Oracles for Testing Software Timeliness with Uncertainty

CHUNHUI WANG, University of Luxembourg - Interdisciplinary Centre for Security, Reliability and Trust, Luxembourg
FABRIZIO PASTORE, University of Luxembourg - Interdisciplinary Centre for Security, Reliability and Trust, Luxembourg
LIONEL BRIAND, University of Luxembourg - Interdisciplinary Centre for Security, Reliability and Trust, Luxembourg

Uncertainty in timing properties (e.g., detection time of external events) is a common occurrence in embedded software systems since these systems interact with complex physical environments.

Such time uncertainty leads to non-determinism. For example, time-triggered operations may either generate different valid outputs across different executions, or experience failures (e.g., results not being generated in the expected time window) that occur only occasionally over many executions. For these reasons, time uncertainty makes the generation of effective test oracles for timing requirements a challenging task.

To address the above challenge, we propose *STUIOS* (Stochastic Testing with Unique Input Output Sequences), an approach for the automated generation of stochastic oracles that verify the capability of a software system to fulfill timing constraints in the presence of time uncertainty. Such stochastic oracles entail the statistical analysis of repeated test case executions based on test output probabilities predicted by means of statistical model checking. Results from two industrial case studies in the automotive domain demonstrate that this approach improves the fault detection effectiveness of tests suites derived from timed automata, compared to traditional approaches.

CCS Concepts: • **Software and its engineering → Software verification and validation**;

Additional Key Words and Phrases: Time Uncertainty, Probabilistic Unique Input Output Sequences, Timing Specifications, Test Oracles Generation

ACM Reference Format:
Chunhui Wang, Fabrizio Pastore, and Lionel Briand. 2018. Oracles for Testing Software Timeliness with Uncertainty. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 00 (2018), 30 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Software timeliness, i.e., the capability to fulfill timing constraints, plays a crucial role in embedded software systems. Timing constraints are used either to specify the period of system operations, a typical requirement of real-time systems, or to specify the system response time, a necessity when an embedded system provides safety critical functionalities.

Although embedded systems are designed to ensure that their operations work within pre-specified timing constraints, most of the timing specifications provided by software engineers are characterized by uncertainty [12]. An example of uncertainty is given by timing constraints

1049-331X/2018/0-ART00 $15.00
https://doi.org/0000001.0000001

expressed as ranges (e.g., the detection time of external events takes anywhere between 3400 and 5100 milliseconds).

Time uncertainty in software specifications can be due to multiple reasons, including design choices. A common example of such design choice is the polling of sensors at a certain frequency. This case is very common in embedded systems because many real-world events can only be detected by polling the environment in a control loop and, therefore, one cannot precisely determine when the system will be able to observe and process them. Example cases are the sensing systems developed by our industrial partner IEE [23], an automotive company located in Luxembourg, which provides the case study systems considered in this paper. The systems developed by IEE detect events such as overheating or the presence of hands on the steering wheel of a car. Time uncertainty may, in turn, lead to non-deterministic system behaviors that make the definition of test oracles challenging (e.g., determining whether it is acceptable for the system not to detect an error condition after a certain time delay).

The automated generation of oracles for non-deterministic systems where the source of non-determinism is time uncertainty has only been partially targeted by existing research. Indeed, most of the existing work on verifying the timing properties of software systems focuses on schedulability analysis for real-time systems [14]. These approaches address a different problem than ours, i.e., the development of techniques to test or verify that software tasks are able to meet their deadlines.

The few approaches that address the problem of testing software timeliness, in the presence of non-deterministic behaviors caused by time uncertainty, rely upon feedback-driven model-based test input generation [12, 19, 25]. These approaches generate test inputs during test case execution, based on the feedback from system execution. Models of the expected software behavior (e.g., timed automata) are used to identify system inputs and to validate system outputs. Unfortunately, feedback-driven approaches are often inappropriate for testing real-time systems because of the latency introduced by run-time processing of outputs and generation of test inputs, which comes in addition to the latency of test drivers and stubs.

An additional limitation of most existing timeliness testing approaches, including feedback-driven approaches, is that they do not fully address the problems related to time uncertainty. More specifically, they do not verify if the frequency of appearance of the expected outputs complies with the specifications, nor do they verify if the observed results can provide statistical guarantees for the validity of the test outcome. To the best of our knowledge, the only approach that deals with the effects of time uncertainty [20] focuses on the generation of complete test suites built to uncover specific types of faults, and cannot be adopted to generate oracles for arbitrary test suites.

In this work we assume that the timing constraints of the system are specified by means of timed automata since this is one of the most common formalisms adopted for this purpose [38]. Timed automata are a type of extended finite-state machines (EFSM) enabling the specification of guard conditions on clock variables and thus are adequate to define timing constraints. The type of uncertainty that we consider in this paper is common in industrial settings and relates to time-triggered state transitions (i.e., internal transitions that are enabled when an inequality constraint on clock variables evaluates to true), which are a typical cause of uncertainty [12]. For example this is the case when the arrival time of events is unknown or the duration of operations is uncertain.

Traditional approaches that derive test cases from finite state machine (FSM) specifications rely on Unique Input-Output (UIO) sequences to derive effective test oracles [15, 26]. Unfortunately, existing approaches for deriving UIO sequences from FSM and EFSM [31] do not work with timed automata and, more specifically, cannot deal with time-triggered state transitions. Furthermore, they do not verify if the frequency of appearance of the expected output over multiple executions of a same test case complies with the specifications, nor do they verify if the observed results can give statistical guarantees for the validity of the test outcome.

In this paper we propose Stochastic Testing with Unique Input-Output Sequences (*STUIOS*), an approach for the automated generation of effective oracles for timeliness test cases derived from timed automata, in the presence of time uncertainty. As main components of the approach, we introduce the concepts of *stochastic test cases* and *probabilistic UIO sequences* (PUIO sequences). A PUIO sequence is a unique input-output sequence with an associated probability of observing the given output sequence in response to the inputs. The underlying idea is that PUIO sequences enable fault detection by determining if the output sequences observed through testing are unlikely, based on multiple executions of the same test cases. Stochastic test cases extend the same idea to the entire test case. A stochastic test case specifies the expected probability of observing a specific output sequence given a test input sequence and, in addition, includes a PUIO sequence that is used to check if the test execution has brought the system to the expected state.

STUIOS receives as input the timing specifications of the system expressed as a network of timed automata, and a set of test cases derived from these specifications, either manually by the software engineers or automatically. *STUIOS* generates stochastic test cases that include the input-output sequences from the test cases provided by software engineers, and automatically derives corresponding expected probabilities of observing the test output and PUIO sequences.

One significant contribution and innovation is that *STUIOS* combines the principles of UIO techniques and statistical model checking to address the challenges stated above. *STUIOS* automatically identifies UIO sequences from timed automata by relying on the simulation and verification capabilities of the UPPAAL model checker. Furthermore, it derives probabilities for both the identified UIO sequences and the provided test cases by exploiting the features of statistical model checking software (the statistical extension of UPPAAL). Finally, *STUIOS* identifies failures by repeatedly executing test cases with the associated PUIO sequences and making use of statistical hypothesis testing to determine the test verdicts (pass or fail).

The paper proceeds as follows. Section 2 introduces the context and motivations of our work. Section 3 provides background information about the modeling and testing of timing specifications. Section 4 discusses the issues to expect when generating oracles from timed automata with uncertainty. Section 5 presents the *STUIOS* approach and its steps. Section 6 discusses the effectiveness of the approach when time uncertainty is not explicitly modeled in timed automata. Section 7 reports on the empirical results obtained from two industrial case studies in the automotive domain. Section 8 discusses related work. Section 9 concludes the paper.

2 MOTIVATION AND CONTEXT

The context of our research is that of embedded safety-critical systems (e.g., automotive controllers). An example is *Hands Off Detector* (*HOD* [22]), an embedded system developed by our industrial partner IEE, which we use as running example in the paper. *HOD* is an embedded system that detects if the car driver has both hands on the steering wheel; this device is used to enable and disable automatic braking in cars with driver-assist features to ensure safety (a car should not automatically brake if the driver does not have both his hands on the steering wheel). Another example case is *BodySense*^{TM} [21], the other case study system used in our empirical evaluation, which is a seat occupancy detection system; *BodySense*^{TM} is described in Section 7.

Embedded systems are typically implemented by means of a main control loop triggered with a constant frequency (e.g., every 1700 ms in the case of *HOD*). At the beginning of the control loop, the system executes the main system task, which performs all the safety critical operations of the system (e.g., error detection). Other functionalities (e.g., message handling) are implemented by lower priority tasks executed after the end of the main system task. The beginning of the main task is guaranteed to be triggered with a constant frequency by the underlying real-time operating system.

Embedded safety-critical systems are typically required to identify error conditions in a timely manner. Error conditions may alter the proper functioning of the system and, for this reason, companies developing embedded systems put particular attention to ensure that the system complies with the requirements specifications regarding the detection of error conditions (including detection time). As running example for this paper, we use timing properties concerning the qualification (i.e., confirmation) of temperature errors (e.g, overheating) in *HOD*. For example, the accuracy of *HOD* might be altered by overheating caused by a malfunctioning steering wheel heater. In the case of *HOD*, temperature error conditions are detected by polling data from a temperature sensor. More precisely, at every execution of the main task, *HOD* processes the value recorded by the temperature sensor to detect the presence of high or low temperature. A temperature error is qualified if it remains present in the system for at least two executions of the main control loop (i.e., 3400 ms), and it is similarly disqualified if it remains absent for at least two executions of the main control loop.

What complicates the testing of timing requirements in embedded systems is time uncertainty, which introduces non-deterministic software behavior. Time uncertainty may depend on multiple factors including polling operations, delays on the communication bus and multithreading. In the case of *HOD*, time uncertainty is due to the fact that polling operations are repeated according to a specified frequency (i.e., the duration of the main control loop) while error conditions (e.g., temperature errors in *HOD*) occur at unpredictable times during the loop execution. The effect of this uncertainty in *HOD* is that the time required to qualify temperature errors can be specified only in terms of a time range. More precisely, the timing requirements for *HOD* indicate that temperature errors should be qualified between 3400 ms and 5100 ms after they arise in the system while, ideally, one would like errors to be qualified in two cycles (i.e., 3400 ms after they arise).

Figure 1 shows an example of non-deterministic outputs during the execution of a test case that simulates overheating. Case 1 in Figure 1 shows that a temperature error is qualified in 3400 ms if the error condition (e.g., overheating) appears when the system executes the instructions that detect the error condition. Case 2 in Figure 1 shows that when the error condition appears just after error detection instructions, then the temperature error is only detected after 5100 ms. This happens because the error will be detected only in the next execution of the main control loop (i.e., after 1700 ms) and then it will be qualified after two executions of the main control loop (i.e., after 3400 ms). Case 3 in Figure 1, instead, shows that if the error condition appears 850 ms after error detection (i.e., in the middle point between two consecutive error detection operations), then a temperature error will be qualified in 4250 ms. This happens because the error will be detected only in the next execution of the main control loop (i.e, after 850 ms) and then it will be qualified after two executions of the main control loop.

In the presence of time uncertainty, software engineers need automated test oracles capable of determining if the non-deterministic outputs produced by test case executions are the result of time uncertainty or rather due to other factors such as software and hardware faults. For example, in the case of a test case that simulates overheating (Figure 1), engineers are interested in automatically determining if the non-deterministic outputs observed during multiple executions of a same test case follow the expected time distribution. Since time uncertainty depends on polling operations repeated with a constant frequency, the uncertainty is characterized by a uniform distribution (i.e., in Figure 1, Case 1 should occur as frequently as Case 2); if the observed *HOD* outputs are not uniformly distributed then the system does not behave as expected. In the context of embedded systems, variations in the frequency of the expected outputs might be triggered, for example, by configuration errors (e.g., number of execution cycles required to qualify a temperature error in

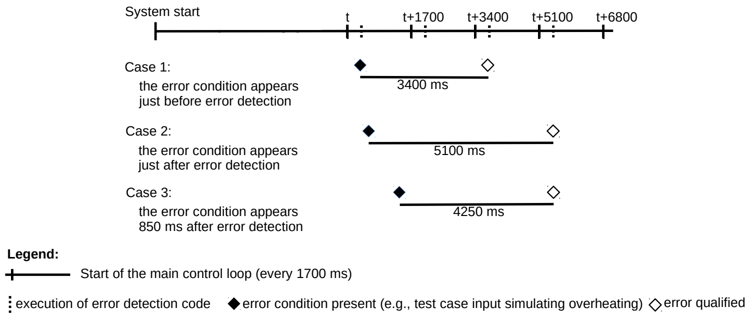

System start
t
t+1700
t+3400
t+5100
t+6800
Case 1:
the error condition appears just before error detection
3400 ms
Case 2:
the error condition appears just after error detection
5100 ms
Case 3:
the error condition appears 850 ms after error detection
4250 ms
Legend:
Start of the main control loop (every 1700 ms)
execution of error detection code
error condition present (e.g., test case input simulating overheating)
error qualified

Fig. 1. Effects of cyclic sampling on time uncertainty

HOD), unstable clocks, bus faults, or programming errors. The testing process should be capable of detecting these problems to ensure safety.

