Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies

Muhammad Zohaib Iqbal, Andrea Arcuri, Lionel Briand

Simula Research Laboratory, P.O. Box 134, Lysaker, Norway
{zohaib, arcuri, briand}@simula.no

Abstract. The behavior of real-time embedded systems (RTES) is driven by their environment. Independent system test teams normally focus on black-box testing as they have typically no easy access to precise design information. Black-box testing in this context is mostly about selecting test scenarios that are more likely to lead to unsafe situations in the environment. Our Model-Based Testing (MBT) methodology explicitly models key properties of the environment, its interactions with the RTES, and potentially unsafe situations triggered by failures of the RTES under test. Though environment modeling is not new, we propose a precise methodology fitting our specific purpose, based on a language that is familiar to software testers, that is the UML and its extensions, as opposed to technologies geared towards simulating natural phenomena. Furthermore, in our context, simulation should only be concerned with what is visible to the RTES under test. Our methodology, focused on black-box MBT, was assessed on two industrial case studies. We show how the models are used to fully automate black-box testing using search-based test case generation techniques and the generation of code simulating the environment.

1. Introduction

Real-Time Embedded Systems (RTES) are largely used in critical domains where high system dependability is required and expected. The basic characteristic of RTES is that they react to external events within certain time constraints. Extensive testing of such systems is important in order to verify their correct behavior under different timing constraints and adverse situations of the *environment* (or context). It is also important to verify that the system under test (SUT) does not lead the environment to a hazardous state. Testing RTES is particularly challenging since they operate in a physical environment composed of possibly large numbers of sensors and actuators. There is usually a great number and variety of stimuli with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. Testing all possible sequences of stimuli/events is not feasible. Hence, systematic testing strategies that have high fault revealing power must be devised. Manually writing appropriate test cases for such complex systems would be a far too challenging and time consuming task. If any part of the specification of the RTES changes during its development, a very common occurrence in practice, then the expected output of many test cases would potentially need to be recalculated manually. Automated test-generation and the use of an automated oracle are essential requirements when dealing with complex industrial RTES.

Moreover, testing the RTES in the real environment usually entail a very high cost and in some cases the consequences of failures would not be acceptable, for example when leading to serious equipment damages or safety concerns. In many cases the hardware, e.g., sensors and actuators, is not yet available at the time of testing as software and hardware are typically developed concurrently in RTES development. Since testing RTES on the real environment is not a viable solution, the use of a simulator is a common alternative.

In our work, we address the above issues by devising a comprehensive, practical methodology for black-box, model-based testing (MBT). The main contributions of this paper are as follows: It provides an environment modeling methodology based on industrial standards and targeted at MBT, and evaluates it on two industrial case studies. The models describe both the structural and behavioral properties of the environment. Given an appropriate level of detail, defined by our methodology, they enable the automatic generation of the environment simulator. The models can also be used to generate automated test oracles. These could, for example, be invariants and error states that should never be reached by the environment during the execution of a test case. Moreover, the models can further be used to automatically choose test cases. Sophisticated heuristics to choose appropriate test cases are automatically derived from the models without any intervention of the tester. To summarize, the only required artifacts to be developed by testers is the environment model and the rest of the process is expected to be fully automated. This paper focuses on how to make environment modeling as easy as possible for the purpose of supporting black-box, MBT, and shows its use for test automation. Due to space constraints, we only briefly discuss the details for code generation.

To support environment modeling in a practical fashion, we have selected standard and widely accepted notation for modeling software systems, the UML and its standard extensions. We use the MARTE [1] extensions for modeling real-time features and OCL for specifying constraints. We have also provided lightweight extension to UML to make it more useful in our context. As we will discuss later, environment modeling is not a new concept. But, most of the approaches use non-standardized notations or grammars for modeling, which makes them difficult to apply from a practical standpoint. To the best of our knowledge, modeling the environment of industrial RTES systems using a combination of UML, MARTE, and OCL has not been addressed in the literature. By using the proposed methodology, the software testers (who are primarily software engineers) can model the environment with a notation that they are familiar with and at a level of precision required to support automated MBT.

The importance of selecting standards for modeling was highlighted by the application of methodology on the two industrial case studies that belonged to completely different domains. An alternative to using standard notations for modeling could have been to create a domain specific language (DSL) for environment modeling. Since the methodology needed to be generic for RTES irrespective of their application domain, making a DSL was not feasible. Making a DSL would have also reduced the benefits that we obtained from using standards and could have only been justified if existing standards did not fit our needs. Our case studies were developed using Enterprise Architect and IBM Rational Software Architect, though any of the widely available UML tools could have been used for this purpose.

