncertainty resolution and simulation.

Some model checking research strands have been developed to account for uncertainty, represented either as unknown or random behavior, in their reasoning frameworks. For example, three-valued state abstraction has been used to deal with unknown systems' state spaces (Bruns and Godefroid, 1999). Three-valued model checking aims at determining conclusive or inconclusive verification results for system properties based on partially-known behaviors of a system. With respect to random behavior of a system, statistical or probabilistic model checking techniques have been proposed to verify stochastic systems that exhibit probabilistic behaviors (Legay et al., 2010; Kwiatkowska et al., 2002). HITECS model checking objectives align with those of the three-valued model checking in terms of identifying conclusive and inconclusive properties. Unlike statistical or probabilistic model checking, HITECS model checking does not provide quantitative analysis results (or probability estimations). Instead, HITECS provides an automated analysis method to identify conditions on uncertain parameters under which test cases are guaranteed to satisfy their assertions.

Zhang et al. (2019a) propose uncertainty-wise test case generation and minimization methods for CPS. Their test case generation approach is a model-based testing framework which requires an explicit model of system behavior in the presence of uncertainty in its operating environment. Their test case minimization technique is based on a multi-objective search optimizing cost, uncertainty, and effectiveness objectives. Unlike the prior research, HITECS is developed to directly specify and analyze CPS test cases which do not necessarily require detailed

behavioral models of the SUT. This enables test engineers to focus on specifying how the SUT is to be tested instead of its internal behavior.

Sampson et al. (2014) present an approach to specify probabilistic assertions and to verify these assertions. A probabilistic assertion states that the probability that a Boolean expression e holds in a given program execution is at least p with a confidence level c. Their assertion analysis workflow combines (1) Bayesian network for optimization and (2) sampling and hypothesis testing for verification. In contrast, HITECS model checking analyzes assertions and divides them into conclusive and inconclusive ones. For inconclusive assertions, HITECS uncertainty resolution uses sampling and ML techniques to identify specific conditions on uncertain parameters of test cases under which test cases are guaranteed to satisfy their inconclusive assertions.

8. Conclusions

This article studied for the first time the problem of specifying and analyzing CPS HiL test cases. HiL testing is a complex and time-consuming process. To minimize the risks associated with HiL testing, engineers have to ensure that: (1) HiL test cases are well-behaved, i.e., they implement valid test scenarios and do not accidentally damage hardware, as verified by assertions. (2) When the assertion checking of HiL test cases is inconclusive due to uncertainties in their behavior, we provide an uncertainty resolution strategy to identify conditions on uncertain parameters of HiL test cases under which they are guaranteed to satisfy their assertions. (3) The test cases execute within the time budget allotted to testing. We presented the HITECS specification and analysis framework, consisting of (1) an executable, uncertainty-aware modeling language for specifying HiL test cases and HiL platforms, (2) a verification method to ensure the well-behavedness of HiL test cases, (3) a strategy to resolve uncertainties by identifying conditions on parameters of HiL test cases under which they are guaranteed to satisfy their well-behavedness assertions, and (4) a simulation method to estimate the execution times of HiL test cases. We evaluated HITECS on an industrial case study in the satellite domain, which shares many of the common characteristics among Cyber-physical systems. Our evaluation showed that HITECS helps engineers define complete and effective assertions for checking the well-behavedness of HiL test cases, verify the well-behavedness of these test cases in practical time, resolve uncertainties by identifying conditions under which test cases can be conclusively verified, and accurately estimate the execution times of these test cases.

In the future, we would like to incorporate further analytical capabilities into our approach and extend the HITECS specification language to facilitate such new analysis tasks. For instance, emerging network technologies enable developing intelligent distributed CPS such as IoT-enabled emergency management systems (Shin et al., 2020) and smart manufacturing systems for Industry 4.0 (Lee et al., 2015). Testing such distributed CPS requires accounting for the underlying communication networks of the systems. We plan to extend HITECS to specify and analyze communication behaviors for testing intelligent distributed CPS. Another important direction for future work is to perform additional case studies in different application domains in order to more conclusively assess the applicability and usefulness of HITECS.

Acknowledgements

This project has received funding from SES, the Luxembourg National Research Fund under the grant C-16PPP/IS/11270448, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277) and NSERC of Canada under the Discovery, Discovery Accelerator, and CRC programs.

