HITECS: A UML Profile and Analysis Framework for Hardware-in-the-Loop Testing of Cyber Physical Systems

Seung Yeob Shin, Karim Chaouch, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand
SnT Centre, University of Luxembourg, Luxembourg
{seungyeob.shin,karim.chaouch,shiva.nejati,mehrdad.sabetzadeh,lionel.briand}@uni.lu

Frank Zimmer
SES Networks, Luxembourg
frank.zimmer@ses.com

ABSTRACT

Hardware-in-the-loop (HiL) testing is an important step in the development of cyber physical systems (CPS). CPS HiL test cases manipulate hardware components, 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) HiL test cases are well-behaved, i.e., they implement valid test scenarios and do not accidentally damage hardware, and (2) HiL test cases can execute within the time budget allotted to HiL testing. This paper proposes an approach to help engineers systematically specify and analyze CPS HiL test cases. Leveraging the UML profile mechanism, we develop an executable domain-specific language, HITECS, for HiL test case specification. HITECS builds on the UML Testing Profile (UTP) and the UML action language (Alf). Using HITECS, we provide analysis methods to check whether HiL test cases are well-behaved, and to estimate the execution times of these test cases before the actual HiL testing stage. We apply HITECS to an industrial case study from the satellite domain. Our results show that: (1) HITECS is feasible to use in practice; (2) HITECS helps engineers define more complete and effective well-behavedness assertions for HiL test cases, compared to when these assertions are defined without systematic guidance; (3) HITECS verifies in practical time that HiL test cases are well-behaved; and (4) HITECS accurately estimates HiL test case execution times.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Specification languages**; **Software verification and validation**;

KEYWORDS

Test Case Specification and Analysis, Cyber Physical Systems, UML Profile, Model Checking and Simulation, JavaPathFinder

ACM Reference Format:
Seung Yeob Shin, Karim Chaouch, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand and Frank Zimmer. 2018. HITECS: A UML Profile and Analysis Framework for Hardware-in-the-Loop Testing of Cyber Physical Systems. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239382

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239382

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 [25] – typically takes place at the far end of the system quality assurance spectrum and as part of acceptance testing [5].

An important characteristic of HiL testing is that, due to the involvement of actual hardware, 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 software, this is not the objective during HiL testing.

A second 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. A key complexity that arises when performing such estimations is the uncertainty in the SUT environment. 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 paper, we develop an *executable* language for specifying HiL test cases and HiL platforms. Our language aims at enabling

the two tasks described above, i.e., well-behavedness checking of HiL test cases and estimating their execution times. Both 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 [16, 24, 27, 33, 41, 51]. In particular and in the area of model-based testing, approaches exist for automated generation of CPS test cases [7, 8, 50]. 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 [48]. Industry standards such as TTCN-3 [42] and UTP [44] 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 paper 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* [43]. A key characteristic of HITECS is that it has an execution semantics and specific constructs to capture uncertainty. HITECS customizes the UML Testing Profile (UTP) [44] to the HiL testing context. To do so, HITECS further uses the textual syntax of the Action Language for Foundational UML (Alf) [3], adopting Alf's execution semantics. HITECS is informed and motivated by the existing literature on HiL testing [1, 4, 9].

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, and (ii) 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 [46], to HITECS test specifications in order to determine whether they satisfy their assertions. 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 real-world 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 conclusively verifies satellite HiL test cases in practical time; and (iv) HITECS simulation provides accurate estimates for the execution times of satellite HiL test cases.

Structure. Section 2 motivates the paper. 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 paper.

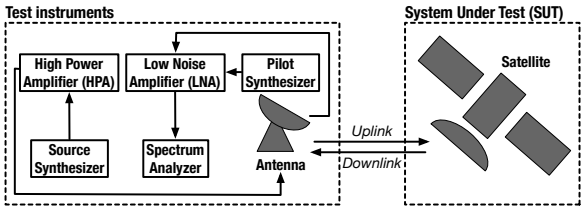

Figure 1: A simplified and partial view of the HiL test platform for a satellite after launch [39].

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 [39]: *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.

During our collaboration with our industry partner, 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. Engineers therefore need techniques to ensure that HiL test cases are well-behaved and exercise valid test scenarios prior to executing them on the actual HiL

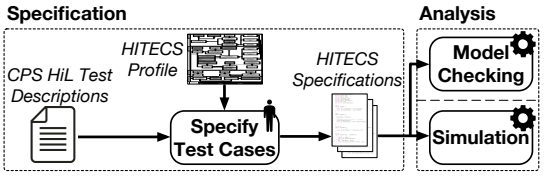

Figure 2: HITECS overview.

platform. In Section 3, we precisely define the well-behavedness requirements that HiL test cases should meet.

