System Testing of Timing Requirements based on Use Cases and Timed Automata

Chunhui Wang, Fabrizio Pastore, Lionel Briand
SNT - University of Luxembourg
Email: {chunhui.wang,fabrizio.pastore,lionel.briand}@uni.lu

***Abstract*—In the context of use-case centric development and requirements-driven testing, this paper addresses the problem of automatically deriving system test cases to verify timing requirements. Inspired by engineering practice in an automotive software development context, we rely on an analyzable form of use case specifications and augment such functional descriptions with timed automata, capturing timing requirements, following a methodology aiming at minimizing modeling overhead. We automate the generation of executable test cases using a test strategy based on maximizing test suite diversity and building over the UPPAAL model checker. Initial empirical results based on an industrial case study provide evidence of the effectiveness of the approach.**

I. Introduction

For most embedded systems, standards require system testing to explicitly demonstrate that the software meets its functional and safety requirements, e.g. the capability of the system to promptly detect error conditions.

Motivated and inspired by actual engineering practice in the automotive domain, this paper addresses the automatic generation of system test cases for testing software timeliness, i.e. the ability of the software to satisfy timing constraints. A solution is devised to target embedded system development contexts where use case specifications are used to specify functional requirements and partial timed automata are adopted to capture timing requirements. We tackle two major issues: (1) the automation of the test generation process with minimal modelling overhead, and (2) the identification of input sequences that increase the likelihood that the system will break timing constraints.

Existing techniques for testing software timeliness rely upon models that capture the timing constraints of the system, mostly Timed Automata [1], [2], [3], [4] or UML statecharts [5]. To limit modelling effort, our observation is that, in practice, software engineers exclusively use timed automata (or statecharts) to model the state transitions controlled by timing constraints, but not the complex functional logic that brings the system into a specific state. The consequence is that *automated techniques are currently used only to generate abstract test cases* that are then concretized into executable test cases by means of test adaptation or test transformation approaches [6].

Unfortunately, the cost for transforming an abstract test case into an executable test case is not negligible. Test engineers actually need to carefully read the functional specifications of the software to determine the inputs that bring the system into a specific state. The only possible alternative to fully automate the generation of executable test cases is to rely on very detailed system models, e.g. automata that integrate timing constraints and constraints on events and data inputs. However, if functional requirements have already been documented, for example in the widely-used form of use case specifications, software engineers are unlikely to produce additional detailed models.

Semi-automated approaches that support software engineers in automatically deriving system test cases from textual requirement specifications exists [7], [8], [9], [10], but they do not deal with the problem of testing software timeliness, e.g. a temperature error can only be confirmed when the measurement value goes out of range for a certain time to prevent signal toggling. For example, UMTG [10], from our previous work, addresses this challenge in the context of use case-driven development and relies on the natural language processing of use case specifications, elicited using a restricted format, to extract the information required for testing the system.

When testing software timeliness, software engineers must take into account the fact that the violation of timing constraints may depend on specific sequences of inputs. In other contexts, meta-heuristic search has shown to be effective in identifying the software inputs that lead to worst-case scenarios in embedded systems in terms of reaching critical states [11] or missing task deadlines [12]. In our context, our objective is different as we want to identify critical sequences of inputs leading to violations of timing requirements and, further, we must search for such sequences without running test cases on the deployment platform to guide us.

This paper proposes *Test Generation combining Timed Automata and Use Case Specifications*, TAUC, an approach and a tool for automatically generating executable test cases targeting software timeliness. TAUC automatically builds the detailed models necessary to generate executable test cases by combining the information appearing in functional specifications (use case specifications) and models of the timing requirements of the system (timed automata). It thus prevents software engineers from designing very detailed timing automata when use case specifications are available. In addition, TAUC automatically builds test suites that exercise the functionality regulated by timing requirements by optimising diversity among test inputs, thus increasing the probability of identifying problems dependent on specific input sequences.

This paper has been accepted for publication in the proceeding of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017), IEEE. DOI: 10.1109/ICST.2017.34.

To automate testing, TAUC needs to determine what functional inputs are needed to reach certain system states, and for this reason, TAUC must identify the dependencies between functional scenarios and timed automata. More specifically, TAUC determines which functional scenarios bring the system into a specific state, and which functional scenarios can take place when the system is in a given state. TAUC relies upon UMTG to identify the test inputs that exercise functional scenarios, and automatically processes test inputs and timed automata to identify their dependencies.

TAUC automatically models these dependencies by augmenting the set of user-provided timed automata capturing timing requirements. This helps contain the engineers' modelling effort and enables the use of UPPAAL [13], a model checker for symbolic reachability analysis of timed automata. It is employed for the generation of test cases that include both inputs derived from use case specifications and timing constraint on the inputs, e.g. the delay between two inputs, derived from timed automata.

Test generation with UPPAAL guarantees edge coverage, i.e. functional coverage, but does not aim at increasing the chance to identify the violation of timing requirements. For this reason TAUC includes a meta-heuristic search algorithm that iteratively modifies the test cases generated by UPPAAL to maximize test suite diversity within a certain test budget. Test suite diversity is maximized by deriving test cases that provide diverse sequences of inputs to the system, thus increasing the probability to identify violations of timing requirements that depend on specific input sequences.

The paper proceeds as follows. Section II provides background information on UMTG. Section III motivates our work. Section IV provides an overview of TAUC. Sections V to IX detail TAUC steps. Section X reports on empirical results obtained with an industrial case study in the automotive domain. Section XI discusses related work. Section XII concludes the paper.

II. Background on UMTG

TAUC relies upon *UMTG* to process functional requirements. This Section provides background information on the modelling methodology that should be adopted by software engineers to elicit functional requirements that can be processed with *UMTG*. To this end, we present a simplified version of the design artifacts of *BodySense*, an automotive system developed by our industrial partner IEE [14].

A. Eliciting Use Case Specifications

We expect that software engineers elicit functional requirements by means of use case specifications written according to the RUCM format. RUCM is a use case format that provides restriction rules and keywords constraining the use of natural language in use cases. For details, the reader is referred to [15]. UMTG [10], on which TAUC builds, relies on the application of RUCM in order to enable the automated analysis of use case specifications.

TABLE I
BodySense Use Cases

0	**Use Case: Identify the Occupancy Status of a Seat**
1	**Precondition**
2	The system has been initialized
3	**1.1 Basic Flow**
4	1.The system REQUESTS capacity FROM the seat sensor.
5	2.INCLUDE USE CASE Self diagnosis.
6	3.INCLUDE USE CASE Classify occupancy status.
7	4.The system VALIDATES THAT no error is detected and no error is qualified.
8	5.The system VALIDATES THAT the occupant class is valid.
9	6.The system SENDS the occupant class TO AirbagControlUnit.
10	Postcondition: The occupant class has been sent to AirbagControlUnit.
11	**1.2 Specific Alternative Flow**
12	RFS 4
13	1. The system sends the error class to AirbagControlUnit.
14	2. ABORT.
15	Postcondition: Classification filters have been reset.
...	...
30	**Use Case: Self Diagnosis**
31	**2.1 Basic Flow**
32	1.The system REQUESTS temperature FROM the temperature sensor.
33	2.The system VALIDATES THAT the temperature is valid.
...	...
37	**2.2 Specific Alternative Flow**
38	RFS 2
39	1. The system sets the TemperatureError as detected.
40	2. RESUME 3.