The rest of the paper is organized as follow. Section 2 discusses the related work on environment modeling and testing based on environment models. The environment modeling methodology and simulation is discussed in Section 3. Section 4 describes the use of the environment modeling methodology for automated testing. Section 5 discusses the case studies on which the methodology was applied on and finally Section 6 concludes the paper.

2. Related Work

In an early work on environment modeling, Zave and Yeh [2] discuss the environment modeling of embedded systems for requirements engineering purposes. More recently, Ubayashi *et al.* [3], Burmeister [4], and Petit and Street [5] discuss the importance of modeling the environment of an RTES to completely understand its functionality. Kishi and Noda [6] present an approach for modeling the environment of an embedded system using an aspect-oriented modeling technique. Karsai *et al.* [7] propose a language, Embedded System Modeling Language for modeling the environment of an embedded system. Choi *et al.* [8] use annotated UML class and sequence diagrams for modeling and simulation of environment. Kreiner *et al.* [9] present a process to develop environment models for simulation of automatic logistic systems and its environment. Axelsson [10] evaluates how UML can be used to model real-time features and provides extension to UML for modeling of real-time systems and their environments. Gomaa [11] discusses the use of a context diagram for modeling the environment of real-time systems. The context diagram only shows the relationship between the system and its external entities. Friedentahl *et al.* use the concept of SysML block diagram and activity diagrams to represent the system and its interfaces with environment components [12].

There are a few works reported in literature that discuss testing based on the environment of a system. Auguston *et al.* [13] discuss the development of environment behavioral models using Attributed Event Grammar for testing of RTES. Bousquet *et al.* [14] present an approach for testing of synchronous reactive software by representing the environmental constraints using temporal logic. Larsen *et al.* [15] propose an approach for online testing of RTES based on time automata and environmental constraints. Heisel *et al.* [16] propose the use of a requirement model and an environment models using UML state machines along with the model of the SUT for testing. These models are used to send stimuli to the SUT for testing. Adjir *et al.* [17] discuss a technique for testing RTES based on the model of the system and model of intended assumptions in the environment in Labeled Prioritized Timed Petri Nets.

As discussed above, there are approaches in literature that deal with modeling the environment of a system for various purposes. Most of these approaches are only limited to modeling the static structure of the environment, as they do not focus on test automation. The approaches that deal with modeling of behavioral aspects either use notations with which the software engineers are not familiar, or provide extensions for environment modeling that do not have well-defined semantics. Moreover, the properties of the environment, like its timeliness and non-determinism are not modeled in a standard way. Well-defined semantics (or at least abstract syntax) are essential if the environment model is to be compatible with other standard techniques available for model manipulation, e.g., model transformations, consistency checking. All environment modeling approaches aimed at supporting testing, except by Heisel *et al.* [16], use non-standard languages for modeling. Heisel *et al.* models both the SUT and the environment, which does not fit our purpose: black-box, system testing. Moreover, they model the concepts of probabilities and time using non-standard notations, without using the UML extension mechanisms. Last but not least, none of the relevant work assesses their environmental methodology on an actual RTES system, which we believe is a requirement to assess the credibility and applicability of any MBT approach.

3. Environment Modeling - Methodology

If environment models are to be used for RTES, they should not only be sufficiently detailed, but should also be easy to understand and modify as the environment and RTES evolve. To handle the complexity of realistic RTES environments, the modeling language should have provision for modeling at various levels of abstraction. The modeling language should also have well-defined syntax and semantics for the tools to analyze the models and for the humans to accurately understand them. The language should also provide features (or allow possible extensions) for modeling real world concepts, real-time features, and other concepts, such as non-determinism, required by the environment components. The UML, MARTE profile, and the OCL together fulfill the important requirements of an environment modeling language.

Even though we are using the same notations to model the environment that are used for modeling software systems, it is important to note that the methodology for environment modeling is significantly different from system modeling. While modeling for the industrial cases, we abstracted the functional details of the environment components to an extent that only the details visible to the SUT were included. For environment behavior modeling, non-determinism is widely used, which is not nearly as common when modeling the internal behavior of a system.

For testing the system based on its environment, the behavior details of the environment are as important as its structural details. Structural details of the RTES environment are important to understand the overall composition of the environment (e.g., number and configuration of sensors/actuators), the characteristics of various components, and their relationships. We choose to model these details in the form of a *Domain Model* developed using UML class diagrams. The behavioral details of environment components are required to specify the dynamic aspects of the environment, for example, to determine the possible environment states, before and after its interactions with the SUT, and to specify the possible interactions between the SUT and its environment. For behavioral details, we used the UML State Machines augmented with the MARTE profile.