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.

The analysis tasks discussed above are not specific to in-orbit testing of satellites, and are recurring in other CPS domains, as observed in both our earlier work [1, 29, 30] and that of others [27, 31, 51]. In the next sections, we provide an approach for specifying, verifying and simulating HiL test cases in such a way that the above tasks can be performed systematically and with computerized support.

3 OVERVIEW

Figure 2 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 two 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 [17] to HITECS test specifications to determine whether the assertions hold.

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.

Table 1: HITECS contributions to UTP.

C#	Descriptions of HITECS contributions to UTP
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 enables model checking of HiL test cases for well-behavedness
C5	HITECS provides simulation facilities for estimating the execution time of HiL test cases

4 TEST SPECIFICATION

HITECS tailors the UML Testing Profile (UTP) [44] and Action Language for Foundational UML (Alf) [3] to specify CPS HiL test cases. In Section 4.1, we provide background on UTP and Alf, and in Section 4.2, we present HITECS.

4.1 Background on UTP 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.

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 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) [21].

4.2 The HITECS language

HITECS extends UTP and uses the textual notation and executable semantics of Alf. Since Alf and UTP are both 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 formalization of acceptance testing concepts in our earlier work [39] and the CPS testing literature [1, 4, 9]. Table 1 outlines the main improvements and extensions that HITECS provides over UTP. Overall, HITECS provides five new extensions that are instrumental either to specifying HiL test cases, or to verifying or simulating them. In this section, we introduce HITECS by describing and illustrating the contributions outlined in Table 1. Figure 3 shows the HITECS profile. As shown by the figure,

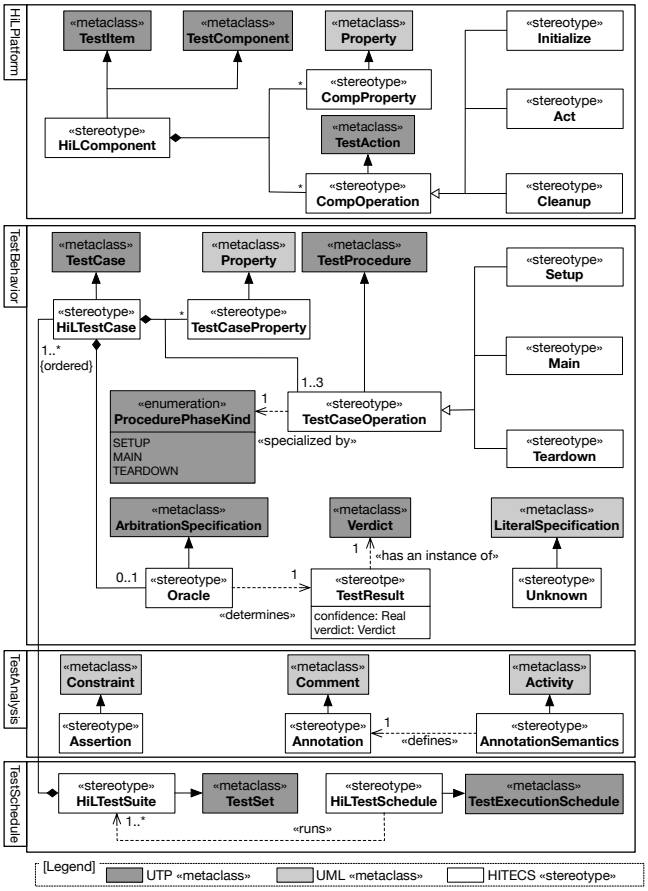

Figure 3: The HITECS profile (extension of UTP).

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

HITECS is organized into four packages, `HiLPlatform`, `TestBehavior`, `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 blackbox 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 the internals of the components or how they interact with one another.