Table I shows two use cases of *BodySense*, *Identify the Occupancy Status of a Seat* (Line 0) and *Self Diagnosis* (Line 30). The former describes the main functionality of *BodySense*, while the latter deals with the identification of runtime errors, e.g. it detects the presence of a *TemperatureError* if the temperature measured by the sensor is out of range (Line 39). In Table I, RUCM keywords enabling automated processing of use case specifications are written in capital letters.

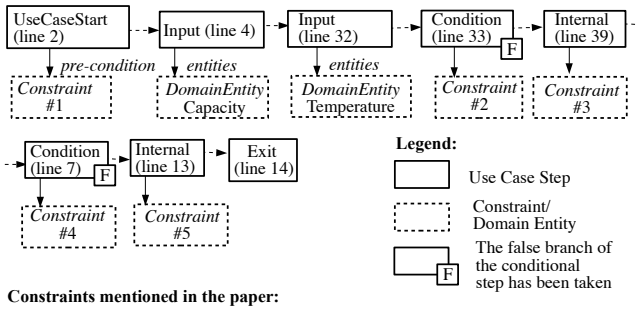
In addition to use case specifications, UMTG also requires a domain model of the system. The design of domain models is a common software analysis practice [16]. To enable test case generation, UMTG requires that software engineers specify constraints expressed in the Object Constraint Language (OCL [17]), based on the domain model. These constraints precisely specify conditional statements, effects of internal steps, and post-conditions of use case flows.

The constraint *TemperatureSensor: self.temperature.value* $>$ *0 and self.temperature.value* $<$ *40* for example, is used to further specify the conditional statement in Line 33 (*the temperature is valid*) and shows that the temperature is valid when its value lies between 0 and 40.

B. Identification of Test Inputs with UMTG

UMTG identifies a set of *functional scenarios*, i.e. sequences of steps in a use case specification, that ensure that the basic flow and each alternative flow of the use case specifications are covered at least once.

Fig. 1 shows the functional scenario that covers line 39 of *BodySense* use case specifications. Each scenario is a sequence of steps in the use case specification. *UMTG* associates to each step the following information: Input steps are linked to the domain entities received as input by the system. Conditional steps are linked to the OCL constraints that further specify them, e.g. the step of Line 33, which detects the presence of a temperature value out of range, is linked to constraint



Constraints mentioned in the paper:

Constraint#2: context TemperatureSensor: self.temperature.value > 0 and self.temperature.value < 40)

Constraint#3: context TemperatureError: self.isDetected = true

Constraint#4: context Error: self.isDetected = false and self.isQualified = false

Fig. 1. Example of a functional scenario

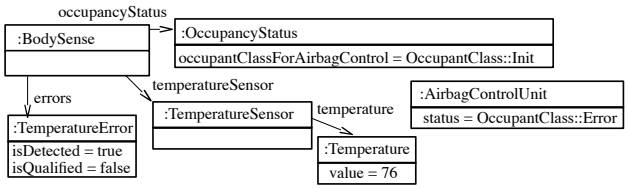


Fig. 2. Object diagram generated by *UMTG*.

#2 (see Fig. 1). Internal steps are linked to postconditions that characterize the state resulting from their execution. The internal step of Line 39, for example, is linked to constraint #3, which indicates that the system has set *TemperatureError* as being detected (see Fig. 1).

The constraints appearing in a use case scenario are used by *UMTG* to identify test inputs. For each scenario under test, *UMTG* builds a path condition and uses a constraint solver dedicated to OCL to identify an instance of the domain model, i.e. an object diagram, for which the path condition evaluates to true. Fig. 2, for example, shows an object diagram that satisfies the path condition required to exercise the scenario in Fig. 1.

III. Motivating example

This section presents an example that shows that abstract test cases generated with traditional approaches usually require additional engineers effort to be used as executable test cases, while TAUC automatically generates executable test cases thus saving engineers effort.

A. Modelling of Timing Requirements

The timing properties of a system are typically modelled using Communicating Timed Automata [18] or UML statecharts. In this paper we make use of the former, and adopt the formalization from [13].

A timed automaton 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 communicating timed automata, the state of the system is captured by the values of state variables and the set of active locations across all the automata. Actions are used to synchronize different automata. 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.

Fig. 3-a shows a timed automaton that captures the timing properties concerning the qualification of temperature errors. The variable x is a clock variable, while $isDetected$ and $isQualified$ are two state variables. The edge that connects locations *NotDetectedNotQualified* and *DetectedNotQualified* can be fired only when the event *detected*, that indicates that a temperature error has been detected, has been received by the automaton. The location *DetectedNotQualified* has an invariant that indicates that the clock variable *x* must be below 4800 ms when the location is active. The guard condition on the edge between the location *DetectedNotQualified* and *DetectedQualified* indicates that this edge cannot be fired if the clock *x* is below 3100. In effect the edge between locations *DetectedNotQualified* and *DetectedQualified* can be fired anytime when the clock variable x has a value between 3100 and 4800.

B. Limitations of Test Generation Based on Timed Automata

Software engineers may use the timed automaton of Fig. 3-a to automatically generate timeliness abstract test cases by using an automated technique [19]. The first three operations performed by an automatically generated test case might be: (1) generate the event *detected* (which is supposed to bring the system into the state *DetectedNotQualified* from the initial state *NotDetectedNotQualified*), (2) wait for 4801 milliseconds (to be sure that the edge has been fired), (3) check that the system is in the state *DetectedQualified* and otherwise fail.

To transform this abstract test case into an executable test case the software engineer needs to determine how to generate the event *detected*. This activity is not trivial because one has to carefully read the software specifications to determine the conditions under which a temperature error is detected. For a complex system, finding all the temperature requirements could be particularly expensive. For example, the engineer might need to read all the specification documents to be sure that the temperature range does not vary according to working conditions (e.g. in the presence of other system errors).

C. Automated Testing with TAUC

TAUC assumes that software engineers have produced use case specifications written in the RUCM format like the one shown in Section II, and timed automata that capture timing requirements for the identification of errors, such as the one in Fig. 3-a.