References

Abdessalem, R.B., Nejati, S., Briand, L.C., Stifter, T., 2018. Testing vision-based control systems using learnable evolutionary algorithms, in: Proceedings of the 40th International Conference on Software Engineering (ICSE'18), pp. 1016–1026.

Adiego, B.F., Darvas, D., nuela, E.B.V., Tournier, J., Bliudze, S., Blech, J.O., Suárez, V.M.G., 2015. Applying model checking to industrial-sized PLC programs. IEEE Transactions on Industrial Informatics 11, 1400–1410.

Ali, S., Yue, T., 2015. U-Test: Evolving, modelling and testing realistic uncertain behaviours of cyber-physical systems, in: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST'15), pp. 1–2.

Ammann, P., Offutt, J., 2016. Introduction to Software Testing. 2 ed., Cambridge University Press.

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments?, in: Proceedings of the 27th International Conference on Software Engineering (ICSE'05), pp. 402–411.

Aranha, E., Borba, P., 2007. An estimation model for test execution effort, in: Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM'07), pp. 107–116.

Arrieta, A., Sagardui, G., Etxeberria, L., Zander, J., 2017a. Automatic generation of test system instances for configurable cyber-physical systems. Software Quality Journal 25, 1041–1083.

Arrieta, A., Wang, S., Markiegi, U., Sagardui, G., Etxeberria, L., 2017b. Search-based test case generation for cyber-physical systems, in: Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC'17), pp. 688–697.

Asadollah, S.A., Inam, R., Hansson, H., 2015. A survey on testing for cyber physical system, in: Proceedings of the 27th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS'15), pp. 194–207.

Bagnato, A., Sadovykh, A., Brosse, E., Vos, T.E., 2013. The OMG UML testing profile in use–an industrial case study for the future internet testing, in: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13), pp. 457–460.

Baker, P., Dai, Z.R., Grabowski, J., Haugen, Ø., Schieferdecker, I., Williams, C., 2007. Model-Driven Testing: Using the UML Testing Profile. Springer-Verlag New York, Inc.

Bettaieb, S., Shin, S.Y., Sabetzadeh, M., Briand, L.C., Nou, G., Garceau, M., 2019. Decision support for security-control identification using machine learning, in: Proceedings of the 25th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'19), pp. 3–20.

Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y., 2003. Bounded model checking. Advances in Computers 58, 117–148.

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J., 1984. Classification and Regression Trees. Wadsworth International Group.

Bruns, G., Godefroid, P., 1999. Model checking partial state spaces with 3-valued temporal logics, in: Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99), pp. 274–287.

Buchmann, T., Rimer, A., 2016. Unifying modeling and programming with ALF, in: Proceedings of the 2nd International Conference on Advances and Trends in Software Engineering (SOFTENG'16), pp. 10–15.

Ciccozzi, F., 2016. On the automated translational execution of the action language for foundational UML. Software and Systems Modeling , 1–27.

Clarke, E., Kroening, D., Lerda, F., 2004. A tool for checking ANSI-C programs, in: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), pp. 168–176.

Clarke, E.M., Zuliani, P., 2011. Statistical model checking for cyber-physical systems, in: Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis (ATVA'11), pp. 1–12.

Clarke, Jr., E.M., Grumberg, O., Peled, D.A., 1999. Model Checking. MIT Press.

Elaasar, M., Badreddin, O., 2016. Modeling meets programming: A comparative study in model driven engineering action languages, in: Proceedings of the International Symposium on Leveraging Applications of Formal Methods (ISoLA'16), pp. 50–67.

Elbert, B.R., 2008. Introduction to Satellite Communication. 3 ed., Artech House.

Enoiu, E.P., Čaušević, A., Ostrand, T.J., Weyuker, E.J., Sundmark, D., Pettersson, P., 2016. Automated test generation using model checking: An industrial evaluation. International Journal on Software Tools for Technology Transfer (ICTSS'16) 18, 335–353.

ETSI, 2017b. Testing and Test Control Notation version 3. ETSI Standard.

ETSI, 2018a. Test Description Language. ETSI Standard.

Fisher, R.A., 1959. Statistical Methods and Scientific Inference. Oliver & Boyd.

Grumberg, O., Veith, H. (Eds.), 2008. 25 Years of Model Checking: History, Achievements, Perspectives. Springer-Verlag.