Table 2 describes the modeling concepts in the `HiLPlatform` package of Figure 3. 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

```
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 4: HITECS specification of the synthesizer in Figure 1.

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
`TestCaseProperty`	Inputs, outputs, and HiL components used by a test case
`TestCaseOperation`	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
`Unknown`	A literal constant to represent uncertainty

tagged by `@Initialize`, `@Act`, and `@Cleanup` annotations. Note that a component may have several operations tagged as `@Act`. 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 4 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 paper, we use /* ... */ (e.g., line 6 in Figure 4) in HITECS specification figures to omit irrelevant details.

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

Test cases. A test case is defined by `HiLTestCase` and includes `TestCaseProperty`, `TestCaseOperation`, 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`

```
1  testcase TranslationFrequency {                    HITECS
2    private expTF: Real;
3    private meaTF: Real;
4    private frequency: Real;
5    private sat: Satellite;
6    private synth: Synthesizer;
7    private acu: ACU;
8    private sa: SpectrumAnalyzer;
9
10   @Setup
11   public setup() {
12     /* ... */
13     assert acu.satLongitude == sat.longitude;
14     assert acu.satLatitude == sat.latitude;
15   }
16
17   @Main
18   public measure() {
19     /* ... */
20     attenuation = Unknown;
21     /* ... */
22     //@SimTime("synth.time.record", "uniform")
23     synth.generateSignal();
24     assert sa.PowerLevel() < Const::PowerThreshold;
25     /* ... */
26   }
27
28   @Teardown
29   public teardown() {
30     /* ... */
31     assert synth.RFMode() == Synthesizer::OFF;
32   }
33
34   @Oracle
35   public testOracle() : TestResult {
36     /* ... */
37     if (meaTF == expTF) {
38       return new TestResult(PASS);
39     } else {
40       diff = Abs(meaTF - expTF);
41       return new TestResult(diff/(1+diff), FAIL);
42     }
43   }
44 }
```

Figure 5: (Simplified) HITECS specification for the translation frequency test case.

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.

Figure 5 shows an example of a HITECS specification for the translation frequency test case of a satellite. The expTF and meaTF variables specify the expected and the actual outputs of the test case, respectively; frequency is the test input; and 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 translation frequency function of the satellite under test; and the teardown() operation turns off the HiL component (e.g., synthesizer) used by the test case.

Test oracles. A test oracle determines whether a test case execution passes or fails. In HITECS, TestResult extends UTP's Verdict with 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.

Table 4: 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)
AnnotationSemantics	An (operational) semantics of a simulation annotation

Table 5: 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

Specifically, testOracle() (simplified for exposition) in Figure 5 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.

Uncertainties. 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. 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 5 shows an example of using Unknown for attenuation. 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 verification and simulation (discussed in Section 5), the Unknown literals are replaced with random-number generators, which yield random numbers with the same type as the respective uncertain variables and within ranges specified by engineers.

Test analysis. The TestAnalysis package in Figure 3 contains the modeling concepts used for verification and simulation (Section 5). Table 4 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 TranslationFrequency test specification in Figure 5 exemplifies Assertion and Annotation. As for AnnotationSemantics, an example is provided in Figure 6 (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 5, 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 5 are related to R1 in Table 5 and specify that an antenna must point to the satellite under test after executing setup() in Figure 5. The assertion on line 24 in Figure 5 is related to R5 in Table 5 and specifies that the power of the signals sent to a satellite must be less than a threshold to avoid any damage

```
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 input
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 6: HITECS specification of `@SimTime` semantics.

to the satellite. Finally, the assertion on line 31 in Figure 5 is related to R3 in Table 5 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 5 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 5 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 5.2). 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 represented 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 22 in Figure 5 is an annotation providing information about the execution times of its next statement, i.e., line 23. 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 6 illustrates the semantic routine related to the `@SimTime` annotation in Figure 5. 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

Table 6: 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

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

Figure 7: HITECS specification of a test scheduler.

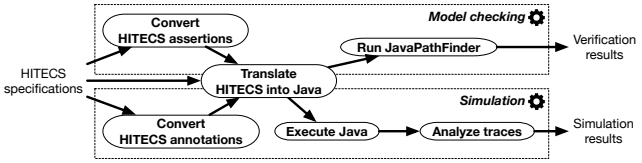

Figure 8: Overview of the analysis component in Figure 2 performing model checking and simulation.

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 6 (lines 7–10), the `@SimTime` annotation in Figure 5 indicates that the execution time of the statement `synth.generateSignal()` can take, with an equal probability, any value from `synth.time.record`. In Section 5.2, we will discuss how the annotation semantics are used by our simulator. Note that, as we discuss in Section 5.2, 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 6 describes the stereotypes in `TestSchedule` of HITECS. For instance, `ScheduleInOrbitTest` in Figure 7 runs the test cases in the `InOrbitSatTest` test suite based on the order specified in `suite`. Line 4 in Figure 7 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 3); hence, they may or may not be invoked by test schedules.