The test generation process implemented by TAUC guarantees that all the edges of the timing requirements automata

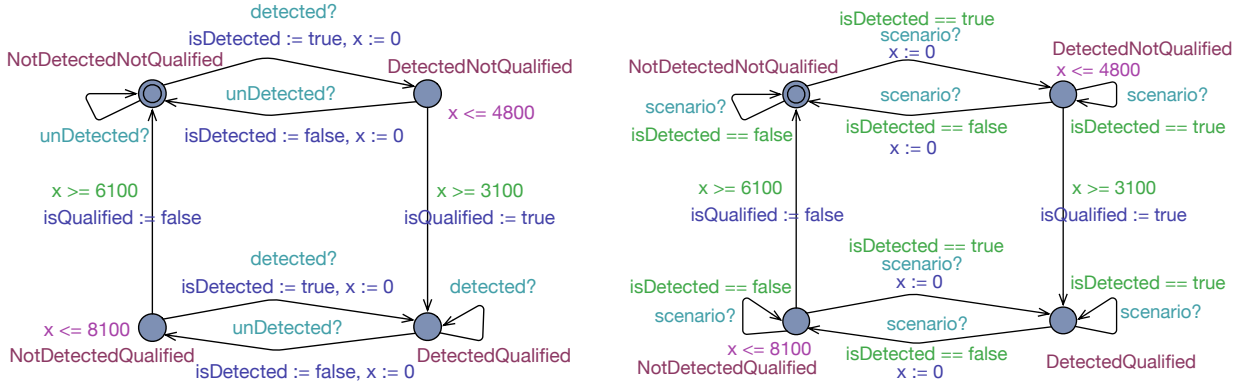


Fig. 3. (a) Automaton that captures how TemperatureErrors are qualified and dequalified in *BodySense*. (b) Same automaton, but automatically augmented by TAUC with dependencies to functional scenarios. Edges are labelled by the triple guard (green), action (light blue), and update (blue).

are covered at least once and that, furthermore, test suites contain test inputs that are combined in ways to maximize their diversity.

For example, TAUC may generate a timeliness test case for *BodySense* that begins in the state *NotDetectedNotQualified* (initial state of the system), simulates the sending of the value 76 from the temperature sensor (in this way the edge is fired and the new active location is *DetectedNotQualified*), waits for 4801 milliseconds, and checks that the system is in the state *DetectedQualified*. In contrast to the test case generated with a traditional approach (Section III-B), the test case generated by TAUC does not require that the software engineer manually determines that an input temperature with a value above 40 is needed to reach location *DetectedNotQualified*. The concrete test input to be used is automatically generated by TAUC.

To identify the concrete test inputs to be used in a test case, TAUC relies upon the identification of dependencies between functional scenarios and timed automata. These dependencies are used to determine which functional scenarios need to be exercised to bring the system to a specific state. In the case of *BodySense*, TAUC automatically detects that there is a dependency between the event *detected* on the automata in Fig. 3-a and the use case step of Line 39, *The system sets the TemperatureError as detected*. This dependency enables TAUC to determine that the inputs that exercise the use case scenario that covers Line 39, i.e. an object diagram that assigns the value 76 to the temperature sensor, is necessary to bring the system into the state *DetectedNotQualified*.

TAUC also generates test cases that highly differ from one another, to maximize the diversity of the test suite. A test case, different from the one presented above, is generated by TAUC to send an interrupt, followed by a message, while the system is in the location *DetectedNotQualified*. Such test cases check the effect of different inputs on timing requirements. Details are provided in the coming sections.

IV. Overview of TAUC

When use case specifications are used to specify the software functional behaviour, TAUC spares software engineers

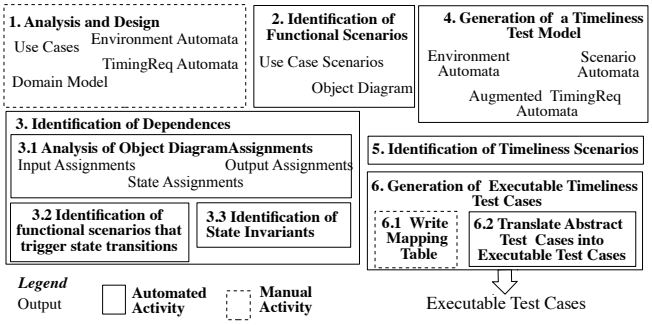


Fig. 4. Steps of TAUC.

from the burden of manually determining the functional inputs required for testing timing requirements. TAUC works in six steps, shown in Fig. 4.

In Step 1, *analysis and design*, software engineers produce RUCM use case specifications that capture the functional requirements of the system, and timed automata that capture both the timing requirements of the system and the timing properties of the environment. Note that in practice, our observation is that such specifications are commonly used to communicate with customers and other stakeholders.

In Step 2, *identification of functional scenarios*, the technique relies upon UMTG to automatically identify functional scenarios, and the inputs required to trigger each scenario.

Step 3 concerns the *identification of the dependencies between functional scenarios and timed automata*. We identify two types of dependencies: (1) the outputs produced by the system during the execution of specific functional scenarios may correspond to events that trigger state transitions, (2) specific functional scenarios might be executed only in certain system states, i.e. only when specific state invariants hold. TAUC automatically identifies these dependencies.

In Step 4, TAUC generates a Timeliness Test Model, i.e. a collection of models that include environment automata provided by software engineers in Step 1, automata generated by TAUC to model functional scenarios (scenario automata), and automata generated by TAUC by extending timing require-

ments automata. Timeliness Test Models capture the dependencies between functional scenarios and timing requirements automata, and thus enable the adoption of UPPAAL for the generation of executable test cases targeting timeliness.

In Step 5, TAUC derives *timeliness scenarios*. A timeliness scenario is a sequence of delays, edges and locations of the Timeliness Test Model, that specifies a valid execution. A single timeliness scenario generally covers multiple functional scenarios. TAUC generates an initial suite of *timeliness scenarios* with UPPAAL. This initial test suite is then extended by TAUC using a meta-heuristic search strategy that maximizes diversity among the *timeliness scenarios* in the test suite.

In Step 6, TAUC generates *executable test cases* from the timeliness scenarios derived in Step 5. This activity is performed by means of a mapping table provided by software engineers.

The next sections describe in details the different steps of TAUC with the exception of Step 2, which coincides with the execution of *UMTG*, already described in Section II.

V. Analysis and Design

Software engineers produce the artifacts required by TAUC: use case specifications written according to the RUCM format, the domain model (designed as UML class diagram), and timed automata for timing requirements. The notations for use case specifications and timed automata have already been introduced in Sections II and III. This section focuses on the methodological aspects of the analysis and design phase, where we specify the system aspects that should be modelled by means of timed automata: the timing requirements under test, and the relevant aspects of the environment.

A. Modelling Timing Requirements

The identification of timing requirements is a manual activity. We expect that software engineers define a timed automaton for each entity in the domain model that features one or more timing constraints.