In this paper, we address the problem of generating test oracles that are robust to time uncertainty. The proposed solution works in contexts where timing requirements (including time uncertainty) are modelled by means of timed automata.

3 BACKGROUND

This section provides background information about the modeling of timing specifications with timed automata (TAs) and the modeling of test cases derived from such TAs.

3.1 Modelling of Timing Specifications

The timing properties of a system are typically modelled using networks of TAs or UML statecharts. In this paper, we make use of the former, and adopt the formalization from UPPAAL [8].

A timed automaton (TA) is a tuple $(L, l0, C, A, V, E, I)$, where L is a set of locations, $l0 \in L$ is the initial location, C is a set of clocks, A is a set of actions, V is a set of state variables, E is a set of edges between locations, I is a set of invariants assigned to locations. Each edge may have an action, a guard and a set of updates. Updates are expressed in form of assignments that can reset clocks or state variables. Each location might be associated with a state invariant that constrains clocks or state variables.

With a network of TAs, the state of the system is captured by the values of state variables and the set of active locations across all the TAs.

Actions are used to synchronize different TAs. Each action is expressed with the notation *event!* or *event?*. The notation *event!* indicates that the event is sent when the edge is fired, while the notation *event?* indicates that the edge is fired only if this event has been received from another automaton.

Figure 2 shows a simplified TA for *HOD*. The TA in Figure 2 captures the timing properties concerning the qualification (i.e., confirmation) of temperature errors (e.g, overheating). In the TA in Figure 2, the variable `x` is a clock variable, while `isQualified` and `qc` are two state variables. The former indicates if a temperature error is qualified while the latter counts the number of times temperature errors have qualified so far.

In Figure 2, the edge that connects locations `Absent` and `Present` can be fired only upon the reception of the event `tempOutOfRange`, which indicates that the temperature is out of the valid range (e.g., because of overheating). The event `tempOutOfRange` is an abstraction used to capture the fact that an error condition is present in the system.

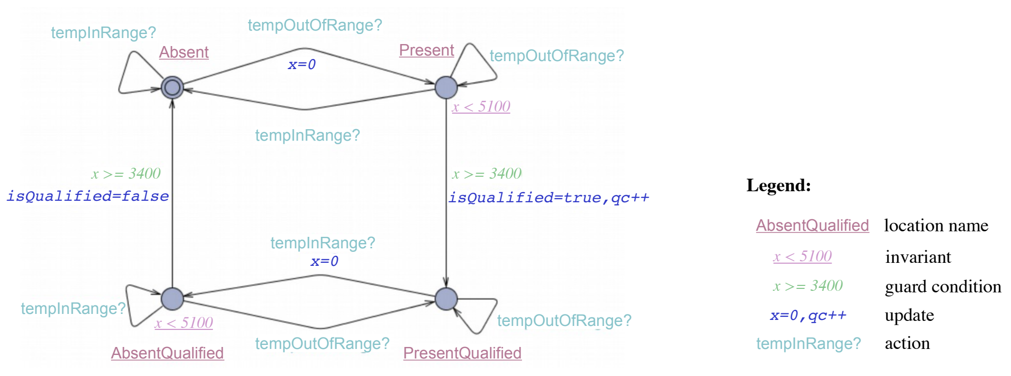


Fig. 2. Automaton that captures the timing requirements for the qualification of temperature errors in *HOD*.

The location `Present` has an invariant that indicates that the clock variable `x` must be below 5100 ms when the location is active. The guard condition on the edge between the location `Present` and `PresentQualified` indicates that this edge cannot be fired if the clock `x` is below 3400. In effect, the edge between locations `Present` and `PresentQualified` can be fired anytime when `x` has a value between 3400 and 5100. In other words, this edge models a *time-triggered state transition* whose time range is captured by the location invariant and the guard condition on the edge.

TAs are usually modeled by including time ranges to allow for time uncertainty. As mentioned before, in the case of *HOD*, time uncertainty is driven by the polling of the environment for data.

When time uncertainty depends on polling operations repeated with a constant frequency then the uncertainty is characterized by a uniform distribution. This is the case of *HOD*, where polling operations are guaranteed to be performed every 1700 ms since they are executed within the main task (see Section 2). Uncertainty regarding functionalities that are not executed with a constant frequency might not be characterized by a uniform distribution. For simplicity, though in the rest of the paper we focus on the former case, the proposed approach can be applied to systems characterized by different types of distributions, which can be modelled in TAs as indicated in Section 5.2.

3.2 Modelling of Test Cases

In the presence of software specifications elicited by using TAs or FSMs, software engineers derive test cases according to criteria whose objective is to ensure the compliance of the software with its specifications. A commonly adopted criterion, for example, is edge coverage, which consists in generating test cases so that all the edges of the timed automata are covered, at least once, during test execution.

We model test cases derived from TAs as sequences of input / output pairs. Each pair indicates the output expected to be generated by the system after a given input. In embedded systems, inputs and outputs typically correspond to system events (e.g., messages and interrupts). However, since TAs include state variables, the expected output indicated in a test case also includes the expected value of observable state variables. In addition, since edges in TAs might be triggered after specific time delays, inputs might be expressed as wait operations.

```
<tempOutOfRange>/ (isQualified == false, qc == 0)
<wait for 5100 ms> /  (isQualified == true , qc == 1)
```

Fig. 3. A Test Case that checks if TemperatureErrors are qualified on time. We use the symbol '/' to separate inputs from the expected outputs.

An example test case is shown in Figure 3. This test case is derived from the TA in Figure 2 and checks if temperature errors are qualified on time. This test case covers the edges of the shortest path from location `Absent` to location `PresentQualified`, and checks if state variables and outputs contain the expected result after each input or time delay. The first line of the test case indicates that the event `tempOutOfRange` has no visible effect on the system because this event simply indicates that an error condition is present (i.e, the temperature is out of the valid range) but we cannot know in advance when the system will detect it. As a consequence, the observable state variables `isQualified` and `qc` should remain set to `false` and `zero`, respectively. The second line of the test case indicates that after 5100 ms, the values of these two variables should be set to `true` and `one`, respectively. This line of the test case checks if the system is capable of qualifying an error in at most 5100 ms (which corresponds to the maximum allowed error qualification time).

3.3 Testing with UIO Sequences

The state of the actual software system that implements the timing specifications expressed in TAs is partially observable. More specifically, not all the state variables used to implement the actual system can be inspected during testing. Unobservable variables capture state information that in the TAs is modelled thanks to the use of locations. This limited observability makes oracles that simply check if the expected output sequence has been generated by test inputs ineffective. More precisely, these oracles may not detect transition faults [29, 39], which manifest themselves when the active location of a TA following a state transition is incorrect but the system updates its state variables and generates outputs as expected.

In the case of *HOD*, a transition fault may keep the system in location `Present` for more than the allowed time (5100 ms). In the presence of this transition fault, *HOD* would behave as if the edge between locations `Present` and `PresentQualified` was turned into a self-loop that begins and ends in location `Present`. This fault cannot be detected with the test case in Figure 3 which simply checks the values of the observable state variables `isQualified` and `qc` after waiting for 5100 ms, since these two variables are updated as expected.

When specifications are given in the form of FSMs, common approaches for testing in the presence of partial observability are based on the identification of Unique Input-Output (UIO) sequences [15, 26]. A UIO sequence is composed of a sequence of inputs and corresponding outputs generated by the system. It provides the guarantee that the expected output sequence is observed only when the system is in a specific known state. An oracle, derived from a UIO sequence, is executed after a test case by checking if the system is in the expected state. To do so, inputs specified by the UIO sequence are sent and the system response is compared to the expected output sequence that contains the values assigned to the observable state variables. We model UIO sequences in the same way as test cases, i.e., by using sequences of IO pairs that include time delays and observable state variables. Details about the generation of UIO sequences for TAs are provided in Section 4.

4 TESTING TIMED AUTOMATA WITH TIME UNCERTAINTY: OPEN ISSUES

We now discuss the challenges to consider in order to derive reliable oracles from TAs in presence of time uncertainty.

```
<wait for 5100ms> / ( isQualified == true, qc == 1 )
<tempInRange> / ( isQualified == true, qc == 1)
<wait for 5100ms> / ( isQualified == false, qc == 1 )
```

Fig. 4. UIO sequence derived for the test case in Figure 3.

Table 1. Output generated by the UIO sequence in Figure 4 when the location reached after the execution of the test case in Figure 3 is different than PresentQualified. Bold red values highlight the differences with the expected output; the symbol '-' is used when a previous input in the sequence leads to an output different than the one expected by the UIO (i.e., to detect a transition fault).

Input	Absent	Present	AbsentQualified
`<wait for 5100ms>`	`isQualified == true, qc == 1`	`isQualified == true, qc ==` **2**	`isQualified ==` **false**`, qc == 1`
`<tempInRange>`	`isQualified == true, qc == 1`	-	-
`<wait for 5100ms>`	`isQualified ==` **true**`, qc == 1`	-	-

4.1 Generation of UIO sequences from TAs

Since TAs include state variables, the output generated by a TA while processing a certain input sequence depends on both the active location and on the values assigned to state variables. UIO sequences derived from TAs are thus context dependent, i.e., they characterize a state of the system, not just a location, and their output depends on the test inputs already processed [30]. This characteristic is shared with UIO sequences derived from FSMs including state variables (e.g., EFSM). As a result, we generate a different UIO sequence for each test case of the test suite.

Figure 4 shows a UIO sequence derived for the test case of Figure 3 that checks if the system has reached location PresentQualified after test execution. The first line of the UIO sequence indicates to wait for 5100 ms, after which the system should keep location PresentQualified active (this cannot be observed by a tester), and keep the variables isQualified and qc set to true and one, respectively. The second line of the UIO sequence indicates to remove the error condition from the system (e.g., by resetting the temperature of the test environment within the valid range). Since, after this operation, the system should not make any change to any observable state variable, the values of the variables isQualified and qc are expected to remain set to true and one. The third line of the UIO sequence indicates to wait for 5100 ms, after which the system should make location Absent active (this cannot be observed by a tester), set the variables isQualified to false while variable qc should remain set to one.

The UIO sequence in Figure 4 ensures that the execution of the test case terminates in location PresentQualified. To clarify this better, we show in Table 1 the output generated by the system when the inputs of the UIO sequence are processed in locations different than PresentQualified. The input on the first line of the UIO sequence in Figure 4 leads to an output different than the expected one when the active location is Present or AbsentQualified (see first row of Table 1). The input on the third line of the UIO sequence, instead, helps determine if the test erroneously terminated in location Absent (the generated output is different from the expected one, as shown in the last row of Table 1).

Unfortunately, there are no approaches focused on deriving UIO sequences from TAs characterised by time uncertainty. In the case of TAs without uncertainty (e.g., TAs that do not include any time-triggered transition), existing approaches for generating UIO sequences from EFSMs could be applied because of the similarity between EFSM and TA formalisms. For example, the approach proposed by Robinson et al. [31] requires deterministic EFSMs with transitions that are always triggered by some input event.

```
<tempOutOfRange>/ (isQualified == false, qc == 0)
<wait for 4250 ms> /  (isQualified == true , qc == 1)
<tempInRange>/ (isQualified == true , qc == 1)
<wait for 5100 ms> /  (isQualified == false, qc == 1)
```

Fig. 5. Test case that verifies the qualification of a temperature error that appears for a short time. The bold line matches the UIO sequence that checks if the test case brings the system to location `AbsentQualified`.

4.2 Soundness and effectiveness of test oracles

One of the effects of time uncertainty is non-determinism, which might lead to ineffective and unsound test oracles.

Test oracles are ineffective when they are not capable of discovering the presence of a failure. Consider for example a faulty implementation of *HOD* that may take up to 6000 ms to qualify an error. In this case, the test case and the UIO sequence in Figures 3 and 4 will be able to detect the fault only when *HOD* takes more than 5100 ms to qualify an error during test execution. If the fault manifests itself in a non-deterministic way, a single test case execution may not be able to detect its presence. To address this problem, when testing timing requirements, multiple test case executions might thus be needed, and test oracles should therefore provide a verdict (i.e., pass or fail) only when there are statistical guarantees about the accuracy of results.

Software testing is unsound if test oracles can flag valid software as faulty. In the presence of non-determinism caused by time uncertainty, a same test input sequence may lead to multiple valid output sequences and bring the system to multiple valid locations. For example, in the case of *HOD*, this happens with the test case shown in Figure 5, which checks if the system behaves properly when a temperature error appears just for a short time. More precisely, this test case checks if the system can qualify a temperature error that remains active just for 4250 ms. If *HOD* does not qualify the temperature error between 3400 ms and 4250 ms, the test case in Figure 5 will fail when checking the condition `isQualified==true, qc==1`. A single failure of this test case does not indicate the presence of a fault in *HOD* since the specifications indicate that *HOD* can take more than 4250 ms to qualify an error.

To correctly evaluate test results, we thus need to know the expected probability for the system to respond with a specific output sequence. In the presence of time uncertainty, an implementation is faulty if and only if, over multiple executions, the various possible outputs are not observed with their expected frequency. In the context of embedded systems, slight variations in the frequency of the expected outputs should be detected (see Section 2). This observation shows the need for oracles to take into consideration the probability of observing a given output to determine if a test case passes or fails. We will refer to such test cases as *stochastic test cases* in this work.