5 SPECIFICATION ANALYSIS

In this section, we describe how HITECS enables model checking and simulation of HiL test cases. Figure 8 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"; and HITECS simulation contains the following four steps: "Convert HITECS Annotations", "Translate HITECS into Java", "Execute Java", and "Analyze traces". Both analysis tasks translate HITECS specifications into Java (see the common step "Translate HITECS into Java" in Figure 8). 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++. Due to

Table 7: Guidelines prescribing assertions to be inserted into HITECS specifications to verify the requirements in Table 5.

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

space limitations, we omit the technical details of the translation, and refer the interested reader to existing work [13, 14]. Below, we explain the steps of model checking 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 5. 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 7 presents our guidelines for specifying the assertions induced by the requirements in Table 5. The guidelines specify the content of assertions and their expected locations in HITECS specifications. For example, the guideline in the first row of Table 7 (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 5. The two assertions on lines 13 and 14 in Figure 3 are written based on this guideline. Similarly, the assertion on line 24 in Figure 3 is written based on the guideline on the second row of Table 7. Finally, the assertion on line 31 in Figure 3 follows the guideline on the last row of Table 7.

Having defined our guidelines for assertion specification, we now describe the steps of HITECS model checking in Figure 8. The "Convert HITECS assertions" step converts HITECS assertions into Java 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 (Java) random-number generators, as explained in Section 4.2. We apply JavaPathFinder [46] – a well-known and widely-used model checking tool – to the generated Java translations. Applying JavaPathFinder to a Java program containing assertions leads to one of the following situations: (1) JavaPathFinder terminates and computes inputs violating some assertions in the program, or (2) JavaPathFinder terminates and reports that all assertions hold for all inputs, or (3) JavaPathFinder fails to terminate within the time allotted. In cases (1) and (2), we say JavaPathFinder is *conclusive*, and in case (3), say it is *inconclusive*. 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 [15]).

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

Figure 9: A pre-defined aggregator function of the "Analyze traces" step in Figure 8.

5.2 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 8. We further show how the simulation annotations can be used for estimating HITECS specification execution times.

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 9 shows (in pseudo-code form) an example aggregator function, pre-defined by the "Analyze traces" step. This function computes a vector `v` such that every element of `v` 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 10 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 `@SimTime` and provide a `SimTime` semantic routine (e.g., Figure 6). We use the `Sum` aggregator function in Figure 9 to combine the

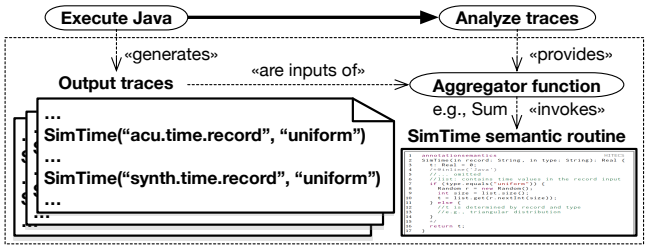

Figure 10: Estimating the execution times of HiL test cases using the HITECS simulation in Figure 8.

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 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 10, 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 `v`. 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 [38].

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 conclusively 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 [28, 45, 47], it has not been previously evaluated for CPS test cases. In RQ2, we investigate whether JavaPathFinder is able to conclusively verify well-behavedness assertions of industry HiL test cases in practical time (see the definition of conclusiveness in Section 5.1).

RQ3 (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 5.2). 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 our industry partner, SES Networks. Using textual documents provided by our industry partner on 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 and between two to 27 operations. Each test case specification has five input parameters and interacts with between 13 to 15 components.

The textual descriptions from our industry partner 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 7.

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 RQ3.

6.3 Experiment design

To answer RQ1 and RQ2, we rely on mutation analysis [26] 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. [49], 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 7. On average, per specification, we had 106.1 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.

To answer RQ3, 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 5).

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

6.4 Results

RQ1. We applied JavaPathFinder to the 609 mutants containing ad-hoc assertions and to the 609 mutants containing assertions prescribed by our guidelines. Table 8 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.

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.

RQ2. JavaPathFinder was able to conclusively 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 11 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.

Table 8: 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

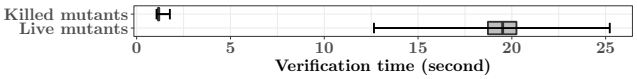

Figure 11: HITECS verification time for the live mutants and killed mutants. Box plot: Min-25%-50%-75%-Max.

We note that the conclusiveness and 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 speed and conclusiveness 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 conclusively verified all the well-behavedness assertions in less than 25.2s.