State variables appearing in the timed automata have counterparts in the domain model. More specifically, each state variable is also an attribute of the entity modelled by the timed automaton. Fig. 3-a shows the automaton that captures the timing constraints related to the entity *TemperatureError*. The variables *isDetected* and *isQualified* are two attributes of the entity *TemperatureError* and, therefore, the assignments to the state variables *isDetected* and *isQualified* capture changes to the state of the entity *TemperatureError*.

We observed that in practice, software engineers avoid details that are unnecessary to describe timing requirements, i.e. details about the functional operations that trigger certain state transitions. In fact software engineers often produce timed automata with edges synchronized with events that stand for the execution of one, or several, specific functional scenarios. These events are not generated by any other automata of the system; we call these events *scenario events*, since they are expected to be generated as a result from the completion of a functional scenario.

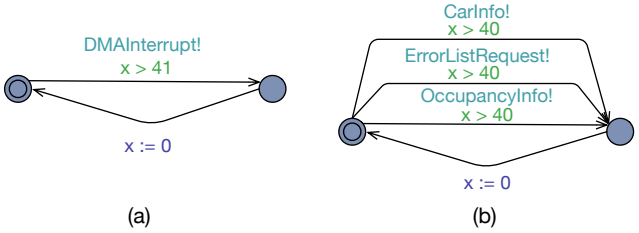


Fig. 5. Environment automata for *BodySense*

Section VI shows how TAUC automatically detects *scenario events* without requiring additional information from software engineers. This allows software engineers to follow usual design practices with reasonably small timed automata models.

B. Modelling the Environment

Environment automata are used to capture the arrival frequency of inputs, including interrupts or messages. This information is necessary to avoid the generation of invalid test cases, e.g. test cases that send inputs with a frequency not supported by the system bus.

We model the arrival of inputs by means of events. The event names must match entity names used in the domain model. Fig. 5 shows two environment automata of *BodySense* that capture timing characteristics of interrupts and messages. For example, the minimal interarrival time of DMA interrupts is 41 milliseconds (a), while messages' minimal interarrival time is 40 milliseconds (b).

VI. Identification of Dependencies

Dependencies between timed automata and functional scenarios are of two kinds: (1) outputs produced by functional scenarios may fire state transitions, (2) certain functional scenarios can be executed if and only if the system has reached specific states, i.e. specific state invariants are true.

To identify these dependencies, TAUC works in three steps: First, TAUC looks for scenario outputs and assignments to state variables by analyzing the object diagram associated by *UMTG* to each specific scenario. Then it compares scenario outputs and update operations in the automata to identify the edges fired by each scenario. Finally, it relies upon state assignments to identify the state invariants of each functional scenario.

A. Analysis of attributes in object diagrams

To identify the dependencies between scenarios and timed automata, TAUC first classifies each value assigned to the attributes of the object diagrams to distinguish between the inputs required to exercise a functional scenario, the state invariants that must hold when a functional scenario is executed, and the output generated during the execution of the scenario.

TAUC classifies the assignments in the object diagrams as *input*, *output*, and *state*.

Output assignments set values to satisfy the post-conditions of internal steps. *Input assignments* and *state assignments* instead set values that satisfy the constraints associated to the conditional steps of a scenario. Assignments to attributes of

Temperature.value:=76	**InputAssignment**: regards an attribute of class Temperature, linked to the input step of line 32.
TemperatureError.isDetected:=true	**OutputAssignment**: generated to satisfy constraint#3, which is linked to an internal step.
TemperatureError.isQualified:=false	**StateAssignment**: generated to satisfy constraint #4. TemperatureError is not linked to any input step.

Fig. 6. Assignments required to exercise the scenario in Fig. 1.

domain entities appearing in input steps are considered *input assignments*. Assignments to attributes of domain entities not associated with input steps are *state assignments* (attributes that are not input parameters must be state variables).

Fig. 6 shows a classification of the assignments appearing in the object diagram of Fig. 2. The assignment *Temperature.value := 76* in Fig. 6 is an *input assignment*. Indeed, the attribute *value*, which belongs to the entity *Temperature*, is referred to in the input step of Line 32 (Fig. 1). The assignment *TemperatureError.isDetected := true* is an *output assignment* since it satisfies the *constraint #3*, which is associated with the InternalStep of Line 39 (Fig. 1). The assignment *TemperatureError.isQualified := false* is a *state assignment* since it satisfies the negation of *constraint #4*, which is associated with the conditional step of Line 7 (Fig. 1), and TemperatureError is not an input, thus implying *TemperatureError.isQualified* is a state variable.

B. Identification of functional scenarios that trigger state transitions

In Section V we have described how software engineers adopt *scenario events* to reduce modelling effort. To further reduce engineers effort, TAUC automatically detects the functional scenarios that dispatch each *scenario event*.

By relying upon *scenario events*, software engineers hide details about the complex relationships between system inputs and the firing of an edge. For example in the automaton of Fig. 3-a, the edge that connects states *NotDetectedNotQualified* and *DetectedNotQualified* is fired upon the reception of the scenario event $detected$. The model does not specify the input constraints under which the edge is fired, but shows the effect of the execution of this edge on the system state, i.e. the assignment of the value $true$ to variable *isDetected*.

Since *scenario events* trigger edges that update the system state, to automatically detect the scenarios that dispatch scenario events, TAUC assumes that a functional scenario triggers a scenario event if it brings the system into the same state, i.e. if it generates a set of *output assignments* that is a superset of the assignments in the update operations associated with the edge that consumes the scenario event.

The functional scenario in Fig.1 leads to the output assignment *TemperatureError.isDetected:=true* (see Fig. 6), which appears also in the updates of the edge that connects locations *NotDetectedNotQualified* and *DetectedNotQualified* (see the timed automata of Fig. 3-a). For this reason TAUC determines that the scenario in Fig. 1 is the one that dispatches the event *detected* that triggers the edge between the two abovementioned locations.

C. Identification of state invariants

Functional scenarios are enabled only when specific state invariants hold. For example, in *BodySense*, a scenario that performs self diagnosis in the absence of error conditions cannot be exercised if the system already detected errors in previous executions.

State invariants are implicitly specified in use case specifications by means of constraints that capture properties of state variables. TAUC makes the state invariants associated with each functional scenario explicit by identifying the constraints that determine the value of each state assignment. State assignments aim to satisfy conditions that must hold to execute a specific functional scenario and these conditions are thus state invariants.

In the running example the state assignment *TemperatureError.isQualified:=false* results from Constraint #4 in Fig. 1. This constraint is thus a state invariant that must hold to execute the functional scenario.

In the presence of constraints that relate both input and state variables, TAUC extracts the subexpression concerning state variables only. In the presence of multiple state assignments, TAUC builds a state invariant that joins all the constraints that determine the value of the different state variables.