To compute the expected frequency of a test case output, additional information about the distribution of time-triggered transitions might be needed. In the case of *HOD*, for example, time triggered transitions are fired following a uniform distribution because temperature events can happen at any time during the polling loop execution, with equal probability. For this reason we expect that the test case in Figure 5 should pass in almost 50% of the executions since 4250 is the middle point between 3400 and 5100.

4.3 Scalability of testing

Verifying that the system generates all possible, valid outputs and has reached all the expected states after processing the same test inputs, might easily lead to scalability issues. This may be the case if each test case needs to be executed multiple times, until every possible location is reached, including the least probable ones.

To deal with this problem, and thus balance testing cost and test effectiveness, we assume that engineers are interested in verifying if the system has been able to reach a *single* specific location, with the expected frequency, that we refer to as *desired final test location*, which is the location that one would ideally like to reach all the time but cannot because of non-determinism. Nevertheless, it is the location that is reached with the highest probability, according to the specifications.

In the case of the test case in Figure 5, for example, location `AbsentQualified` is the location that is most likely reached based on the sequence of outputs specified by the test case. Location `AbsentQualified` is thus the location to be verified by means of an UIO sequence (in bold in Figure 5).

5 THE *STUIOS* APPROACH

5.1 An Overview

Figure 6 shows an overview of the *STUIOS* approach. *STUIOS* works in two phases: (1) it *generates stochastic test cases* by extending the test cases provided by software engineers; (2) it *executes the stochastic test cases* and attempts to determine whether they pass or fail.

STUIOS receives as input a test suite, which we assume to be derived from specifications elicited as a network of TAs. Any test criterion can be used to generate the test suite. We simply expect that each test case of the test suite is defined as an IO sequence (i.e., a sequence of pairs <*input*> / <*expected output*>), which matches a feasible path of the automata, as described in Section 3.2. An example test case is shown in Figure 3. For each test case, we also expect that software engineers specify the *desired final test location.*

STUIOS wraps the given test cases into stochastic test cases that handle time uncertainty. We define a stochastic test case as a test case that includes an IO sequence that covers a specific test scenario, followed by a probabilistic UIO sequence (PUIO sequence) that checks if the desired final test location has been reached. A stochastic test case specifies both the probability of observing the test scenario output and the probability of observing the output of the PUIO sequence. The former is necessary to deal with the effects of time uncertainty on the test case execution. Indeed, in the presence of non-determinism caused by time uncertainty, a same input sequence may lead to multiple, valid, output sequences. The probability of observing the output of the PUIO sequence addresses the effects of time uncertainty on the output generated by the UIO sequence.

To *generate stochastic test cases*, *STUIOS* performs four steps (shown in Figure 6). In *Step 1.1*, *STUIOS* identifies a UIO sequence for each desired final test location of each test case provided by software engineers. In *Step 1.2*, *STUIOS* builds corresponding PUIO sequences, which means that it calculates, for each UIO sequence, the probability of observing its output when the UIO sequence is executed after the corresponding test case. In *Step 1.3*, *STUIOS* calculates the probability of observing the output expected for each test case. Steps 1.1 to 1.3 rely on the simulation and verification capabilities of the UPPAAL model checker [8]. We use confidence intervals to capture the probability of observing the test case output and the UIO output since this is how UPPAAL calculates them (see Section 5.2 for details). For example, the probability of observing the output of the test case T1 in Figure 6 is between 0.46 and 0.56 with 95% confidence level.

In *Step 1.4*, *STUIOS* builds stochastic test cases. Each generated test case specifies the probability of observing the output sequence, and includes the PUIO sequence that checks if the desired final test location has been reached.

In the second phase (*Execute Stochastic Test Cases* in Figure 6), *STUIOS* executes each stochastic test case against the software implementation multiple times (until we have sufficient statistical guarantees about the test outcome, as discussed below) and collects test results. *STUIOS* relies upon statistical hypothesis testing to decide when an accurate test result can be obtained from multiple executions of the same test case. Test execution is stopped when we have sufficient statistical guarantees that the test case either passes or fails. More specifically, a test case fails if its output or the PUIO output is not as frequent as expected, considering a confidence interval, or otherwise passes.

After a brief overview of the capabilities of UPPAAL exploited by *STUIOS*, the following sections provide additional details about the activities performed by *STUIOS*.

5.2 Probability Estimation with UPPAAL

To generate PUIO sequences (Steps 1.1 and 1.2 in Figure 6) and calculate test case probabilities (Steps 1.3), *STUIOS* relies on UPPAAL [8], which is a model checker for networks of TAs that provides model checking and statistical model checking capabilities. More specifically, *STUIOS* makes use of the verification of reachability properties, the simulation of the model execution, and the estimation of the true probability of a model property (i.e., the probability that a property holds).

We briefly describe the probability estimation feature of UPPAAL as some readers may not be familiar with it. UPPAAL can estimate the probability that a given property of the system holds within a certain time frame. More precisely, UPPAAL returns a confidence interval; for example UPPAAL can estimate that the probability that a specific location of a network of TAs be reached in 4250 clock ticks is between 0.46 and 0.56.

A confidence interval indicates that the estimated parameter (e.g., the frequency of reaching a given location) has a probability p_c of lying in it (p_c is known as the *confidence level*). We used $p_c = 95\%$ in our experiments. To estimate such intervals, UPPAAL relies upon an algorithm that resembles Monte-Carlo simulation [11]. More precisely, given a time frame specified by the end-user, UPPAAL simulates the triggering of the TA transitions and then determines if the property holds (e.g., by checking that a location has been reached within the given time frame). UPPAAL performs multiple simulation runs sequentially and, after each run, computes the confidence interval using the Clopper-Pearson 'exact' method [10]. The simulation terminates when the interval reaches a length specified by the engineer (in our experiments, we used 0.1 by following common practice [11]).

UPPAAL assumes that the values of the clock variables that control time triggered transitions follow a uniform distribution within the range specified in the TA (e.g., 3400 and 5100 in the case of location `Present` in Figure 2). This assumption holds for *HOD*, and in general for systems with similar characteristics, i.e., embedded controllers that poll the environment for inputs, where we have no information about the arrival time of inputs. However, TAs produced with UPPAAL can be modeled in such a way that time triggered transitions might follow any distribution; this is achieved by defining custom functions to control the delay in transitioning to locations with invariants including clock variables (see [11] for more details).

5.3 Generating Stochastic Test Cases

We now describe the steps followed by *STUIOS* to generate stochastic test cases (Steps 1.1, 1.2, 1.3, and 1.4 in Figure 6). Figure 7 shows these activities in an algorithmic form.

5.3.1 Identify UIO Sequences (Step 1.1). To generate UIO sequences, *STUIOS* performs three activities: (1) it determines the context (active locations and values of state variables) of each UIO sequence, (2) it identifies all the potential UIO sequences for a given final test execution state, (3) it identifies a valid UIO sequence among these potential UIO sequences (i.e., it ensures that the UIO sequence generates a unique output to detect transition faults).

Activity 1 of Step 1.1-Determining the context of a UIO sequence. UIO sequences for TAs are context dependent (Section 4.1). Since *STUIOS* uses UIO sequences to determine if the execution of a test case has brought the system into a given location, the context of a UIO sequence consists of the state of the system after the execution of the corresponding test case (hereafter final test state). In TAs, the state of the system is represented by both the active locations and the values of the state variables. Though the final test state can be deduced by simulating test case executions against the TA network, *STUIOS* relies on reachability analysis to speed up the process.

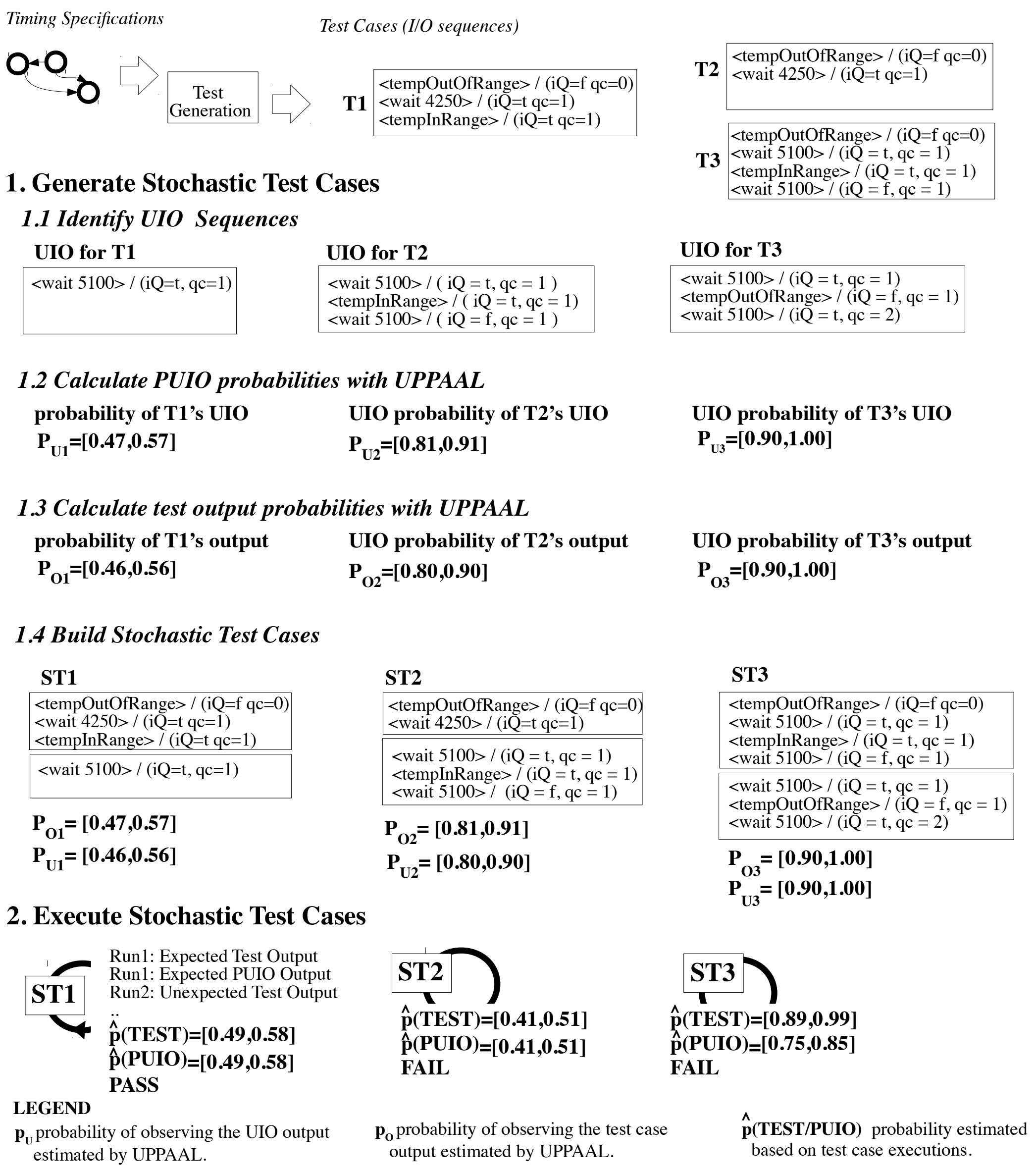

Timing Specifications
Test Generation
Test Cases (I/O sequences)
T1
<tempOutOfRange> / (iQ=f qc=0)
<wait 4250> / (iQ=t qc=1)
<tempInRange> / (iQ=t qc=1)
T2
<tempOutOfRange> / (iQ=f qc=0)
<wait 4250> / (iQ=t qc=1)
T3
<tempOutOfRange> / (iQ=f qc=0)
<wait 5100> / (iQ = t, qc = 1)
<tempInRange> / (iQ = t, qc = 1)
<wait 5100> / (iQ = f, qc = 1)
1. Generate Stochastic Test Cases
1.1 Identify UIO Sequences
UIO for T1
<wait 5100> / (iQ=t, qc=1)
UIO for T2
<wait 5100> / (iQ = t, qc = 1)
<tempInRange> / (iQ = t, qc = 1)
<wait 5100> / (iQ = f, qc = 1)
UIO for T3
<wait 5100> / (iQ = t, qc = 1)
<tempOutOfRange> / (iQ = f, qc = 1)
<wait 5100> / (iQ = t, qc = 2)
1.2 Calculate PUIO probabilities with UPPAAL
probability of T1's UIO
P_{U1}=[0.47,0.57]
UIO probability of T2's UIO
P_{U2}=[0.81,0.91]
UIO probability of T3's UIO
P_{U3}=[0.90,1.00]
1.3 Calculate test output probabilities with UPPAAL
probability of T1's output
P_{O1}=[0.46,0.56]
UIO probability of T2's output
P_{O2}=[0.80,0.90]
UIO probability of T3's output
P_{O3}=[0.90,1.00]
1.4 Build Stochastic Test Cases
ST1
<tempOutOfRange> / (iQ=f qc=0)
<wait 4250> / (iQ=t qc=1)
<tempInRange> / (iQ=t qc=1)
<wait 5100> / (iQ=t, qc=1)
P_{O1}= [0.47,0.57]
P_{U1}= [0.46,0.56]
ST2
<tempOutOfRange> / (iQ=f qc=0)
<wait 4250> / (iQ=t qc=1)
<wait 5100> / (iQ = t, qc = 1)
<tempInRange> / (iQ = t, qc = 1)
<wait 5100> / (iQ = f, qc = 1)
P_{O2}= [0.81,0.91]
P_{U2}= [0.80,0.90]
ST3
<tempOutOfRange> / (iQ=f qc=0)
<wait 5100> / (iQ = t, qc = 1)
<tempInRange> / (iQ = t, qc = 1)
<wait 5100> / (iQ = f, qc = 1)
<wait 5100> / (iQ = t, qc = 1)
<tempOutOfRange> / (iQ = f, qc = 1)
<wait 5100> / (iQ = t, qc = 2)
P_{O3}= [0.90,1.00]
P_{U3}= [0.90,1.00]
2. Execute Stochastic Test Cases
ST1
Run1: Expected Test Output
Run1: Expected PUIO Output
Run2: Unexpected Test Output
..
$\hat{p}$(TEST)=[0.49,0.58]
$\hat{p}$(PUIO)=[0.49,0.58]
PASS
ST2
$\hat{p}$(TEST)=[0.41,0.51]
$\hat{p}$(PUIO)=[0.41,0.51]
FAIL
ST3
$\hat{p}$(TEST)=[0.89,0.99]
$\hat{p}$(PUIO)=[0.75,0.85]
FAIL
LEGEND
P_U probability of observing the UIO output estimated by UPPAAL.
P_O probability of observing the test case output estimated by UPPAAL.
$\hat{p}$(TEST/PUIO) probability estimated based on test case executions.