RQ3. 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 5.2). 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) [20] 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.

Figure 12 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 our industry partner. We note that due to the confidentiality of most satellite operation information, we were provided

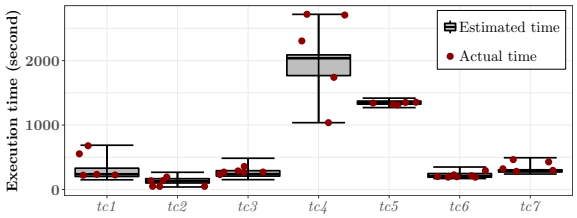

Figure 12: Comparing the estimated execution time distributions and the actual execution time samples of the seven HITECS specifications. Box plot: Min-25%-50%-75%-Max.

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 12, the actual execution time values are within the maximum and minimum ranges of their corresponding estimated distributions. Further, our domain experts validated the estimated distributions for each test case specification in our study.

The answer to **RQ3** *is that* the HITECS simulation framework provides accurate execution time estimations for HiL test cases. Specifically, all the actual execution time samples of the HiL test cases in our study are within the maximum and minimum ranges computed by our simulation approach.

7 RELATED WORK

This section compares HITECS with the existing model-based frameworks for the specification and analysis of test cases.

The standards that HITECS builds on, namely UTP and Alf, have been used in many research strands [10, 11, 13, 14, 18, 23, 34, 36, 37]. For instance, UTP has been used as a base language to specify tests (e.g., Ubtl [23]), and Alf has been integrated into mainstream MDE tools (e.g., Papyrus [37] and MagicDraw [36]) 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) [42] and the Test Description Language (TDL) [40] 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 [2, 12, 19, 22, 28]. However, verifying CPS HiL test cases and estimating their execution times have not been studied much in the existing work. Naik and Sarikaya [32] 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 5). 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 [6] propose an estimation model for test execution times. Their approach specifies test cases using a controlled natural language (CNL [35]), 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 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.

8 CONCLUSIONS

This paper 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, and (2) 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, and (3) a simulation method to estimate the execution times of HiL test cases. We evaluated HITECS on an industrial case study in the satellite domain. 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, and accurately estimate the execution times of these test cases. In the future, we would like to incorporate further analytical capabilities into our approach, e.g., resource utility optimization during HiL testing. 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.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 1–11. (in press).

[2] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Vi nuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor Manuel González Suárez. 2015. Applying Model Checking to Industrial-Sized PLC Programs. *IEEE Transactions on Industrial Informatics* 11, 6 (2015), 1400–1410.

[3] Alf 2017. *Action Language for Foundational UML (Alf)*. OMG Specification formal/2017-07-04. Object Management Group.

[4] Shaukat Ali and Tao Yue. 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)*. 1–2.

[5] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (2 ed.). Cambridge University Press.

[6] Eduardo Aranha and Paulo Borba. 2007. An Estimation Model for Test Execution Effort. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM'07)*. 107–116.

[7] Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria, and Justyna Zander. 2017. Automatic Generation of Test System Instances for Configurable Cyber-physical Systems. *Software Quality Journal* 25, 3 (2017), 1041–1083.

[8] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2017. Search-based test case generation for Cyber-Physical Systems. In *Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC'17)*. 688–697.

[9] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. 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)*. 194–207.

[10] Alessandra Bagnato, Andrey Sadovykh, Etienne Brosse, and Tanja E.J. Vos. 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)*. 457–460.

[11] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams. 2007. *Model-Driven Testing: Using the UML Testing Profile*.

[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Advances in Computers* 58 (2003), 117–148.

[13] Thomas Buchmann and Alexander Rimer. 2016. Unifying Modeling and Programming with ALF. In *Proceedings of the 2nd International Conference on Advances and Trends in Software Engineering (SOFTENG'16)*. 10–15.

[14] Federico Ciccozzi. 2016. On the automated translational execution of the action language for foundational UML. *Software and Systems Modeling* (2016), 1–27.

[15] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 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)*. 168–176.

[16] Edmund M. Clarke and Paolo Zuliani. 2011. Statistical Model Checking for Cyber-physical Systems. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis (ATVA'11)*. 1–12.

[17] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press.

[18] Maged Elaasar and Omar Badreddin. 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)*. 50–67.

[19] Eduard Paul Enoiu, Adnan Čaušević, Thomas J. Ostrand, Elaine J. Weyuker, Daniel Sundmark, and Paul Pettersson. 2016. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer (ICTSS'16)* 18, 3 (2016), 335–353.