VII. Generation of the Timeliness Test Model

We use the term Timeliness Test Model to indicate a network of communicating timed automata that enable the generation of timeliness test cases.

Timeliness Test Models include the environment automata, scenario automata that capture the behaviour of functional scenarios, and augmented timing requirements automata. Scenario automata and augmented timing requirements automata are automatically generated by TAUC. Augmented timing requirements automata include guard conditions and events that capture the dependencies between timing requirements automata and functional scenarios. The Timeliness Test Models capture all the information required for test generation in the form of timed automata, thus enabling TAUC to rely upon UPPAAL for the automatic generation of test cases.

Scenario automata

For each functional scenario, TAUC automatically generates a scenario automaton. Each scenario automaton contains two edges.

The first edge of the scenario automaton captures dependencies by means of a guard condition for the state invariant of the scenario, and an update for each output assignment. This means that the edge can be fired, i.e. the scenario is executed, only if the state invariant holds. The firing of the edge updates the state of the system as specified by the output assignments. In other words, the scenario automaton shows when a scenario executes and how it affects the state of the system. Given that state invariants identified in the previous step are expressed as OCL constraints, TAUC translates these constraints in a format compatible with the UPPAAL syntax. According to our experience, the set of primitive types supported by UPPAAL

(boolean, integer, and double) is rich enough to not limit the kinds of constraint that can be translated in practice.

Fig. 7 shows an example of generated automaton. In the case of Fig. 7, the guard condition on the first edge indicates that TemperatureError should not have been already qualified at the beginning of the scenario (as shown by the guard condition *TemperatureError.isQualified* == *false*). The update operations on the edge capture the scenario output by setting the TemperatureError as detected (see the assignment *TemperatureError.isDetected := true*).

The second edge of the scenario automaton is used to synchronize scenarios and augmented timing requirements automata. This edge generates the event $scenario$, which is consumed by the augmented timing requirement automata, and indicates that the scenario has been executed to completion. This edge also presents a guard condition that captures the scenario frequency (provided by the software engineer). In the case of *BodySense*, each scenario is executed every 1700 milliseconds (See Fig. 7).

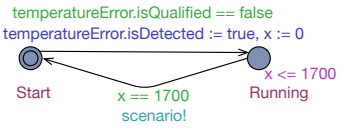


Fig. 7. Automaton for the Scenario in Fig. 1

Augmented timing requirements automata

The next step is to modify and augment timing requirements automata. TAUC proceeds as follows. For each scenario event mapped to a scenario output, TAUC replaces the event name in the timing requirements automata with the synchronization event $scenario?$, which indicates that the event is dispatched only if a scenario had been executed to completion.

Furthermore, TAUC adds a guard condition that checks if the output of the scenario just executed corresponds with the output assignment mapped to the scenario event. This way an edge with a scenario event is fired only if the scenario that generates the corresponding output assignment has been executed.

Fig. 3-b shows, for example, that the edge from location *NotDetectedNotQualified* to location *DetectedNotQualified* is fired only if a functional scenario has been executed and *temperatureError* has been set to detected. The latter is captured through a guard condition which distinguishes different scenarios.

VIII. Identification of Timeliness Scenarios

TAUC automatically generates the timeliness test suite for the system, which consists of a set of timeliness scenarios. A timeliness scenario is a sequence of delays, edges and locations of the Timeliness Test Model, that specifies a valid execution.

A minimal test adequacy criterion to test software timeliness is to exercise all the timing constraints at least once. This is achievable by relying upon a test suite that covers all the locations with invariants including clock variables and all the edges with guard conditions including clock variables. In practice this condition can be easily satisfied by a test suite that achieves edge coverage. However, in addition to this, we should consider that some failures may only be triggered by specific sequences of test inputs that might not be generated by achieving edge coverage.

In our context we thus need to generate timeliness scenarios in such a way that, within a test budget, we achieve edge coverage and maximize our chances to find a sequence of inputs that triggers a failure. Moreover, these two goals should be achieved without relying upon the execution of test cases during test generation. In fact in many embedded systems, the test budget, i.e. the number of test cases that can be executed, is limited because test execution is particularly expensive, e.g. the test execution environment and hardware must be manually set up. Given our objectives and constraints, we present below a test strategy based on maximizing diversity among timeliness scenarios.

Our strategy to increase the chances to trigger a failure, is to generate a test suite that: 1) executes as many diverse paths as possible that include the same edges relevant to timeliness, 2) executes paths with a maximum diversity of inputs, interrupts, and messages. However, in the presence of multiple timed automata, the search space for the identification of execution sequences is large and, for a given test budget, cannot be exhaustively explored to identify an optimal diverse subset of timeliness scenarios. For this reason the test generation process implemented by TAUC relies upon a metaheuristic search algorithm that maximizes the differences among the timeliness scenarios in the test suite.

The algorithm performs four activities described in the following paragraphs: *Generation of Scenarios That Cover All Edges*, *Generation of New Scenarios*, *Computation of Similarity Score*, *Identification of Scenarios that Maximize Diversity*.

Generation of Scenarios That Cover All Edges

To achieve edge coverage, TAUC relies upon the approach proposed in [19]. It builds a reachability formula that checks for the existence of a trace for which all the edges are covered. TAUC then relies upon UPPAAL to identify the shortest timeliness scenario (called trace in UPPAAL) that satisfies the given formula.

Some of the events generated by UPPAAL automata correspond to parametric system inputs. In the case of *BodySense* this happens for messages of type *CarInfo*, which have two associated parameters, *velocity* and *beltStatus*. Parameter values are not generated by UPPAAL, but are required to properly execute the system. For this reason, TAUC processes the timeliness scenarios generated by UPPAAL and automatically selects the parameter values to be associated to parametric inputs. TAUC simply assigns to each parameter a valid, randomly selected value. In the case of *BodySense*, we deal with both enumerations and numeric values. Enumeration values are selected from the values indicated in the domain model, while numeric values are randomly selected within a range specified by the software engineer.

Generation of New Scenarios New scenarios are then generated by mutating an existing scenario. To this end TAUC selects an existing timeliness scenario and a mutation point, i.e. a position in the scenario, and then generates a new scenario by copying the events in the scenario up to the mutation point and then by filling the rest of the new scenario with new events. These new events are identified by employing the simulation functionality provided by UPPAAL, which, given the current state of the automata, returns a list of enabled edges. New events are randomly selected till the same length of the initial timeliness scenario is reached. Values of parametric events are generated as well, by following the same procedure indicated previously.

Computation of Similarity Score The diversity among the timeliness scenarios in a test suite is maximized when the average pairwise similarity between all pairs of timeliness scenarios in a test suite is minimized [20].