In the following subsections, we discuss the methodology for modeling the environment of an RTES. We also discuss various guidelines based on our experience of applying the methodology on two industrial case studies.

3.1. Modeling Structural Details as Environment Domain Model

The environment domain model provides information of the components of the environment, their characteristics, their relationships with one another and the SUT, and information regarding signal sending and reception. The various components modeled in the domain model together form the overall environment of the SUT. This means that all these components (their instances) will run in parallel with each other. Each component in the domain model can have a number of instances in the RTES environment. The information about the number of possible instances of a component in the environment is modeled as cardinalities on the associations between different components in the domain model. Therefore, the domain model can be used to obtain a number of potential configurations of the environment. Fig. 1 shows the partial domain model for the environment of a sorting machine (named as *SortingBoard* in the figure) in automated bottle recycling systems. The model shows various motors, sensors, mechanical devices taking part in sorting, and other systems the *SortingBoard* communicates with.

Note that the domain model that we develop is different from the ones commonly discussed in literature (e.g., [18]). The components represented as classes in the environment domain model will not necessarily relate to software classes. They may correspond to systems, users and concepts related to various natural phenomena. Domain modeling here is not a starting point for software analysis. The identification of components in the domain model, their properties, and their relationships is also different from what is commonly done for software analysis. Following, we further discuss various guidelines for modeling the structural details of a RTES environment.

Environment Components to be Included. Initially, all the environment components that are directly interacting with the SUT are included in the domain model. Then, each of these components is further refined to a level where we are certain to cover the important details for simulating the environment needed to test the SUT. If at any time the behavior of an environment component was getting too complex, when possible, we decomposed the component and divided its behavior into multiple concurrent state machines. This is especially useful if a component can be divided into components that are similar to existing components, so that we can specialize existing state machines. We used the stereotype <<*context*>> to represent components of the environment in the domain model. The components of the environment are made to communicate with each other and the SUT through signals, and are modeled as active objects.

Relationships to be Included. All those associations representing the physical or logical relationships among various environment components, or that were needed for components to communicate, should be included. A number of components in the environment might be similar to each other (e.g., various types of sensors). It is useful to relate these components (and their behavior) using the generalization/specialization relationship for simplifying the model, as our experience shows that such domain models get highly complex. For example, in the sorting machine case study, we modeled the association of the *SortingBoard* with the *SortingArm*, which is controlled by the board, and the *ItemSensor* that reports arrival of an *Item* (e.g., bottle). We used generalization in multiple places, including motors and sensors as shown in Fig. 1.

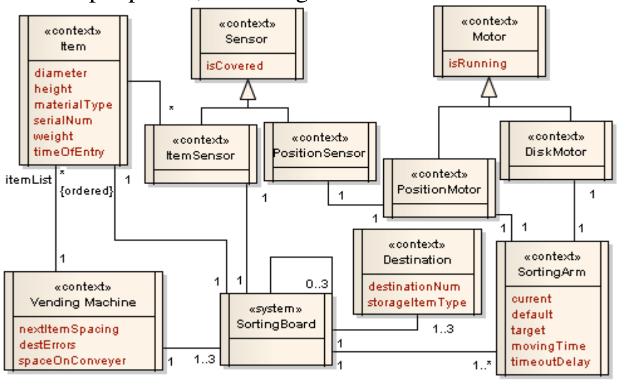

Fig. 1. Partial environment domain model showing properties and relationships of the sorting machine case study

Properties to be Included. From all properties that may characterize environment components, it is important to include only those properties that are visible to the SUT (or have an impact on a component that is visible to the SUT). These may include attributes that have a relationship to the inputs of the SUT, that constrain the behavior of a component with respect to the SUT, or that contribute to the state invariant of a component that is relevant to the SUT. In Fig. 1, all those modeled properties of *Item* are either visible to the *SortingBoard* or are used by other components. For example, the *serialNum* and *materialType* of *Item* is assigned by *VendingMachine* and is used by the *SortingBoard*.

Modeling the SUT. It is important to include the SUT in the environment domain model, so that its relationship with the other environment components can be specified. It is also useful to include the details of signal receptions by the SUT from other environment components. The SUT is stereotyped as <<*system*>>. The stereotype was used initially by Gomaa [11] to refer the system in a context diagram. The SUT modeled in the domain model should represent the SUT and its execution platform, as a single component.

3.2. Modeling Behavioral Details with UML State Machines & MARTE

For modeling the behavior details of the environment that have an impact on the SUT, we developed the UML State Machines with MARTE real-time extensions for various components in the environment. As discussed earlier, the environment components run in parallel to form the environment of the RTES. The components can send signals to each other and to the SUT. We can also view the environment as having one state machine with orthogonal regions, one for each component. Fig. 2 shows the state machine of a component for one of the industrial case studies. We have abstracted out the concepts for confidentiality reasons. Following, we discuss the details of the methodological guidelines we followed.