Fig. 6. An overview of the *STUIOS* approach. The values in the figure are based on the running example presented in the paper.

STUIOS first translates the IO sequence into a sequential timed automaton (hereafter test automaton, see Line 9 in Figure 7). In the test automaton, each edge either sends a test input or checks if the output corresponds to expectations. If the output does not match expectations, an error location is reached, otherwise the computation continues. Figure 8 shows a test automaton derived from the test case in Figure 3. The edge between locations *L1* and *L2* captures the sending of the event `tempOutOfRange`, the edge between locations *L2* and *L3* simulates a delay of 5100 ms, while the other edge checks for the expected values of observable variables, and activates the location `Error` in case the values are not the expected ones.

STUIOS then exploits the reachability analysis feature of UPPAAL to identify a trace that shows the outputs of the test case execution (Line 25 in Figure 7). This is done thanks to a reachability formula that specifies that the desired final test location (`PresentQualified` in Figure 2) should be reached once the test case execution is complete (i.e., we have reached `TestCompleted` in Figure 8).

The trace generated by UPPAAL shows the value of all the state variables and the list of active locations after the execution of the test case, such information being extracted by function *extractContext* in Figure 7, Line 26.

```
Require: $TS$, the test suite
Require: $TAS$, the timing specifications (network of TA)
Require: $DTL$, the desired final test location
Require: $L$, the maximum length of a UIO sequence
Ensure: $STS$, a stochastic test suite derived from TS
 1: function MAIN(TS,TAS,DTL,L)
 2:   for $testCase$ in $TS$ do
 3:     $STS \leftarrow STS \cup buildStochasticTest(testCase)$
 4:   end for
 5:   return $STS$
 6: end function

 7: function BUILDSTOCHASTICTEST($testCase$)
 8:   //1.1 Identify an UIO
 9:   $TA_{test} \leftarrow generateTestAutomaton(testCase)$
10:   $PUIO, TA_{UIO} \leftarrow identifyUIO(TA_{test})$
11:   //1.2 Calculate PUIO probability if a PUIO was found
12:   if $PUIO \neq null$ then
13:     $TA_{testUIO} \leftarrow join(TA_{test}, TA_{UIO})$
14:     $maxTime \leftarrow sumAllTheDelaysIn(TA_{testUIO})$
15:     $p_{PUIO} \leftarrow probabilityAnalysis(TAS, TA_{testUIO}, maxTime)$
16:   end if
17:   //1.3 Calculate test output probability
18:   $maxTime \leftarrow sumAllTheDelaysIn(TA_{test})$
19:   $p_{test} \leftarrow probabilityAnalysis(TAS, TA_{test}, maxTime)$
20:   //1.4 Return stochastic test case
21:   return $< testCase, p_{test}, PUIO, p_{PUIO} >$
22: end function

23: function IDENTIFYUIO($TA_{test}$)
24:   //1.1.1 Determine the context of the UIO
25:   $trace \leftarrow reachability(TAS, TA_{test}, DTL)$
26:   $assignments, activeLocations \leftarrow extractContext(trace)$
27:   $pathLen \leftarrow 1$
28:   repeat
29:     //1.1.2 Identify all the potential UIO sequences
30:     $potentialUIOS \leftarrow findPUIOS(TAS, assignments,$
31:                                 $activeLocations, pathLen)$
32:     for $PUIO$ in $potentialUIOS$ do
33:       //1.1.3 Determine if the potential UIO is valid
34:       $TA_{UIO} \leftarrow generateTestAutomaton(PUIO)$
35:       if $isValidUIO(activeLocations, TA_{UIO}, assignments)$
36:           return $PUIO, TA_{UIO}$
37:     end for
38:     $pathLen \leftarrow pathLen + 1$
39:   until $pathLen < L$
40:   return $null, null$
41: end function

42: function ISVALIDUIO($activeLocations, TA_{UIO}, assignments$)
43:   $locations \leftarrow activeLocations \setminus DTL$
44:   $TA_{DTL} \leftarrow retrieveAutomatonContaining(DTL)$
45:   $otherLocations \leftarrow allLocationsIn(TA_{DTL}) \setminus DTL$
46:   for $oL$ in $otherLocations$ do
47:     $activeLocations \leftarrow locations \cup oL$
48:     $trace \leftarrow reachability(TAS, TA_{UIO}, activeLocations,$
49:                                 $assignments)$
50:     if $trace \neq null$
51:           //final state of $TA_{UIO}$ is reachable from $activeLocations$
52:           return $FALSE$
53:   end for
54:   return $TRUE$
55: end function
```

Function *reachability* performs reachability analysis with UPPAAL by processing the timing specifications and the test automaton. The additional parameters appearing in line 48 are used to set the initial state of the system (otherwise the initial state is specified in the software specifications). Function *probabilityAnalysis* returns the probability of reaching the final state of the given test automaton in the provided time bound.

Fig. 7. The algorithm to generate stochastic test cases.

Activity 2 of Step 1.1-Identifying all the potential UIO sequences. To identify all the potential UIO sequences of a specific length, *STUIOS* explores all the feasible paths of the timed automata by taking advantage of the UPPAAL simulator. To guarantee termination, the exploration is performed up to a path length L provided by the software engineer.

The exploration is performed in a breadth-first manner starting from the final test state and is implemented by function *findPUIOS* invoked in Line 30 in Figure 7. The paths traversed during the visit are *potential UIO sequences* since they capture the output sequence expected following additional inputs being sent to the system after test case execution, but do not give any guarantee about uniqueness.

To determine the maximum length L of UIO sequences, engineers can rely on the average UIO length reported in related work [15]. For example, considering that the average length for models

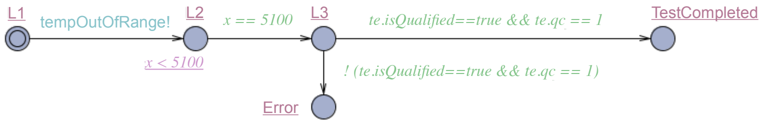

L1
tempOutOfRange!
L2
x < 5100
x == 5100
L3
te.isQualified==true && te.qc == 1
TestCompleted
! (te.isQualified==true && te.qc == 1)
Error

Fig. 8. Test Automaton derived from the test case in Figure 3.

with at most 20 locations is reported to be three [15], and that every TA for *HOD* includes less than 20 locations, we assumed a maximum length L of 10 to increase our chances to discover all required UIO sequences.

The process for finding potential UIO sequences starts with a path length equal to *one* (Line 27 in Figure 7) and then the process iterates by incrementing the path length by one (Line 38) if none of the recorded sequences is recognized as a valid UIO sequence, as explained in the following section.

Activity 3 of Step 1.1-Identification of valid UIO sequences. A valid UIO sequence guarantees that the expected output sequence is observed if and only if the test case has terminated in the desired final test location.

STUIOS identifies a valid UIO sequence by iteratively processing all the potential UIO sequences to find a potential UIO sequence whose inputs never lead to the expected output sequence when the active location is different from the desired final test location.

First *STUIOS* translates each potential UIO sequence into a sequential automaton (hereafter UIO automaton) by following the same process adopted to derive test automata (Line 34 in Figure 7).

Then *STUIOS* relies on the reachability analysis feature of UPPAAL to determine if the UIO sequence can generate the expected output even when the test case terminates in a location different than the desired final test location. This is done by function *isValidUIO*, used in Line 35, which starts reachability analysis from the same active locations and variable values as those of the test termination state, except for the desired final test location. *STUIOS* iterates the process multiple times so that all the locations other than the final test location are considered (see lines 46 to 53).

STUIOS identifies a UIO sequence when the final location of the test automaton is never reached (Line 54). *STUIOS* repeats the process for all the potential UIO sequences until a valid UIO sequence is found (see the `return` instruction in Line 36). Although the proposed algorithm may process a vast number of potentially long UIO sequences, which may affect the scalability of the approach, in practice, *STUIOS* scales well since UIO sequences are usually short [15] and thus the identification of UIO sequences usually terminates after a few iterations.

5.3.2 *Building PUIO Sequences (Step 1.2).*

In the next step (Step 1.2 in Figure 6), *STUIOS* associates to each UIO sequence a confidence interval that captures the probability that the system responds with the expected output of the UIO sequence if the desired final test location was reached (Lines 14 to 15 in Figure 7). As discussed earlier, to calculate this probability, *STUIOS* uses the probability estimation feature of UPPAAL.

To compute the probability of a UIO sequence, *STUIOS* first builds a test automaton that joins the sequence of inputs produced by the test case and by the UIO sequence (Line 13) and then estimates the probability of reaching the final location of the test automaton (Line 15). The time bound considered to compute the probability should match the test case execution time. In our context this is given by the sum of all the delays appearing in the combined automaton, i.e., variable `maxTime` in Line 14.

5.3.3 *Identifying Test Case Probabilities (Step 1.3).*

STUIOS computes the probability of observing the output expected by the test case (Step 1.3 in Figure 6) by following the same approach described

```
(isQualified == false, qc == 0)
 (isQualified == true, qc == 0)
```

Fig. 9. Portion of a faulty output sequence generated after the execution of the test case in Figure 5.

in Section 5.3.2 (i.e., by computing the probability of reaching the final location of the test automaton, which does not include the PUIO sequence, see Lines 18 to 19 in Figure 7).

5.3.4 Building Stochastic Test Cases (Step 1.4). A stochastic test case is then built (Step 1.4 in Figure 6) using the PUIO sequences and the probability of observing the expected test output (Line 21).

5.4 Test Execution

STUIOS executes each stochastic test case in a loop until a stopping condition is met (Step 2 in Figure 6). We have identified three different stopping conditions: (1) the actual output is illegal (in this case, the test case fails), (2) the expected output occurs with the expected frequency (in this case, the test case passes), (3) the expected output does not occur with the expected frequency (in this case too, the test case fails). Below, we describe the three cases in more detail.

Case 1: The output is illegal. The output of a test case is illegal if the observed output sequence cannot be generated by the network of TAs that capture the software specifications. In other terms, there is no path in the TAs that leads to the observed output sequence after the given input sequence.

The output sequence in Figure 9 shows an example of an illegal output for the TA in Figure 2. This output sequence is illegal if it is generated in response to the test inputs of the test case in Figure 5. The specifications for *HOD* indicate that the system, after waiting for 4250 ms (second line of the test case), is allowed to either stay in location `Present`, thus keeping the variables `isQualified` and `qc` set to `false` and `zero`, respectively, or move to location `PresentQualified`, and set the values of the two variables to `true` and `one`, respectively (these are the values expected by the test case Figure 5). The faulty output in Figure 9 shows that the system sets the value of variable `isQualified` to `true` but keeps variable `qc` set to `zero` after waiting for 4250 ms. This combination of values is not a legal output for the TA in Figure 2.

STUIOS thus stops executing a test case if it leads to an invalid output sequence because this clearly indicates that the software does not meet its specifications. The test case is considered to have failed.

Cases 2 and 3: The output is valid. If the system produces legal outputs (i.e., an output that can be generated according to the software specifications), then we need to determine if, across multiple executions of the stochastic test case, the system generates, among all the legal output sequences generated, the desired output sequence with the expected frequency.

Since stochastic test cases include both test cases provided by software engineers and PUIO sequences, we need to observe that both the following output sequences occur with the expected frequency: (1) the sequence of output values (hereafter $output_{TEST}$) generated after the inputs belonging to the original test case and (2) the output sequence (hereafter $output_{PUIO}$) generated after the inputs of the PUIO sequence.