To compute the similarity score, TAUC takes into account differences in both event names and event parameter values. TAUC relies upon the Lenvenshtein string alignment algorithm to identify matching events [21]. Similarly to [20], TAUC assigns a score of +3 to matching events, -2 to mismatching events, and -1 to gaps. In addition to this, in the presence of mismatching parameter values, TAUC decreases the score by a value between -1 (gap) and -2 (mismatch). More precisely, TAUC decreases the score by 1 plus the fraction of mismatching parameter values. Fig. 8-a shows the similarity score calculated for the alignment of two timeliness scenarios of *BodySense*.

Identification of Scenarios that Maximize Diversity

To build the timeliness test suite, TAUC first augments the UPPAAL test suite with newly generated timeliness scenarios until the test suite reaches the desired size (budget). Such new scenarios are then iteratively updated to maximize diversity.

TAUC replaces a scenario already in the test suite with a new scenario only if the new scenario augments the diversity of the test suite. More precisely, every time a new scenario is generated, TAUC identifies the scenario that should be replaced to obtain a test suite with the lowest average pairwise similarity. Test generation terminates after a given number of iterations provided by the user.

IX. Generation of Executable Test Cases

For each *timeliness scenario* TAUC generates a corresponding *abstract test case*. Abstract test cases are an intermediate representation used by TAUC for the generation of the final executable test cases.

Abstract test cases are composed of a sequence of input operations, delays and oracles. Fig. 8-b shows an abstract test case generated from a timeliness scenario of *BodySense*. Input operations are followed by the input parameters to be set, delays are identified by the keyword *wait* followed by the time to wait for, while oracles (identified by the keyword *check* in Fig. 8-b) consist of a list of expressions that check the values of observable system variables.

TABLE II
Mapping Table for *BodySense*

Pattern to match (Operation and Inputs)		Result (Operation and Inputs)	
Input	TemperatureSensor.value = (*)	SetBus	Pin =TEMPERATURE Value = $1
Check	TemperatureError.isDetected=true	CheckBus	TEMPERATURE_ERROR=01h

TABLE III
Fault Coverage Measured Against 323 Faulty Versions.

	Average Coverage (± Std. Dev) by Test suite size (number of test cases)				
	25	50	75	100	122
TAUC	85% ±0	88% ±0.41	91% ±0.46	91% ±0.38	91% ±0.42
Random	7% ±0	12% ±3.26	22% ±3.16	30% ±5.96	40% ±3.58
Manual	-	-	-	-	60%

To generate test inputs TAUC focusses only on the edges that belong to environment automata and scenario automata. For each edge of environment automata that generates an event, TAUC identifies a corresponding test input (event names correspond to entities of the domain model, see *CarInfo* in Fig. 8). Every time TAUC encounters the first edge of a scenario automata, it adds to the abstract test case the *input assignments* associated with the scenario to be executed (see Step 1 in Fig. 8-b). All the other edges of environment and scenario automata are ignored because they correspond to synchronization operations between automata.

To generate delays TAUC takes into account the fact that timing properties are typically characterized by uncertainty expressed in terms of ranges in which edges are expected to be taken. TAUC focusses on the maximum amount of time in the range and introduces a delay using a *wait operation* blocking for the maximum amount of time in the range.

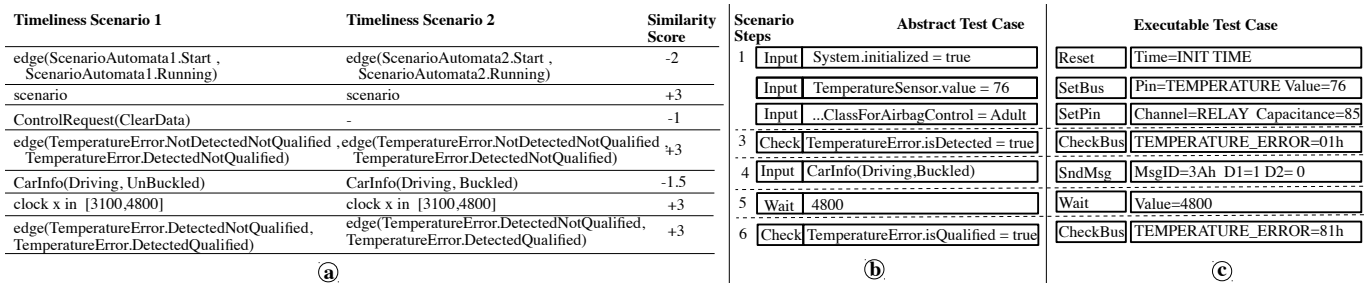
To generate oracles TAUC should ideally add, for each edge, an operation that checks if the expected target location has been reached. However, given that many embedded systems, such as BodySense, do not make their current internal state fully observable, TAUC can be configured to derive an oracle that checks for the values assigned to observable system variables. Fig.8-b shows an oracle derived from the edge that connects location *DetectedNotQualified* with location *DetectedQualified* (Step 6 of Scenario 2).

Abstract test cases are then translated into executable test cases by means of mapping tables as described in [10]. Mapping tables contain regular expressions that match the inputs in the abstract test cases and allow to replace abstract test inputs with concrete test inputs. Table II shows a portion of the mapping table used to transform the abstract test case in Fig. 8-b into the executable test case of Fig. 8-c.

X. Empirical evaluation

We empirically evaluated the fault detection capability of TAUC, i.e. the capability of TAUC to determine if the software implementation does not meet its timing requirements. The case study used for our experiments is *BodySense*, which is a typical, non-trivial embedded system developed on a real-time operating system - MicroC/OS [22]. *BodySense* is already in production and presents several critical timing requirements that IEE engineers must verify in compliance with the ISO-26262 safety standard [23].

To perform our experiments, we implemented TAUC in Java and integrated it with UMTG [24] and UPPAAL. TAUC



Timeliness Scenario 1	Timeliness Scenario 2	Similarity Score
edge(ScenarioAutomata1.Start , ScenarioAutomata1.Running)	edge(ScenarioAutomata2.Start , ScenarioAutomata2.Running)	-2
scenario	scenario	+3
ControlRequest(ClearData)	-	-1
edge(TemperatureError.NotDetectedNotQualified , TemperatureError.DetectedNotQualified)	edge(TemperatureError.NotDetectedNotQualified , TemperatureError.DetectedNotQualified)	+3
CarInfo(Driving, UnBuckled)	CarInfo(Driving, Buckled)	-1.5
clock x in [3100,4800]	clock x in [3100,4800]	+3
edge(TemperatureError.DetectedNotQualified, TemperatureError.DetectedQualified)	edge(TemperatureError.DetectedNotQualified, TemperatureError.DetectedQualified)	+3

Fig. 8. (a) Alignment of two Timeliness Scenarios, (b) Abstract Test Case generated from Timeliness Scenario 2, and (c) Executable Timeliness Test Case.

prototype is available at the following URL: *https://taucgen.github.io*.