Iber, J., Kajtazović, N., Höller, A., 2015. Ubtl UML testing profile based testing language, in: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, pp. 1–12.

Jensen, J.C., Chang, D.H., Lee, E.A., 2011. A model-based design methodology for cyber-physical systems, in: Proceedings of the 7th International Wireless Communications and Mobile Computing Conference (IWCMC'11), pp. 1666–1671.

Jeruchim, M.C., Balaban, P., Shanmugan, K.S. (Eds.), 2000. Simulation of Communication Systems: Modeling, Methodology and Techniques. 2nd ed., Kluwer Academic Publishers.

Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37, 649–678.

Kwiatkowska, M.Z., Norman, G., Parker, D., 2002. PRISM: probabilistic symbolic model checker, in: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS'02), pp. 200–204.

Lee, E.A., 2008. Cyber physical systems: Design challenges, in: Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC'08), pp. 363–369.

Lee, J., Bagheri, B., Kao, H., 2015. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. Manufacturing Letters 3, 18–23.

Legay, A., Delahaye, B., Bensalem, S., 2010. Statistical model checking: An overview, in: Proceedings of the 1st International Conference on Runtime Verification (RV'10), pp. 122–135.

Lindstrom, G., Mehlitz, P.C., Visser, W., 2005. Model checking real time Java using Java Pathfinder, in: Proceedings of the 3rd International Conference on Automated Technology for Verification and Analysis (ATVA'05), pp. 444–456.

Liu, B., Nejati, S., Lucia, L., Briand, L.C., 2019. Effective fault localization of automotive Simulink models: Achieving the trade-off between test oracle effort and fault localization accuracy. Empirical Software Engineering (EMSE) 24, 444–490.

Luke, S., 2013. Essentials of Metaheuristics. second ed., Lulu. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T., 2019. Test generation and test prioritization for Simulink models with dynamic behavior. IEEE Transactions on Software Engineering (TSE) 45, 919–944.

Menghi, C., Nejati, S., Briand, L.C., Isasi Parache, Y., 2020. Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system

identification, in: Proceedings of the 42nd International Conference on Software Engineering (ICSE'20), pp. 1–13.

van der Merwe, H., van der Merwe, B., Visser, W., 2012. Verifying android applications using Java PathFinder. ACM SIGSOFT Software Engineering Notes 37, 1–5.

Mosterman, P.J., Zander, J., 2016. Cyber-physical systems challenges: A needs analysis for collaborating embedded software systems. Software and Systems Modeling (SoSyM'16) 15, 5–16.

de Moura, L.M., Bjørner, N., 2008. Z3: an efficient SMT solver, in: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), pp. 337–340.

Naik, K., Sarikaya, B., 1993. Test case verification by model checking. Formal Methods in System Design 2, 277–321.

Nejati, S., Gaaloul, K., Menghi, C., Briand, L.C., Foster, S., Wolfe, D., 2019. Evaluating model testing and model checking for finding requirements violations in simulink models, in: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19), pp. 1015–1025.

Nguyen, P.H., Ali, S., Yue, T., 2017. Model-based security engineering for cyber-physical systems: A systematic mapping study. Information and Software Technology 83, 116–135.

OMG, 2011b. OMG Unified Modeling Language (OMG UML), Superstructure. OMG Specification.

OMG, 2017a. Action Language for Foundational UML (Alf). OMG Specification.

OMG, 2017c. Semantics of a Foundational Subset for Executable UML Models (fUML). OMG Specification.

OMG, 2017d. UML Testing Profile (UTP) Version 2.0 - Beta. OMG Specification.

Petrovic, G., Ivankovic, M., Kurtz, B., Ammann, P., Just, R., 2018. An industrial application of mutation testing: Lessons, challenges, and research directions, in: Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, (ICST Workshops'18), pp. 47–53.

Quinlan, J.R., 1987. Simplifying decision trees. International Journal of Man-Machine Studies 27, 221–234.

Quinlan, J.R., 1993. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc.

Safavian, S.R., Landgrebe, D.A., 1991. A survey of decision tree classifier methodology. IEEE Transactions on Systems, Man, and Cybernetics (TSMC) 21, 660–674.

Saleh, A.A.M., 1981. Frequency-independent and frequency-dependent nonlinear models of TWT amplifiers. IEEE Transactions on Communications 29, 1715–1720.