We assume that each execution of a test case is independent; this assumption generally holds for stateless systems and for systems that provide means to reset the system state between the executions of different test cases. The latter is true for *HOD* and usually for embedded systems in general. Since each execution of a test case is independent and since the result is either pass or fail, we can assume that test results follow a binomial distribution.

Given that the expected frequencies of $output_{TEST}$ and $output_{PUIO}$ are captured by means of confidence intervals returned by UPPAAL, we perform hypothesis testing based on confidence interval comparison [28].

To compute a confidence interval that captures the frequency of occurrence of the expected test output, we rely upon the Wilson score interval [41], which is known to be reliable even in the presence of a small sample (i.e., executions of a test case in our context), when it follows a binomial distribution. The Wilson score interval is computed according to the formula:

$$\frac{1}{n+z^2}\left[n_S + \frac{1}{2}z^2 \pm z\sqrt{\frac{1}{n} * n_S * n_F + \frac{1}{4}z^2}\right]$$

where z is known as standard score, and depends on the confidence level p_c (if p_c=95% then z=1.96), n is the total number of runs considered, n_S is the number of successes (in our context, the output of the test case is the expected one), n_F is the number of failures (the output of the test case is not the expected one). The value of p_c is chosen by the experimenter (e.g., the software engineer) and captures the desired likelihood of avoiding a Type I error (i.e., erroneously rejecting the null hypothesis). We set p_c to 95%.

A test case fails if the null hypothesis "the expected output occurs with the expected frequency" is rejected. Since we estimate the expected frequency of an output using the confidence interval returned by UPPAAL and we estimate the frequency of the output generated by the actual system using also a confidence interval, we can rely on interval comparison to perform hypothesis testing. The two intervals consist in a range indicating that the estimated parameter should lie in it, at a level of confidence p_c. Thus, if two intervals do not overlap, there is convincing evidence of a difference (i.e., the implemented system differs from the model) [28]. Following this principle, we reject the null hypothesis if the interval computed by UPPAAL and the Wilson score interval computed during testing do not overlap. Confidence interval comparison is known to be more conservative (i.e., rejects the null hypothesis less often) than other methods when the null hypothesis is true, which in our case means fewer false alarms [33].

To provide trustable results we would like to collect a large enough sample to be confident about the outcome. Generally speaking, the more observations, the higher the confidence about the absence of faults. We rely on the length of the confidence interval. With a small interval, when the null hypothesis is not rejected, we can conclude that the estimated parameter is close to the value derived from specifications with a given degree of confidence ($p_c = 95\%$ in our example). Based on this observation we stop testing when the confidence interval is small enough; more specifically, we have chosen to adopt the same criterion adopted by UPPAAL, which means that the length of the confidence interval generated by *STUIOS* is at most $2 * (1 - p_c)$, 0.1 in our example. This way the two compared intervals have approximately the same length.

STUIOS applies hypothesis testing twice for each test case execution, considering both $output_{TEST}$ and $output_{PUIO}$. When the confidence interval is smaller than 0.1 and the null hypothesis is supported for both $output_{TEST}$ and $output_{PUIO}$, then *STUIOS* indicates that the test case has passed. When the confidence interval is smaller than 0.1 and the null hypothesis is rejected in the case of either $output_{TEST}$ or $output_{PUIO}$, then *STUIOS* indicates that the test case has failed. Otherwise, if the confidence interval is not smaller than 0.1, the execution of the test case continues. According to the Wilson score formula, the size of the confidence interval depends on the formula $\frac{z\sqrt{\frac{1}{n} * n_S * n_F + \frac{1}{4}z^2}}{n+z^2}$, which identifies the lower and upper bounds of the interval; thus, the maximum number of executions (max_E) required to obtain a confidence interval below a given value has an upper limit, for a specific value of z. For example, this upper limit is equal to 381 when we compute a confidence interval of length 0.1 with $p_c = 95\%$.

To speed up testing, *STUIOS* enables engineers to specify a value for max_E based on their test budget (we used $max_E = 100$ in our experiment). When the maximum number of executions (max_E)

is reached, *STUIOS* reports that the test case passes if the null hypothesis is confirmed for both $output_{TEST}$ and $output_{PUIO}$. Otherwise the test case fails. When the confidence interval length is still above 0.1, *STUIOS* also reports that the test case outcome might not be reliable. An in-depth analysis of the effects of max_E on the effectiveness of oracles is reported in Section 7.

For example, test case T1 in Figure 6 passes because (1) the Wilson score interval calculated for $output_{TEST}$ after multiple runs of T1 is $[0.49, 0.58]$, which is smaller than 0.1 and overlaps with the confidence interval provided by UPPAAL ($[0.46, 0.56]$), and (2) the Wilson score interval calculated for $output_{PUIO}$ overlaps with the expected interval. In contrast, test case T2 fails because the Wilson score intervals calculated for both $output_{TEST}$ and $output_{PUIO}$ do not overlap with the intervals associated to the stochastic test case. Finally, test case T3 fails because the Wilson score interval calculated for $output_{PUIO}$ ($[0.75, 0.85]$) does not overlap with the expected interval ($[0.9, 1.0]$).

6 IMPLICIT TIME UNCERTAINTY, NON-DETERMINISTIC TRANSITIONS AND UIO SEQUENCES AVAILABILITY

In this section we discuss how implicit time uncertainty, non-deterministic transitions and availability of UIO sequences impact on *STUIOS* effectiveness.

6.1 Implicit Modelling of Time Uncertainty

In this section we write that time uncertainty is *modeled explicitly* when it is captured by means of time ranges in the TAs; this is the case of the model in Figure 2 where a time range is used to indicate when the transition between locations `Present` and `PresentQualified` can be triggered. We refer to time uncertainty being *modeled implicitly* when the system is characterized by time uncertainty but time ranges are not used in the TAs.

A modeling solution for *HOD* where time uncertainty is modeled implicitly is shown in Figure 10. In Figure 10, the TA labeled with *(a)* models the main control loop; this TA indicates that the main control loop is triggered every 1700 ms and that, after being started, the control loop first detects if an error is present (see the event `detect`) and then determines if an error can be qualified (see the event `qualify`). The safety-related operations performed by the main control loop are executed in 1400 ms and, after this time, the system goes into the idle state. The automaton labeled with *(b)* models the detection of temperature errors (the initial location is `NotDetectedNotQualified`) and shows that the location `DetectedNotQualified` becomes active when the main control loop sends the event `detect` and a temperature error is present in the system (i.e., the equality '`TE == true`' holds). Location `DetectedQualified` becomes active if the variable `q` is equal to two when the event `qualify` is triggered (this happens if the temperature error remains present for two consecutive executions of the main control loop). If the error condition disappears (i.e., if '`TE == false`') while the location `DetectedNotQualified` is active, then the location `NotDetectedNotQualified` becomes active. The disqualification of errors is modelled in a similar manner. The automaton labeled with *c* is a simplified environment model that indicates that a temperature error condition (i.e., temperature out of valid range) might be present (i.e., '`TE = true`') or absent (i.e., '`TE = false`', which is set when the temperature is within the valid range).

The purpose of the model in Figure 10 is different than the one in Figure 2; however, both models can be used to derive test cases and oracles. The TA in Figure 2 aims to capture the timing requirements of the system regardless of its internal design and implementation, while the network of TAs in Figure 10 is a design specification for *HOD*. The TA in Figure 2 describes the expected behaviour of *HOD* from the perspective of an actor (e.g., a human tester) interacting with the system. The network of TAs in Figure 10 explicitly captures the presence of a main control loop and specifies the constraints that regulate error qualification in terms of the number of execution

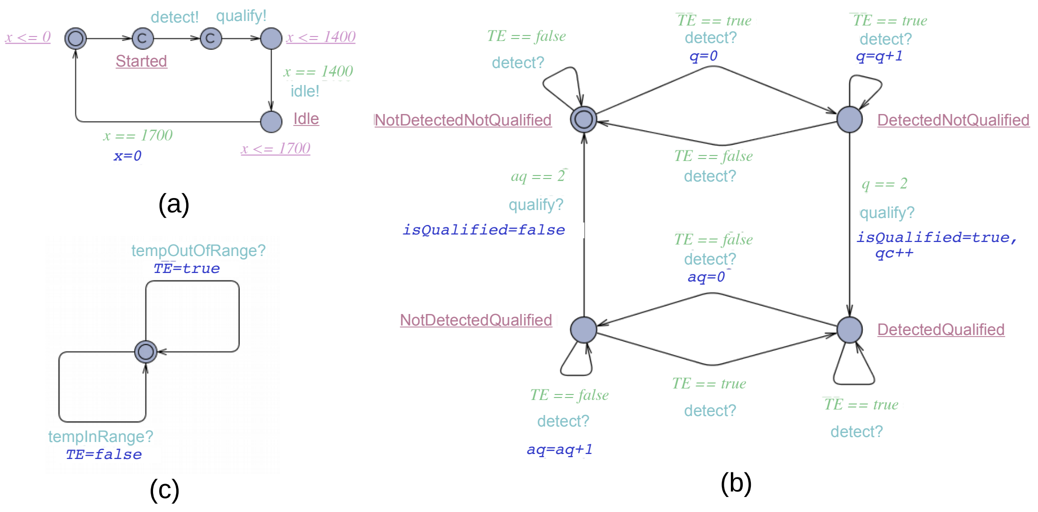


Fig. 10. Network of TAs that captures the frequency of the main control loop and the procedure for qualifying temperature errors in *HOD*. The TA labelled with (a) models the main control loop, the TA labelled with (b) captures the detection and qualification of temperature errors, the TA labelled with (c) is a simplified environment automata that captures the fact that temperature errors might be present or absent. The label 'C' within a location is used to indicate that the location is 'committed'; edges departing from committed locations have priority over other edges in a network of TAs [11]. Committed locations are used, for example, to model atomic operations (e.g., an activity triggered immediately after an interrupt is received).

```
<tempOutOfRange>/ (isQualified == false, qc == 0)
<wait for 4250 ms> /  (isQualified == true , qc == 1)
<tempInRange> /  (isQualified == true , qc == 1)
<wait for 1700 ms> /  (isQualified == true, qc == 1)
<wait for 1700 ms> /  (isQualified == true, qc == 1)
<wait for 1700 ms> /  (isQualified == false, qc == 1)
```

Fig. 11. Test case that verifies the qualification of a temperature error that appears for a short time. Test inputs match the network of TAs in Figure 10. The bold lines match the UIO sequence that checks if the test case brings the system to location `DetectedQualified`.

cycles during which the error remains present in the system (i.e., what happens in the actual implementation).

The explicit modeling of the main control loop in Figure 10 makes the explicit modeling of time uncertainty redundant because the time range in which an error can be qualified (or disqualified) can be derived from the model. In practice, the system is still characterized by time uncertainty although the network of TAs in Figure 10 does not explicitly model time uncertainty. Time uncertainty can be observed by focussing on the delay between the moment in which a temperature error becomes present in the system (i.e., TE is set to `true` by the environment automaton) and the moment in which the temperature error is qualified (i.e., location `DetectedQualified` becomes active), which varies between 3400 ms and 5100 ms. This is due to the fact that error detection occurs every 1700 ms (i.e., the event `detect` is fired every 1700 ms), while the transitions of the environment automaton, including the one that sets TE to `true`, might be fired anytime.

STUIOS can generate effective *stochastic test cases* even in the presence of networks of TAs that do not explicitly model time uncertainty. Indeed, the four activities performed by *STUIOS* to

Table 2. Output generated by the UIO sequence in Figure 11 for all the locations of the TA (b) in Figure 10. For each location we report the value of the state variables *isQualified* and *qc*. Bold red values highlight the differences with the expected output in case of transition faults.

Input	DetectedQualified	NotDetectedNotQualified	DetectedNotQualified	NotDetectedQualified
`<wait for 1700ms>`	`true, 1`	`true, 1`	`true, 1`	`true, 1`
`<wait for 1700ms>`	`true, 1`	`true, 1`	`true, 1`	**`false`**`, 1`
`<wait for 1700ms>`	`false, 1`	**`true`**`, 1`	**`true`**`, 1`	-

build stochastic test cases (i.e., determine the context of a UIO, identify potential UIO sequences, determine valid UIO sequences, and calculate output probabilities) are based on UPPAAL features whose results do not depend on the explicit or implicit modelling of time uncertainty. Regarding *context determination*, STUIOS relies on UPPAAL reachability analysis. The reachability analysis implemented by UPPAAL is based on state traversal with difference bounded matrices and clock difference diagrams [7, 9, 40], which produce correct results independently from the number and type of constraints appearing in the TAs. The identification of *potential UIO sequences* is based on the simulation feature of UPPAAL, which correctly returns all the transitions that are enabled in a given time range, independently of whether time uncertainty is explicit. To determine if a potential UIO sequence is valid, *STUIOS* relies on the reachability analysis feature of UPPAAL, which is guaranteed to provide correct results, as mentioned above. Finally, to calculate output probabilities, *STUIOS* relies on the probability estimation feature of UPPAAL. This feature is based on Monte-Carlo simulation, and, as such, is not affected by how time ranges are modeled, but simply reflects the behaviors captured by the model. Figure 11 shows an example test case generated by *STUIOS* from the TAs in Figure 10; Table 2 shows the output of the UIO sequence in Figure 11 in the presence of transition faults.