In our experiments we compared the fault coverage of the *BodySense* test suite manually written by software engineers familiar with the system and domain, with the fault coverage of 10 test suites (to account for randomness) generated with TAUC and with a random approach, which serves as a baseline. To be fair, we configured the random approach to generate timeliness scenarios with the same number of events in the test cases generated by TAUC. We considered test suites of various sizes, including 25, 50, 75, 100, and 122 test cases. The *BodySense* actual test suite contains 122 test cases and was therefore considered to represent a realistic test budget.

Modifying the implementation of *BodySense* to create faulty versions for purely experimental purposes and running all the test suites on all mutant implementations on the actual deployment platform is far too expensive. For this reason we automatically generated 323 faulty versions of the Timeliness Test Model by means of dedicated mutation operators, then simulated the execution of the test cases against the mutated models by adapting the approach described in [25] to our case.

Different mutation operators for timed automata are described in literature [26], [27]; for our experiments we considered the mutation operators that may impact timing requirements, including Restricting Clock Conditions [26], Widening Clock Conditions [26], Shifting Clock Conditions [26], Change Target Location (CTL) [27]. We did not consider operators that alter the functional behaviour only, for example the ones that change the source and the target of an edge. However, to simulate the case in which the system is stuck in a state and breaks timing requirements, we configured the CTL operator to create self loops on edges with clock variables.

We generated each faulty version of *BodySense* models by executing a single mutation operator on the Timeliness Test Model. All the guard conditions, invariants, and edges of the Timeliness Test Model are mutated once by each applicable mutation operator. Mutation operators may lead to equivalent mutants, i.e. timed automata that satisfy the original requirements. We manually removed all the equivalent mutants and ended up with an evaluation benchmark of 323 mutant versions of BodySense timeliness test models.

Simulation of test cases execution [25] is based on UPPAAL and requires that executable test cases are translated into sequential timed automata to be composed with the *BodySense* mutated models. The generated automata have edges that either send inputs to the system or validate operation results. A dedicated testing clock is used to trace execution time. Operation results are validated by means of guard conditions that check if system variables have specific values when the testing clock reaches a given value, i.e the time appearing in the wait conditions of the executable test cases. An error state is reached when the values are not the expected ones. A test case is considered to fail if an error state is reachable.

Results. We measured the percentage of faults (mutants) identified by each test suite. A test suite identifies a fault if at least one of its test cases fails. Table III reports the results obtained with the three approaches (for TAUC and random we report the average across the 10 runs). The table clearly shows that TAUC detects between two and twelve times more faults than random testing, depending on the test suite size. It is also clear that TAUC is significantly more effective than expertise-based manual testing with 31% additional faults detected, on average. In addition, TAUC is more effective than both random and manual testing even with much smaller-size test suites. This mostly results from the capability of TAUC to generate input sequences covering non-trivial interactions between system components. For example, to increase test diversity, TAUC generates test cases that send multiple messages on the bus thus enabling the detection of faults that slow down the execution of the interrupt handler, which in turn causes a delay in the qualification of errors.

TAUC subsumes the other approaches, i.e. TAUC detects all the faults detected by the random and the manual test suites. However, TAUC does not identify all the faults. Since, in the case of BodySense, TAUC has to rely upon partial oracles that check state variable values only and not the complete system state also captured by active locations. Such information, in embedded systems, is often not observable, as discussed in section IX. The faults not covered by TAUC are caused by the CTL operator, which leads to mutated edges that correctly update system variables but bring the system into a wrong state. TAUC test cases always exercise the fault, i.e. cover the mutated edge, but only the test cases that verify additional system behaviors after reaching the faulty state can detect the fault (some of TAUC test cases do this by sending additional inputs to the system and by detecting that the system response is not the expected one). Test cases that terminate just after reaching the faulty state instead cannot fail. Manual and random test cases suffer from the same problem.

Testing Cost. In the case of TAUC, testing cost depends on the effort required for producing timed automata, use case specifications, and mapping tables.

To perform the experiment, there was no additional cost associated with writing use case specifications as they were already produced for communication purposes and to perform functional testing with UMTG.

We supported software engineers in the design of 25 timed automata: three environment automata (with a total of 6 locations, 11 edges, and 8 guard conditions) and 22 requirement automata that model error conditions. Note that modelling of requirement automata is simplified by the fact that the different errors are managed in similar ways (all the automata match the same template, which has 4 locations, 10 edges, 2 guard conditions and 2 scenarios events). Such numbers suggest that the complexity of the models required by TAUC can be managed by experienced software engineers. In addition, given that timed automata provide a clear representation of timing requirements, usually spread across textual specification documents, engineers agreed that their value go beyond the benefits provided by automatic test generation.

XI. Related Work

Techniques that support testing of software timeliness include model-based approaches, and approaches based on meta-heuristic search.

Model-based approaches for testing software timeliness often rely upon timed automata [1]. Most test generation approaches rely upon coverage strategies adapted from traditional finite state machine testing [28], [29]. Other approaches adopt model checkers for test generation. In these cases, the model is typically annotated with auxiliary variables or automata that enable the formulation of the testing purpose or the coverage criterion as a reachability problem [1], [19]. Finally, some approaches rely upon coverage criteria explicitly defined for guard conditions over clock variables [3], [4]. The main limitation of these approaches is that they require complete models, i.e. models that contain all the information required to identify the inputs to be sent to the system in order to trigger state transitions. Without complete models, model-based approaches can be used to generate abstract test cases only, which then need to be concretized through test adaptation, or test transformation approaches [6]. TAUC instead requires minimal modelling of the timing properties and no additional adaptation or transformation layers. It further relies upon use case specifications, which are commonly used in many domains for communication purposes, to generate the test inputs that lead to targeted state transitions.

Other approaches rely upon other formalisms such as UML Statecharts [5], Extended Finite State Machines [30], and Attribute Event Grammars [31], but they share the same practical limitations mentioned above with approaches based on timed automata.

Metaheuristic search has been successfully applied to testing deadline misses and computing worst case execution time. Di Alesio et al. combine genetic algorithms and constraint programming to identify scenarios in which the tasks execution time is close or breaks the deadline [12]. TAUC has a different goal in a different context, i.e. the system testing of timing requirements, expressed as timed automata, based on information available in use case specifications.

Iqbal et al. [11] use search-based algorithms to stress test real-time software by repeatedly executing the software while simulating the environment. Their goal, which is quite different from ours, is to reach critical error states. Another main difference is that TAUC does not require the execution of test cases, thus being more suitable when test cases execution is expensive or the environment cannot be fully simulated.

A few approaches use genetic algorithms to generate test cases specifically targeting timing properties specific to interrupt handlers [32], [33]. In contrast, TAUC is generally applicable to all systems whose functionality is captured as use case specifications.