Sampson, A., Panchekha, P., Mytkowicz, T., McKinley, K.S., Grossman, D., Ceze, L., 2014. Expressing and verifying probabilistic assertions, in: Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), pp. 112–122.

Schieferdecker, I., Dai, Z.R., Grabowski, J., Rennoch, A., 2003. The UML 2.0 testing profile and its relation to TTCN-3, in: Proceedings of the 15th IFIP International Conference on Testing of Communicating Systems (TestCom'03), pp. 79–94.

Schwitter, R., 2002. English as a formal specification language, in: Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA'02), pp. 228–232.

Seidewitz, E., 2017. A development environment for the Alf language within the MagicDraw UML tool (tool demo), in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE'17), pp. 217–220.

Seidewitz, E., Tatibouet, J., 2015. Tool paper: Combining Alf and UML in modeling tools - an example with Papyrus, in: Proceedings of the 15th International Workshop on OCL and Textual Modeling (OCL'15), pp. 105–119.

Shin, S.Y., Chaouch, K., Nejati, S., Sabetzadeh, M., Briand, L.C., Zimmer, F., 2018a. HITECS: A UML profile and analysis framework for hardware-in-the-loop testing of cyber physical systems, in: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'18), pp. 357–367.

Shin, S.Y., Chaouch, K., Nejati, S., Sabetzadeh, M., Briand, L.C., Zimmer, F., 2019. [case study data] uncertainty-aware specification and analysis for hardware-in-the-loop testing of cyber physical systems. `https://gitlab.uni.lu/ChaouchKarim/hitecs`.

Shin, S.Y., Nejati, S., Sabetzadeh, M., Briand, L.C., Arora, C., Zimmer, F., 2020. Dynamic adaptation of software-defined networks for iot systems: A search-based approach, in: Proceedings of the 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'20), pp. 1–12.

Shin, S.Y., Nejati, S., Sabetzadeh, M., Briand, L.C., Zimmer, F., 2018b. Test case prioritization for acceptance testing of cyber physical systems: A multi-objective search-based approach, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18), pp. 49–60.

Thacker, R.A., Jones, K.R., Myers, C.J., Zheng, H., 2010. Automatic abstraction for verification of cyber-physical systems, in: Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS'10), pp. 12–21.

Ul Haq, F., Shin, D., Nejati, S., Briand, L.C., 2020. Comparing offline and online testing of deep neural networks: An autonomous car case study, in: Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST'20), pp. 1–11.

Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F., 2003. Model checking programs. Automated Software Engineering (ASE) 10, 203–232.

Visser, W., Păsăreanu, C.S., Khurshid, S., 2004. Test input generation with Java PathFinder, in: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04), pp. 97–107.

Wang, C., Pastore, F., Goknil, A., Briand, L., Iqbal, Z., 2015. Automatic generation of system test cases from use case specifications, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15), pp. 385–396.

Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J., 2010. RELAX: A language to address uncertainty in self-adaptive systems requirement. Requirements Engineering (RE) 15, 177–196.

Witten, I.H., Frank, E., Hall, M.A., 2011. Data Mining: Practical Machine Learning Tools and Techniques. 3rd ed., Morgan Kaufmann Publishers Inc.

Yao, X., Harman, M., Jia, Y., 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence, in: Proceedings of the 36th International Conference on Software Engineering (ICSE'14), pp. 919–930.

Zhang, M., Ali, S., Yue, T., 2017. Uncertainty-wise Test Case Generation and Minimization for Cyber-Physical Systems. Technical Report 2016-13. Simula Research Laboratory.

Zhang, M., Ali, S., Yue, T., 2019a. Uncertainty-wise test case generation and minimization for cyber-physical systems. Journal of Systems and Software (JSS) 153, 1–21.

Zhang, M., Ali, S., Yue, T., Norgren, R., Okariz, O., 2019b. Uncertainty-wise cyber-physical system test modeling. Software and Systems Modeling (SoSyM) 18, 1379–1418.

Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., Norgren, R., 2016. Understanding uncertainty in cyber-physical systems: A conceptual model, in: Proceedings of the 12th European Conference on Modelling Foundations and Applications (ECMFA'16), pp. 247–264.

Zheng, X., Julien, C., 2015. Verification and validation in cyber physical systems: Research challenges and a way forward, in: Proceedings of the 1st International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS'15), pp. 15–18.