6.2 Non-deterministic Transitions

The SMC features of UPPAAL require that TAs be *input-deterministic*, which means that for every location l all possible inputs (event or delay) may trigger only one outgoing edge of the location l at a given instant. Input-deterministic TAs may include time-triggered transitions if these are controlled by guard conditions that are disjoint from the other guard conditions appearing in the other edges departing from the same locations. The TA in Figure 10 is input-deterministic because, for every possible clock value, only a single edge can be taken. Recall from Section 3.1, however, that time-triggered transitions introduce time uncertainty in (*input-deterministic*) TAs and thus lead to non-deterministic behaviours.

Since the implementation of *STUIOS* relies on UPPAAL, it also requires input-deterministic TAs. In the context of embedded systems, this is unlikely to constitute a practical limitation since such systems are often reactive systems that perform a single specific action in response to a certain event.

6.3 Availability of UIO Sequences

UIO sequences are not guaranteed to exist for all the locations of a TA. However, like FSMs [2], TAs are often characterized by the presence of UIO sequences.

In this Section we characterize a class of input-deterministic TAs that *always present a UIO sequence for every location*, which is the class of minimal, connected, and deterministic TAs. A deterministic TA is an input-deterministic TA that does not include time-triggered transitions. A TA is connected when each location can be reached from any other location, which is usually true for software systems modeled using TAs and FSMs (for example, TAs that model a control loop are connected) [29]. A TA is minimal when for every pair of locations, it is possible to identify an input sequence that distinguishes one location from the other by generating different outputs, depending on the location on which it is applied. In the software testing literature, an input sequence that distinguishes two locations is known as *signature sequence* [32]. When a deterministic TA is connected, for every pair of locations $< l_a, l_b >$, it is possible to identify a sequence of inputs that transitions the system from l_a to l_b. As a consequence, when a deterministic TA is both connected and minimal, for every location l, it then possible to derive a UIO sequence that has the following characteristics. The UIO sequence traverses the location l multiple times and is characterized by an input sequence that can be represented as $< SS_1, FS_1, SS_2, FS_2, ..., SS_x, FS_x >$, where SS_i is a signature sequence that distinguishes location l from location l_i, and FS_i is an input sequence that transitions the system from the location reached by SS_i to l. The UIO sequence thus contains one signature sequence SS_i for each location l_i to be distinguished from l. The number of *signature sequences* included in the UIO sequence is at most equal to the number of locations in the TA minus one.

Based on the observation above, we can additionally conclude that any TA that is determinizable[1], minimal and connected will thus include UIO sequences. This is the case, for example, of event-clock automata, which is a class of determinizable TAs where each clock is associated to a specific event and is reset after the occurrence of the event [4]. The TA in Figure 10, for example, is an event-clock automaton where the clock is reset after every event.

Finally, despite having identified a class of TAs that present a UIO for all the locations, it is worth mentioning that even in the presence of TAs that are not deterministic, all the desired final test locations of a test suite might present UIO sequences. In addition, when a UIO sequence cannot be generated for a desired final test location, *STUIOS* remains useful since it enables engineers to verify the frequency of the test output sequence.

7 EMPIRICAL EVALUATION

To evaluate the effectiveness of the *STUIOS* approach, we conducted an empirical evaluation, based on two industrial case studies, addressing the following four research questions:

- *(RQ1) Does* STUIOS *generate oracles that are effective in the presence of time uncertainty?* This research question aims to evaluate the effectiveness of *STUIOS* in terms of fault detection capability of the generated test suite when faults affecting timing requirements are characterized by time uncertainty.
- *(RQ2) How does* STUIOS *compare with a simpler and less expensive alternative?* This research question aims to determine whether *STUIOS* performs significantly better than a much simpler and less expensive alternative that derives test cases that do not include UIO sequences and whose oracles rely solely on analyzing output sequences.
- *(RQ3) Is the test execution overhead introduced by* STUIOS *acceptable in an industrial context?* This question aims to evaluate whether *STUIOS* can be successfully adopted in an industrial context and what are the costs associated with its adoption.
- *(RQ4) How does the maximum number of execution per test case (maxE) affect the accuracy of* STUIOS *oracles?* This question aims to evaluate to what extent the choice of the maximum number of test case executions (*maxE*) affects the accuracy of *STUIOS* oracles in terms of failing and passing executions being correctly classified.

7.1 Subject of the study

The subjects of our case study are *HOD* [22] and *BodySense*^{TM} [21], two embedded systems already on the market and developed by our industrial partner, IEE [23]. *HOD* has been introduced in

[1]A TA is determinizable when it is possible to derive an equivalent, deterministic TA [6, 27].

Section 2 and is an embedded system that detects if a car driver has both hands on the steering wheel. *HOD* is a sensing system that measures the capacitance between a conductive layer in the steering wheel and the electrical ground in the car body and the seat frame. HOD provides the input for autonomous driving assistance systems, notifying them whether or not the driver has his hands on the steering wheel.

BodySense^{TM} is a car seat occupant classification system that enables smart airbag deployment. *BodySense*^{TM} relies on a capacitive sensor in the vehicle's passenger seat to determine whether the seat is occupied, as well as classifying the occupant. If the passenger seat has a child in a child seat or is unoccupied, the system disables the airbag. For seats occupied by adult passengers, *BodySense*^{TM} ensures the airbag is deployed in the event of an accident. *BodySense*^{TM} is integrated with multiple car sensors (e.g., capacitive sensors and temperature sensors) and communicates with other car controllers by means of a LIN bus. For example, *BodySense*^{TM} communicates the car occupancy status to the airbag controller in order to automatically enable or disable airbags; also, it communicates with the seat belt controller to support seat belt reminder functions.

BodySense^{TM} targets occupant classification in compliance with the U.S. National Highway & Traffic Safety Administration and also covers the Euro NCAP requirements for automatic airbag disabling, required for any rearward-facing child seat.

The two systems run on a real-time operating system and present several critical timing requirements that are currently verified by developers with an automated test suite, in compliance with the ISO-26262 safety standard [24].

Table 3. Purpose and size of the TAs belonging to the case study systems.

	Modeled Specification (number of TAs)				**Total number of TAs**	**Total Size**	
	Error qualification /disqualification	Control Loop	Message Handling	Environment		Locations	Edges
HOD	33	-	1	-	34	223	107
BodySense^{TM}	22	2	2	2	28	238	110

HOD and *BodySense*^{TM} are two embedded systems that are required to timely identify error conditions in order to guarantee the safety of car passengers and drivers. In our experiments, we focus on test cases that verify the conformance of these two systems with the timing requirements related to the qualification of error conditions. Table 3 provides an overview of the TAs for the two case studies. The timing specification of *HOD* are captured by means of 34 TAs: 33 TAs capture the timing requirements for the qualification and disqualification of different error conditions while one TA models message handling. Of the 33 TAs capturing error qualification, 20 TAs share the template of temperature errors shown in Figure 2, while the remaining 13 TAs match the templates in Figures 12-a and 12-b. In total, the timing specifications of *HOD* include 223 edges and 107 locations. The specifications of *HOD* model high-level timing requirements regarding error qualification, not their implementation, thus they do not explicitly model the control loop and the environment (see Section 6.1). The timing specification of *BodySense*^{TM} are captured by means of 28 TAs: 22 TAs capture the qualification and disqualification of error conditions, two TAs model the main control loop, two TAs model interrupts and message handling, two TAs model the environment. The TAs of *BodySense*^{TM} resemble the ones in Figure 10. In total, the timing specifications of *BodySense*^{TM} include 238 edges and 110 locations.

7.2 Experiment setup

In our experiments we deal with the evaluation of the effectiveness of test suites in detecting faults in a program. To generate faulty versions of *HOD* and *BodySense*^{TM} we relied on mutation analysis.

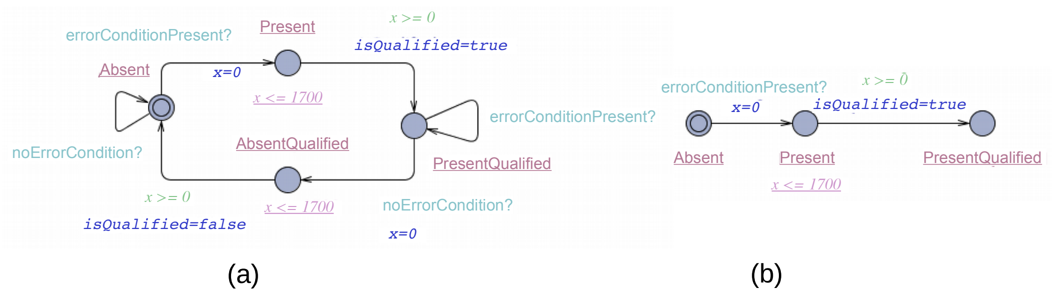


Fig. 12. Additonal templates for TAs modeling timing requirements for *HOD*. Figure (a) models the timing requirements for errors that are immediatly qualified (or disqualified) when they are recognized as present (or absent). Figure (b) models the timing requirements for (critical) errors that cannot be disqualified after being (immediatly) qualified.

More specifically, we derived the faulty versions of *HOD* and *BodySense*^{TM} by means of specification mutation (i.e., by using mutation operators to derive software specifications that characterize the behaviour of a faulty program). In the literature, different mutation operators for TAs have been described (e.g., [1, 3]). For our experiments, we considered mutation operators that directly affect timing properties, which are: Restricting Clock Conditions [1], Widening Clock Conditions [1], Shifting Clock Conditions [1], and Change Target Location [3]. Applying the above mutation operators led to a total of 651 mutated specifications, 328 for *HOD* and 323 for *BodySense*^{TM}.

Manually modifying the implementation of *HOD* and *BodySense*^{TM} to create 651 faulty versions would require unacceptable engineering effort. This would also be the case for deploying the faulty versions on dedicated hardware for purely experimental purposes. For this reason, we generated 651 programs that implement the mutated specifications of the two systems. Since these programs are generated for experimental purposes, they only implement the behavior under test related to timeliness and are therefore much simpler to implement and modify.

Because of the large number of test suite runs[2], we used a high-performance computing cluster (HPC) to run our experiments [37]. The experiments were run on a Bullx B500 blade system [5] with each node having two processors (Intel Xeon L5640, 2.26 GHz). In our experiments, since each test suite has been executed on a dedicated node, test cases belonging to the same test suite have been executed in sequence, while different test suites have been executed in parallel.

The artifacts required to replicate the empirical results presented in the paper (case study systems and test suites) are available for download from the following URL: https://sntsvv.github.io/STUIOS/.

7.3 Results

7.3.1 RQ1. RQ1 aims to evaluate the effectiveness of *STUIOS*, that is the capability to detect failures. To respond to *RQ1*, we evaluated the effectiveness of the oracles generated by *STUIOS* when failures are related to software timeliness, and the executed test cases lead to non-deterministic behaviour because of time uncertainty. For our experiments we focused on failures that may affect the capability of *HOD* and *BodySense*^{TM} to qualify error conditions (e.g., overheating) within the given time ranges.

We considered both a valid implementation of the specifications of the two systems and a subset of the 651 faulty versions described above. Since we aim to evaluate the effectiveness of *STUIOS*

[2]To respond to RQ1 we executed one test suite against three versions of the system; to respond to RQ2 we executed ten test suites against 651 faulty programs. Details are provided in the following subsections.

in the presence of failures related to software timeliness, we considered mutated specifications that lead to error conditions being qualified with an erroneous timing. This is the case of the mutated specifications derived by applying the mutation operators Restricting Clock Conditions and Widening Clock Conditions on the TA edges that capture the qualification time of each different error condition. Applying the two mutation operators on the *HOD* and *BodySense*^{TM} specifications leads to two different faulty versions for each TA regarding the detection of error conditions, each containing a single fault. *HOD* specifications include 33 TAs modeling error conditions, which leads to 66 faulty versions of *HOD*. *BodySense*^{TM} specifications include 22 TAs modeling error conditions, which leads to 44 faulty versions of *BodySense*^{TM}.

Since the injected faults affect the qualification of error conditions, to perform our experiments, we considered a set of test cases verifying if an error condition is qualified according to its timing specifications. In order to assess the worst-case performance of STUIOS, we have considered test cases for which *STUIOS* requires a high number of executions before determining the test outcome. To identify these test cases, we have considered the fact that the Wilson score formula requires the largest sample to generate a confidence interval of length 0.1 when the probability of observing the expected output is 0.5. In the case of *HOD*, for example, this occurs with test cases that check if *HOD* is able to qualify a specific error condition within 4250 ms (i.e., the middle point in the allowed range). In total we considered 55 test cases, each one focussing on one of the error conditions detected by *HOD* (33 error conditions) and *BodySense*^{TM} (22 error conditions).

We executed each test case against both the valid version and the two faulty versions for a total of 165 (i.e., 55×3) test case runs (recall that each run may imply repeated executions of the test case because of the probabilistic nature of *STUIOS*). Test cases executed against the correct implementation are expected to pass while test cases executed against faulty versions are expected to fail.