XII. Conclusion

In this paper we presented TAUC, a model-driven approach, inspired by industrial practice, to automate the system testing of software timeliness properties in the context of use-case driven development, when system requirements are expressed as use cases to communicate with clients and other stakeholders. In addition to use case specifications, TAUC relies on timed automata to capture timing requirements and relevant environment properties. It generates effective test cases by means of a meta-heuristic search algorithm that maximizes test suite diversity. Our objective is to minimize modelling overhead while enabling effective test generation in contexts where it cannot be guided by test execution results, e.g., embedded systems.

TAUC processes use case specifications to identify the test inputs that fire the state transitions in timed automata. It automatically identifies dependencies between use case specifications and timed automata, and captures them in timeliness test models that are also timed automata, thus enabling the use of UPPAAL for test automation. Timeliness test models are fed into UPPAAL to generate test cases that guarantee a basic coverage of edges with timing requirements.

To generate test cases that effectively stress timing requirements given a certain budget (test suite size), TAUC uses a meta-heuristic search algorithm that iteratively improve the test cases generated with UPPAAL to build a test suite that maximizes test case diversity, which has shown in past studies to increase fault detection.

Empirical results obtained with an industrial case study show that, given a constant test budget, TAUC is significantly more effective than a manual test suite devised by experienced engineers and random testing.

Acknowledgment

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03) and the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement number 694277). We would like to thank Thierry Stephany and the IEE software engineers for their support.

REFERENCES

[1] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, R. Hierons, J. Bowen, and M. Harman, Eds. Springer Berlin Heidelberg, 2008, vol. 4949, pp. 77–117.

[2] A. En-Nouaary, R. Dssouli, and F. Khendek, "Timed wp-method: testing real-time systems," *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1023–1038, Nov 2002.

[3] M. S. AbouTrab, M. Brockway, S. Counsell, and R. M. Hierons, "Testing real-time embedded systems using timed automata based approaches," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1209 – 1223, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121212003391

[4] A. En-Nouaary and A. Hamou-Lhadj, "A boundary checking technique for testing real-time systems modeled as timed input output automata (short paper)," in *Proceedings of the 2008 The Eighth International Conference on Quality Software*, ser. QSIC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 209–215. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2008.53

[5] T. Mücke and M. Huhn, "Generation of optimized testsuites for uml statecharts with time," in *Testing of Communicating Systems*, ser. Lecture Notes in Computer Science, R. Groz and R. Hierons, Eds. Springer Berlin Heidelberg, 2004, vol. 2978, pp. 128–143.

[6] M. Utting and B. Legeard, *Practical Model Based Testing*. Morgan Kaufmann Publishers, 2006, ch. 8.

[7] M. Kaplan, T. Klinger, A. M. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, "Less is more: A minimalistic approach to UML model-based conformance test generation," in *1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, 2008, pp. 82–91.

[8] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "Test case generation from natural language requirements based on SCR specifications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*, 2013, pp. 1217–1222.

[9] A. L. L. de Figueiredo, W. L. Andrade, and P. D. L. Machado, "Generating interaction test cases for mobile phone systems from use case specifications," *SIGSOFT Software Engineering Notes*, vol. 31, no. 6, pp. 1–10, 2006.

[10] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771812

[11] M. Z. Iqbal, A. Arcuri, and L. Briand, "Empirical investigation of search algorithms for environment model-based testing of real-time embedded software," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 199–209. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336777

[12] S. Di Alesio, S. Nejati, L. Briand, and A. Gotlieb, "Combining genetic algorithms and constraint programming to support stress testing of task deadlines," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2015.

[13] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems," in *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, ser. Lecture Notes in Computer Science, no. 1066. Springer–Verlag, Oct. 1995, pp. 232–243.

[14] "IEE sensing solutions," http://www.iee.lu.

[15] T. Yue, L. C. Briand, and Y. Labiche, "Facilitating the transition from use case models to analysis models: Approach and experiments," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, 2013.

[16] C. Larman, *Applying UML and Patterns:An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall Professional, 2002.

[17] "The Object Constraint Language (OCL)," http://www.omg.org/spec/OCL/.

[18] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, Apr. 1994. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[19] A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou, "Time-optimal real-time test case generation using uppaal," in *Formal Approaches to Software Testing*, ser. Lecture Notes in Computer Science, A. Petrenko and A. Ulrich, Eds. Springer Berlin Heidelberg, 2004, vol. 2931, pp. 114–130.

[20] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, p. 6, 2013.

[21] V. LEVENSHTEIN, "Binary codes capable of cor- recting spurious insertions and deletions of ones." *Probl. Inf. Transmission*, vol. 1, pp. 8–17, 1965.

[22] Micrium Embedded Software, "Micro c/os," https://www.micrium.com, visited in 2016.

[23] ISO, "ISO-26262: Road vehicles – functional safety."

[24] C. Wang, F. Pastore, A. Goknil, L. C. Briand, and Z. Iqbal, "Umtg: A toolset to automatically generate system test cases from use case specifications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 942–945.

[25] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou, *Time-Optimal Real-Time Test Case Generation Using Uppaal*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 114–130.

[26] M. S. Aboutrab, M. Brockway, S. Counsell, and R. M. Hierons, "Testing real-time embedded systems using timed automata based approaches," *J. Syst. Softw.*, vol. 86, no. 5, pp. 1209–1223, May 2013.

[27] B. K. Aichernig, F. Lorber, and D. Ničković, *Time for Mutants — Model-Based Mutation Testing with Timed Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 20–38.

[28] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp. 178–187, May 1978. [Online]. Available: http://dx.doi.org/10.1109/TSE.1978.231496

[29] R. Cardell-Oliver and T. Glover, "A practical and complete algorithm for testing real-time systems," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, A. Ravn and H. Rischel, Eds. Springer Berlin Heidelberg, 1998, vol. 1486, pp. 251–261. [Online]. Available: http://dx.doi.org/10.1007/BFb0055352

[30] M. Zheng, V. Alagar, and O. Ormandjieva, "Automated generation of test suites from formal specifications of real-time reactive systems," *Journal of Systems and Software*, vol. 81, no. 2, pp. 286–304, Feb. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2007.05.009

[31] M. Auguston, J. B. Michael, and M.-T. Shing, "Environment behavior models for scenario generation and testing automation," in *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, ser. A-MOST '05. New York, NY, USA: ACM, 2005, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083284

[32] J. Regehr, "Random testing of interrupt-driven software," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 290–298. [Online]. Available: http://doi.acm.org/10.1145/1086228.1086282

[33] T. Yu, W. Srisa-an, M. B. Cohen, and G. Rothermel, "Simlatte: A framework to support testing for worst-case interrupt latencies in embedded software," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST'14)*. IEEE, 2014, pp. 313–322.