[20] Ronald A. Fisher. 1959. *Statistical Methods and Scientific Inference*. Oliver & Boyd.

[21] fUML 2017. *Semantics of a Foundational Subset for Executable UML Models (fUML)*. OMG Specification formal/2017-07-02. Object Management Group.

[22] Orna Grumberg and Helmut Veith (Eds.). 2008. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag.

[23] Johannes Iber, Nermin Kajtazović, and Andrea Höller. 2015. Ubtl UML Testing Profile based Testing Language. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. 1–12.

[24] Jeff C. Jensen, Danica H. Chang, and Edward A. Lee. 2011. A Model-Based Design Methodology for Cyber-Physical Systems. In *Proceedings of the 7th International Wireless Communications and Mobile Computing Conference (IWCMC'11)*. 1666–1671.

[25] Michel C. Jeruchim, Philip Balaban, and K. Sam Shanmugan (Eds.). 2000. *Simulation of Communication Systems: Modeling, Methodology and Techniques* (2nd ed.). Kluwer Academic Publishers.

[26] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.

[27] Edward A. Lee. 2008. Cyber Physical Systems: Design Challenges. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC '08)*. 363–369.

[28] Gary Lindstrom, Peter C. Mehlitz, and Willem Visser. 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)*. 444–456.

[29] Bing Liu and Lionel C. Briand Shiva Nejati, Lucia Lucia. 2018. Effective Fault Localization of Automotive Simulink Models: Achieving the Trade-Off between Test Oracle Effort and Fault Localization Accuracy. *Empirical Software Engineering* (2018), 1–47. (in press).

[30] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2018. Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior. *IEEE Transactions on Software Engineering (TSE'18)* (2018), 1–25. (in press).

[31] Pieter J. Mosterman and Justyna Zander. 2016. Cyber-physical Systems Challenges: A Needs Analysis for Collaborating Embedded Software Systems. *Software and Systems Modeling (SoSyM'16)* 15, 1 (2016), 5–16.

[32] Kshirasagar Naik and Behcet Sarikaya. 1993. Test Case Verification by Model Checking. *Formal Methods in System Design* 2 (1993), 277–321. Issue 3.

[33] Phu H. Nguyen, Shaukat Ali, and Tao Yue. 2017. Model-Based Security Engineering for Cyber-Physical Systems: A Systematic Mapping Study. *Information and Software Technology* 83 (2017), 116–135.

[34] Ina Schieferdecker, Zhen Ru Dai, Jens Grabowski, and Axel Rennoch. 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)*. 79–94.

[35] Rolf Schwitter. 2002. English as a Formal Specification Language. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA'02)*. 228–232.

[36] Ed Seidewitz. 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)*. 217–220.

[37] Ed Seidewitz and Jérémie Tatibouet. 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)*. 105–119.

[38] Seung Yeob Shin, Karim Chaouch, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. 2018. [Case study data] HITECS: A UML Profile and Analysis Framework for Hardware-in-the-Loop Testing of Cyber Physical Systems. https://github.com/ChaouchKarim/HITECS. (2018).

[39] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. 2018. 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)*. 1–12. (in press).

[40] TDL 2018. *Test Description Language*. ETSI Standard TR 103 119. ETSI.

[41] Robert A. Thacker, Kevin R. Jones, Chris J. Myers, and Hao Zheng. 2010. Automatic Abstraction for Verification of Cyber-physical Systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS'10)*. 12–21.

[42] TTCN-3 2017. *Testing and Test Control Notation version 3*. ETSI Standard ES 201 873-1. ETSI.

[43] UML Profile 2011. *OMG Unified Modeling Language (OMG UML), Superstructure*. OMG Specification formal/2011-08-06. Object Management Group.

[44] UTP 2017. *UML Testing Profile (UTP) Version 2.0 - Beta*. OMG Specification ptc/2017-09-29. Object Management Group.

[45] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[46] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (2003), 203–232.

[47] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*. 97–107.

[48] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. 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)*. 385–396.

[49] Xiangjuan Yao, Mark Harman, and Yue Jia. 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)*. 919–930.

[50] Man Zhang, Shaukat Ali, and Tao Yue. 2017. *Uncertainty-wise Test Case Generation and Minimization for Cyber-Physical Systems*. Technical Report 2016-13. Simula Research Laboratory. 1–31 pages.

[51] Xi Zheng and Christine Julien. 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)*. 15–18.