To answer *RQ1*, we computed the precision and recall according to standard formulas. In our context, true positives correspond to test cases correctly failing against the faulty version, whereas false positives correspond to test cases unexpectedly failing against the correct version. True negatives match test cases not failing when executed against a faulty version.

Empirical results show that *STUIOS* provides perfect precision and recall (1.0) in both case study systems. Please note that test cases with *simple oracles*, i.e., test cases that do not include UIO sequences but only analyze output sequences, cannot be executed in this context. For example, in the case of *HOD*, simple oracles would fail every time the system does not qualify errors in 4250 ms, which happens in 50% of the passing executions since the qualification time follows a uniform distribution.

7.3.2 RQ2. *RQ2* aims to determine whether *STUIOS* performs significantly better than a much simpler and less expensive alternative that derives test cases that rely upon *simple oracles*. No existing approach focuses on deriving UIO sequences from specifications provided as TAs with time uncertainty.

To perform the experiment, and avoid bias in the selection of the test cases, for the case of *BodySense*^{TM} we considered ten test suites of 122 test cases each, which were automatically generated in previous work for the same software under test [39]. The test suites were generated using the TAUC test generation technique (detailed in [39]), which is a model-based testing technique that relies on meta-heuristic search to generate test suites that (1) cover all the transitions in a network of TAs and (2) stress timing requirements by sending multiple, diverse inputs to the system. TAUC maximizes the diversity among the sequences of inputs being sent to the system by the different test cases in a test suite. In previous work, TAUC has shown to be able to generate test suites that discover faults not detected by the test suite manually produced by software engineers [39]. For the case of *HOD*, we considered 100 test cases generated according to the same criterion adopted for the *BodySense*^{TM} test suites.

The test suites generated with TAUC (hereafter *standard test suites*) include only simple oracles. For each standard test suite we used *STUIOS* to generate a test suite containing stochastic test cases.

Although the case study systems process multiple types of inputs (not just error conditions) and all the test cases considered in the evaluation send multiple inputs to the system, we expect that system outputs always adhere to the distribution calculated by *STUIOS*. This happens because of the two reasons discussed in Section 3.1: (1) time uncertainty depends on the polling of the environment repeated at a given frequency and (2) inputs are not expected to interfere with the polling frequency (inputs are processed at the end of the control loop).

We executed each test suite on each of the 328 faulty software versions (mutants) for *HOD* and on each of the 323 mutants for *BodySense*^{TM}, and we kept track of passing and failing test cases. We measured fault detection effectiveness at the test suite level and report the percentage of faults discovered by both the stochastic and standard test suites.

The empirical results show that all the test suites are sound, i.e., they never report failures when executed against the valid version of the software. Furthermore, the test suites generated by *STUIOS* are more effective than the standard test suites since they are able to uncover on average, across the ten generated test suites, 96.95% of the faults in the case of *HOD* (96.95% min, 96.95% max) and 99.26% of the faults (98.76% min, 100% max) in the case of *BodySense*^{TM}, while the test suites integrating simple oracles detect on average 89.16% of the faults in the case of *HOD* (89.16% min, 89.16% max) and 88.76% of the faults in the case of *BodySense*^{TM} (86.38% min, 90.71% max). In the case of *BodySense*^{TM} *STUIOS* does not detect faults when their effect is masked by additional inputs (e.g., when inputs overwrite the incorrect values). In the case of *HOD*, the faults not being discovered are due to not being able to distinguish invalid final states from valid final states when exclusively relying on I/O sequences. This happens when the fault consists of turning into a self loop the edge that connects the locations `Present` and `PresentQualified` in Fig. 12-b; this fault affects ten faulty versions. The number of faults being discovered by *STUIOS* in *HOD* is constant since none of the test suites can discover these ten faults.

The results show that the test suites generated by *STUIOS* significantly improve the effectiveness of the standard test suites from which they have been derived, even in the presence of very effective test suites (the standard test suites detect 89% of the faults). *STUIOS* test suites uncover at least 7% more faults.

7.3.3 RQ3. *RQ3* aims to evaluate the costs associated with the adoption of *STUIOS* in an industrial context. *STUIOS* requires multiple executions of the same test case in order to determine the test outcome. In our context, like in many other embedded development environments, the repeated execution of test cases is not a problem. Software engineers are expected to set up the physical environment only once, at the beginning of the testing process, and as in many other industrial contexts, the execution of test suites is usually conducted automatically by test automation frameworks. The repeated execution of a same test case thus does not impact on the human effort required to perform testing but may introduce delays in the testing process. For this reason, to respond to *RQ3*, we discuss the impact on the time required to perform automated testing.

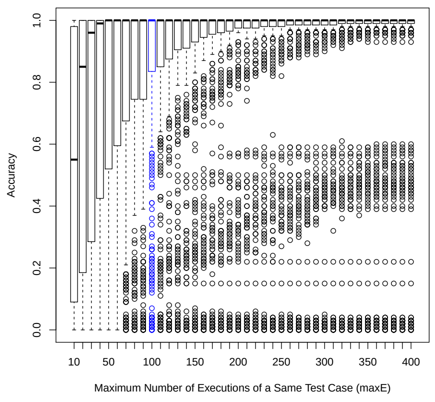
We compared the number of test executions required to generate test results with *STUIOS* against those required by the standard test suite used to respond to *RQ2*, i.e., 222 executions, one for each of the 222 test cases. In our empirical study, we determined that *STUIOS* requires 57 executions, on average, for each test case, with a minimum and maximum of 35 and 100 executions. Although the number of test case executions is high, the execution time is acceptable in practice. In our experiments, the execution of a test suite took on average 34 hours in the case of *HOD* and 32 hours in the case of *BodySense*^{TM}[3]. The difference between the two case studies is caused by differing characteristics of the two systems, which lead to test cases with longer execution times for *HOD*. For example, *BodySense*^{TM} is characterized by a shorter qualification time for all the error conditions, which in turn leads to quicker test cases.

Given that these test cases can be concurrently executed on several test environments and that acceptance testing usually lasts days, the reported increase in number of test cases executions has limited practical consequences.

7.3.4 RQ4. *RQ4* aims to evaluate the accuracy of *STUIOS* oracles (i.e., the capability of correctly classifying test case executions as passing or failing), in the presence of different thresholds for the maximum number of test case executions (*maxE*). In general, the accuracy of *STUIOS* oracles may vary across different execution scenarios; for example, a fault that leads to a frequency of the expected output which is very different from the expected one is easier to detect than a fault that leads to an output frequency that is close to the expected one. Hereafter, for brevity, we refer to the expected frequency of the expected output computed by UPPAAL as *expected output frequency* and we refer to the frequency of the expected output observed when executing the implemented system as *observed output frequency*. To evaluate the accuracy of *STUIOS*, ideally, one would like to consider multiple systems, correct and faulty, in order to cover different combinations of expected and observed output frequency. Since it is unrealistic to consider as many real systems as needed to perform such an experiment, we simulated *STUIOS* by considering 420 fictitious scenarios resulting from the combination of 20 different expected output frequencies and 21 different observed output frequencies. The 20 expected output frequencies considered in our study are represented as confidence intervals of length 0.1 with a central point p_e covering the values in the range [0.05,1.0]. The 21 observed output frequencies have been used to generate 21 sequences of test outcomes (i.e., sequences of 'success' and 'fail') that follow a Bernoulli distribution with a probability of success p_a; each output sequence thus simulates the output distribution of multiple runs of a same test case against the implemented system. For p_a, we considered all the values in the range [0,1], in steps of 0.05. In all these 420 scenarios we applied *STUIOS* (more precisely, the oracle strategy described in Section 5.4) on the sequences of test outcomes and determined if the output reported by *STUIOS* was correct or not. In this context, the output reported by *STUIOS* is correct in two cases: (case-1) *STUIOS* reports that a test case passes in the presence of an observed output frequency p_a that matches the expected output frequency p_e; (case-2) *STUIOS* reports that a test case fails in the presence of an observed output frequency p_a that does not match the expected frequency p_e.

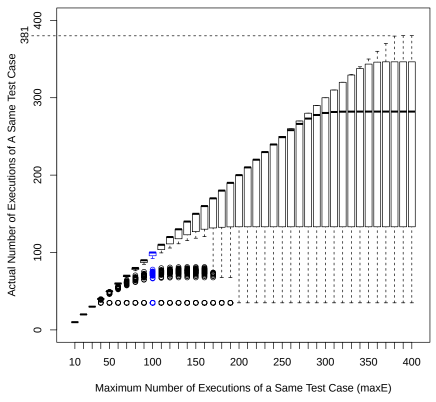
For all the 420 scenarios, we have executed *STUIOS* using a confidence level p_c equal to 95% and 40 different configuration values for *maxE* (we considered values in the range [10,400] with steps of 10). For each scenario and for each configuration, we have executed *STUIOS* 100 times. For each scenario, we computed the accuracy of *STUIOS* as the fraction of correct results generated in 100 *STUIOS* executions. The boxplot in Figure 13 shows the accuracy obtained for different values of *maxE*. Each plot is based on 420 data points (each data point is the accuracy computed after executing *STUIOS* 100 times on a certain scenario with a certain configuration for *maxE*) and reflects the accuracy variation for a given configuration of *maxE* across the 420 scenarios. The boxplot in Figure 14, instead, depicts cost by showing the number of test case executions required for each scenario, for the 40 configurations of *maxE*. The boxplot in Figure 13 shows that the accuracy of results improves for increasing values of *maxE* but this comes at the cost of a higher number of executions required for each single test case. In general, the median number of test case executions is close to *maxE*, which means that engineers will directly affect the test execution costs

[3]Note that the use of a HPC cluster does not introduce bias in the results (i.e., in the test suites execution time) because the test cases belonging to a same test suite are not executed in parallel but are executed in sequence on the same HPC node.



Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range. Default *STUIOS* configuration (i.e., $maxE = 100$) in blue.

Fig. 13. Accuracy boxplot with $p_c = 95\%$ for different values of *maxE*.



Minimum whisker value is Q1 - 1.5*IQR, maximum whisker value is Q3 + 1.5*IQR; where IQR is the interquartile range. Default *STUIOS* configuration (i.e., $maxE = 100$) in blue.

Fig. 14. Number of test case executions with $p_c = 95\%$ for different values of *maxE*.

by increasing the value of *maxE*. The median number of test case executions reaches a plateau as *maxE* gets closer to the upper limit for for the given confidence interval (e.g., values of *maxE* above 300 when $p_c = 95\%$, where 381 is the upper limit). The default configuration of *STUIOS* (i.e., $maxE = 100$) is a compromise between execution costs and accuracy (e.g., the first quartile in the boxplot in Figure 13 shows that 75% of the data points have an accuracy above 0.8). However, engineers may select different configurations based on the requirements of the system under test. For example, in the presence of an unstable system (e.g., with a high risk of clock drifts [36]), engineers may increase the chances of obtaining accurate results by setting $max_E = 300$. Similarly, when a certain degree of clock drift is acceptable, engineers may set the confidence level to 90%, which makes *STUIOS* more conservative (i.e., reject the null hypothesis less often) and thus prevents false alarms.

The outliers in Figure 13 are generated by executions where the observed output frequency is close to the expected frequency (e.g., 0.45 vs. 0.50). To clarify this, we show in Figure 15 the average STUIOS accuracy obtained with scenarios presenting various discrepancies between the expected and the actual frequency. Figure 15 does not include the case in which the observed frequency matches the expected frequency (i.e., $discrepancy = 0$) because the accuracy is trivially 1 in that case. Figure 15 shows that, as expected, the worst results are obtained when the discrepancy is minimal and equal to 0.05 (i.e., half the confidence interval). If engineers are interested in detecting faults that may lead to a discrepancy below or equal to 0.05, they should configure *STUIOS* with a

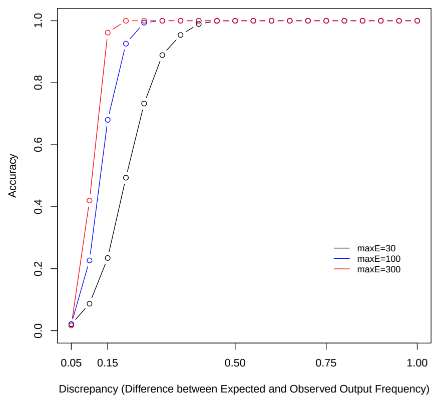


Fig. 15. Average accuracy for different discrepancy values and different values of *maxE*.

smaller confidence interval. However, failing to detect small discrepancies has limited practical consequences as the behavior of the implemented system is close to that of the specified one.

For example, in the case of *BodySense*^{TM}, a confidence interval of 0.1 is sufficient to detect faults and prevent false alarms because a discrepancy below 0.05 characterizes outputs that are delayed by 85 ms (0.05×1700, the duration of the control loop), which is the minimal time required for end-to-end communication between *BodySense*^{TM} and the test driver API, and thus may not be indicative of the presence of a fault. The plot in Figure 15 also shows that higher values for $maxE$ improve *STUIOS* accuracy in the presence of small but possibly relevant discrepancies. For example, with a discrepancy of 0.15, $maxE = 100$ leads to an average accuracy above 0.75, while $maxE = 30$ yelds a very low average accuracy, below 0.3.

7.4 Threats to Validity

Internal threats. To limit the threats to the internal validity of the empirical evaluation, that is, a faulty implementation of our toolset that may lead to erroneous results, we have manually verified that all the UIO sequences generated by our toolset are valid.

External threats. Threats to external validity concern the generalisability of results. *HOD* and *BodySense*^{TM} are control systems with a main control loop that polls data from the environment and then perform control operations. These kinds of control components are common in embedded systems. In addition to this, *HOD* and *BodySense*^{TM} are representative embedded systems in the automotive domain, with typical complexity (in terms of size and number of models required to capture its timing properties), and thus we expect that results generalise and *STUIOS* can be successfully adopted to test many other systems.

8 RELATED WORK

Related work includes techniques for the generation of test cases and oracles from state-based formalisms, e.g., finite state machines (FSMs), TAs, and Extended FSMs (EFSMs), as well as the testing of real-time systems.

Most of the related work for the generation of test cases and oracles based on finite-state specifications deals with FSMs, not TAs. Two surveys are relevant, one focusing on FSM-based testing approaches [26] and a more recent one also including experimental comparisons [16]. Other approaches focus on EFSMs [17, 31], but require deterministic EFSMs and thus cannot be applied in the context of this paper, as discussed in Section 5.3.1.

On the other hand, several approaches working with TAs target the generation of test cases but do not address the problem of generating precise test oracles in the presence of time uncertainty [1, 18, 34, 35]. For example, Springintveld *et al.* introduced a technique for translating TAs into grid automata to enable generation of test sequences by means of traditional algorithms for FSM testing [34].

Few techniques focussing on testing software timeliness in the presence of time uncertainty have been reported [12, 13, 19, 20, 25]. The main difference with *STUIOS* is that these approaches focus on test case generation and do not deal with the problem of generating test oracles for existing test suites.

Some of the techniques dealing with time uncertainty perform feedback-driven, model-based testing [12, 19, 25]. These techniques combine test generation and test execution in the same process. Test inputs are generated at runtime during test execution by performing a random exploration of the TA specifications. In the presence of non-determinism, a list of possibly active locations is updated by simulating the actual system execution on the TAs based on the outputs generated by the system. These approaches require the processing of models at runtime, an activity that may introduce delays, and thus make these techniques unsuitable to test real-time systems [12].

In addition to this, existing feedback-driven approaches, different from *STUIOS*, ignore some of the effects of uncertainty. More specifically, they simply verify that the system does not generate invalid results, but they do not verify if test outputs occur with the expected frequency. Further, they do not rely upon confidence interval estimation to decide when to stop testing, and are thus unable to spot faults that alter the frequency of occurrence of valid outputs.

Among the existing test generation approaches addressing time uncertainty, the only one dealing with the verification of the frequency of expected outputs is that of Hierons *et al.* [20], who rely on mutation testing and probability estimation to build test suites that verify if a software conforms to specifications given as probabilistic FSMs. The approach derives test cases that aim to spot specific implementation faults. To this end, the approach identifies sequences of test inputs that can determine if a mutated specification, which models a fault, differs from the original one. The approach proposed by Hierons *et al.* focuses on test generation — thus preventing engineers from reusing test suites derived with different criteria — while *STUIOS* focuses on the generation of oracles for existing test suites, regardless of how they were generated. Furthermore, the effectiveness of the approach from Hierons *et al.* has not been evaluated.

9 CONCLUSION

In the context of embedded systems, this paper addresses the automated generation of effective test oracles for timeliness test cases in the presence of time uncertainty. We propose an approach, named *STUIOS*, to generate stochastic cases that extend a set of test cases provided by software engineers. A stochastic test case includes the input-output sequences from initial test cases, a probability value that indicates the expected probability of observing the output sequence generated by the test case, plus a *probabilistic Unique Input Output (UIO) sequence*. The latter captures the probability of observing the output of a UIO sequence, identified by *STUIOS*, which helps predict if the test case has terminated in the desired final test location. Probability values capture the expected frequency of non-deterministic output sequences and thus make it possible for *STUIOS* to deal with non-determinism. *STUIOS* then relies on executing multiple times each test case in such a way that it can employ hypothesis testing to determine whether a test case has passed, i.e., if it has generated the expected output with the expected frequency, at a given level of confidence.

Results from two industrial case studies in the automotive domain, involving typical embedded systems, have demonstrated that *STUIOS* is able to achieve higher fault detection effectiveness compared to a previous test approach which did not include stochastic test cases. In practice, though *STUIOS* expectedly requires additional test execution time, this can be considered to have negligible practical consequences if test case execution is automated.

ACKNOWLEDGMENTS

This work is supported by IEE S.A. and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277). We would like to thank Thierry Stephany and the IEE software engineers for their support.

REFERENCES

[1] M. Saeed AbouTrab, Michael Brockway, Steve Counsell, and Robert M. Hierons. 2013. Testing Real-Time Embedded Systems using Timed Automata based approaches. *Journal of Systems and Software* 86, 5 (2013), 1209 – 1223. https://doi.org/10.1016/j.jss.2012.12.030

[2] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. 1991. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *IEEE Transactions on Communications* 39, 11 (Nov 1991), 1604–1615. https://doi.org/10.1109/26.111442

[3] Bernhard K. Aichernig, Florian Lorber, and Dejan Ničković. 2013. Time for Mutants — Model-Based Mutation Testing with Timed Automata. In *Proceedings of Tests and Proofs: 7th International Conference (TAP 2013)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 20–38. https://link.springer.com/chapter/10.1007/978-3-642-38916-0_2

[4] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. 1999. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science* 211, 1 (1999), 253 – 273. https://doi.org/10.1016/S0304-3975(97)00173-4

[5] Atos. 2018. Bullx B500 blade system. https://atos.net/en/products/high-performance-computing-hpc/bullsequana-x-supercomputers. (2018).

[6] Christel Baier, Nathalie Bertrand, Patricia Bouyer, and Thomas Brihaye. 2009. When Are Timed Automata Determinizable?. In *Automata, Languages and Programming*, Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 43–54.

[7] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. 2002. UPPAAL Implementation Secrets. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2 (FTRTFT '02)*. Springer-Verlag, London, UK, UK, 3–22. http://dl.acm.org/citation.cfm?id=646847.707113

[8] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1995. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proceedings of Workshop on Verification and Control of Hybrid Systems III (Lecture Notes in Computer Science)*. Springer–Verlag, 232–243. https://doi.org/10.1007/BFb0020949

[9] Johan Bengtsson and Wang Yi. 2004. *Timed Automata: Semantics, Algorithms and Tools.* Springer Berlin Heidelberg, Berlin, Heidelberg, 87–124. https://doi.org/10.1007/978-3-540-27755-2_3

[10] C. J. Clopper and E. S. Pearson. 1934. The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. *Biometrika* 26, 4 (1934), 404–413. http://www.jstor.org/stable/2331986

[11] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. 2015. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer* 17, 4 (01 Aug 2015), 397–415. https://doi.org/10.1007/s10009-014-0361-y

[12] Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Marius Mikucionis, and Brian Nielsen. 2012. *Testing Real-Time Systems under Uncertainty.* Springer Berlin Heidelberg, Berlin, Heidelberg, 352–371. http://doi.org/10.1007/978-3-642-25271-6_19

[13] A. David, K. G. Larsen, S. Li, and B. Nielsen. 2009. Timed Testing under Partial Observability. In *2009 International Conference on Software Testing Verification and Validation.* 61–70. https://doi.org/10.1109/ICST.2009.38

[14] Robert I. Davis and Alan Burns. 2011. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.* 43, 4, Article 35 (Oct. 2011), 44 pages. https://doi.org/10.1145/1978802.1978814

[15] Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. 2006. Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs. *Comput. J.* 49, 3 (May 2006), 331–344. https://doi.org/10.1093/comjnl/bxl003

[16] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R. Cavalli, and Nina Yevtushenko. 2010. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology* 52, 12 (2010), 1286 – 1297. https://doi.org/10.1016/j.infsof.2010.07.001

[17] A. Y. Duale and M. U. Uyar. 2004. A method enabling feasible conformance test sequence generation for EFSM models. *IEEE Trans. Comput.* 53, 5 (May 2004), 614–627. https://doi.org/10.1109/TC.2004.1275300

[18] Abdeslam En-Nouaary. 2008. A scalable method for testing real-time systems. *Software Quality Journal* 16, 1 (2008), 3–22. https://doi.org/10.1007/s11219-007-9021-8

[19] Susanne Graf and Laurent Mounier (Eds.). 2004. *Black-Box Conformance Testing for Real-Time Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-24732-6_8

[20] Robert M. Hierons and Mercedes G. Merayo. 2009. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software* 82, 11 (2009), 1804 – 1818. https://doi.org/10.1016/j.jss.2009.06.030

[21] IEE. 2018. BodySense product description. https://www.iee.lu/en/products/automotive/bodysense. (2018).

[22] IEE. 2018. HOD product description. https://www.iee.lu/en/products/automotive/hands-off-detection. (2018).

[23] IEE. 2018. Sensing solutions. http://www.iee.lu. (2018).

[24] ISO. 2011. ISO-26262: Road vehicles – Functional safety. (2011).

[25] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. 2005. Testing Real-time Embedded Software Using UPPAAL-TRON: An Industrial Case Study. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05)*. ACM, New York, NY, USA, 299–306. https://doi.org/10.1145/1086228.1086283

[26] D. Lee and M. Yannakakis. 1996. Principles and methods of testing finite state machines-a survey. *Proc. IEEE* 84, 8 (Aug 1996), 1090–1123. https://doi.org/10.1109/5.533956

[27] Florian Lorber, Amnon Rosenmann, Dejan Ničković, and Bernhard K. Aichernig. 2017. Bounded determinization of timed automata with silent transitions. *Real-Time Systems* 53, 3 (01 May 2017), 291–326. https://doi.org/10.1007/s11241-017-9271-x

[28] N. Schenker M. E. Payton, M. H. Greenstone. 2003. Overlapping confidence intervals or standard error intervals: What do they mean in terms of statistical significance? *Journal of Insect Science* 34, 3 (2003), 19–35. https://doi.org/10.1016/j.infsof.2015.03.009

[29] Aditya P. Mathur. 2009. *Foundations of Software Testing* (2nd ed.). Pearson Education.

[30] T. Ramalingom, K. Thulasiraman, and A. Das. 1996. Context independent unique sequences generation for protocol testing. In *INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, Vol. 3. 1141–1148. https://doi.org/10.1109/INFCOM.1996.493058

[31] Christopher Robinson-Mallett, Peter Liggesmeyer, Tilo Mücke, and Ursula Goltz. 2006. Extended state identification and verification using a model checker. *Information and Software Technology* 48, 10 (2006), 981–992. https://www.sciencedirect.com/science/article/abs/pii/S0950584906000425

[32] Krishan Sabnani and Anton Dahbura. 1988. A protocol test generation procedure. *Computer Networks and ISDN Systems* 15, 4 (1988), 285 – 297. https://doi.org/10.1016/0169-7552(88)90064-5

[33] Nathaniel Schenker and Jane F. Gentleman. 2001. On Judging the Significance of Differences by Examining the Overlap Between Confidence Intervals. *The American Statistician* 55, 3 (2001), 182–186. http://www.jstor.org/stable/2685796

[34] Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. 2001. Testing Timed Automata. *Theor. Comput. Sci.* 254, 1-2 (March 2001), 225–257. https://doi.org/10.1016/S0304-3975(99)00134-6

[35] Mohammad Saeed Abou Trab, Bachar Alrouh, Steve Counsell, Rob M. Hierons, and George Ghinea. 2010. *A Multi-criteria Decision Making Framework for Real Time Model-Based Testing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 194–197. https://doi.org/10.1007/978-3-642-15585-7_20

[36] D. Trawczynski, J. Sosnowski, and J. Zalewski. 2007. The Effect of Large Clock Drifts on Performance of Event and Time Triggered Network Interfaces. In *2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX '07)*. 344–351. https://doi.org/10.1109/DEPCOS-RELCOMEX.2007.48

[37] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. 2014. Management of an Academic HPC Cluster: The UL Experience. In *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*. IEEE, Bologna, Italy, 959–967.

[38] Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie. 2013. A survey of timed automata for the development of real-time systems. *Computer Science Review* 9 (2013), 1 – 26. https://doi.org/10.1016/j.cosrev.2013.05.001

[39] Chunhui Wang, Fabrizio Pastore, and Lionel Briand. 2017. System Testing of Timing Requirements based on Use Cases and Timed Automata. In *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 299–309. https://doi.org/10.1109/ICST.2017.34

[40] Mihalis Yannakakis and David Lee. 1997. An Efficient Algorithm for Minimizing Real-Time Transition Systems. *Formal Methods in System Design* 11, 2 (Aug. 1997), 113–136. https://doi.org/10.1023/A:1008621829508

[41] Guang Yong Zou, Wenyi Huang, and Xiaohe Zhang. 2009. A note on confidence interval estimation for a linear function of binomial proportions. *Computational Statistics and Data Analysis* 53, 4 (2009), 1080 – 1085. https://doi.org/10.1016/j.csda.2008.09.033

Received June 2018