Identifying Stateful Components. Components whose states either affect the SUT or are affected by the SUT should be modeled with state machines. Apart from these components, it is also useful to model the behavior of other components on which we would like control during the simulation.

Overall, the environment should be modeled in a way that enables, after the initial configuration and provision of input data (parameters and guards), the full simulation of the interactions with the SUT. All the context components shown in Fig. 1 are stateful components of the sorting machine case study. For example, the *SortingArm* component was modeled as stateful since it receives signals from the *SortingBoard* and reacts differently based on its current state.

States to be Included. It is important to determine the right level of abstraction for a component state machine. If we want to precisely model the behavior of an environment component, this might lead to a large number of states. We are, however, only interested in state changes that have an impact on the SUT. A single state in an environment model state machine may correspond to a large number of concrete or physical states. For example, in the sorting machine, the *Item* states that were modeled were all related to its movement through the sorting machine whereas its other possible states were not of interest as an environment component of the *SortingBoard*.

Modeling Users in the Environment. Generally, for software system modeling users are only modeled as sources of inputs and data. The behaviors of users with respect to the system are not considered. In the environment modeling methodology, it is useful to model the behavior of users in the environment to have a control over the inputs/outputs of the various components or the SUT. If a user participates in multiple roles, it is useful to model each role a user plays as a separate component. In the sorting machine case study, we modeled two different users (the operator and the persons who enters the items for sorting), each of them had considerable non-deterministic behavior.

Modeling Abstract Phenomena. Sometimes it is necessary to model abstract physical concepts, such as temperature, heat, voltage, and current. Mostly, information regarding these phenomena can be obtained and controlled through sensors and controllers, such as a temperature controller or sensor. Modeling of such concepts explicitly as environment components can be useful if a change in the state of these concepts impacts multiple components simultaneously, or if it is not possible to identify a related component in the environment that can act as a controller or sensor of this concept for simulation. As an example, consider a RTES on a vehicle that indicates its driver the time for a pit stop. The tires of a vehicle can burst when the temperature of the road gets too high. If there is no sensing mechanism available in the environment, then it is useful to make a state machine of temperature, with possibly two states representing below and above danger temperatures.

Modeling Transitions & Action Durations. Most of the transitions in the state machines of the components will either be based on signal events or time events. Timeout transitions are an important concept in RTES environment models. The

MARTE *TimedEvent* concept is used to model timeout transitions, so that it is possible for them to explicitly specify a clock. Each environment component may have its own clock or multiple components may share the same clock for absolute timing. The clocks are modeled using the MARTE's concept of clocks. Specifying a threshold time for an action execution or for a component to remain in a state is possible using the MARTE *TimedProcessing* concept. This is also a useful concept and can be used, for example, to model the behavior of an environment component when the RTES expects a response from it within a time threshold. When a *SortingArm* is signaled to move, after staying some time in the *Moving* state, it transitions to the *Not Moving* state Fig. 2.

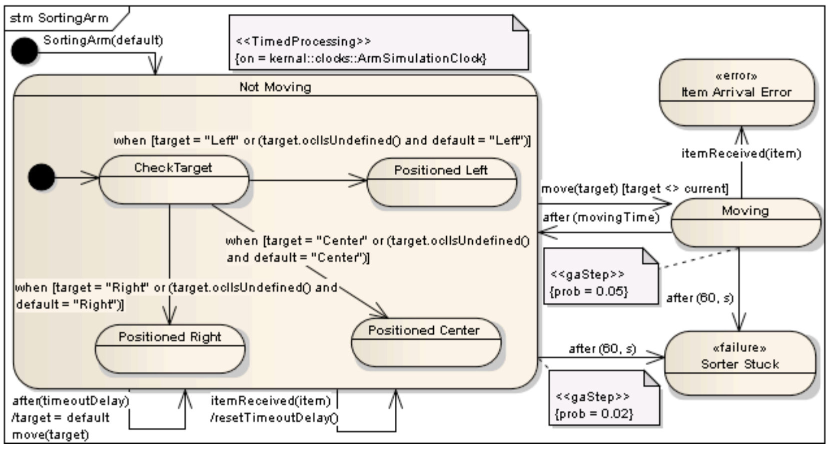
stm SortingArm
SortingArm(default)
<<TimedProcessing>>
{on = kernal::clocks::ArmSimulationClock}
Not Moving
when [target = "Left" or (target.oclIsUndefined() and default = "Left")]
CheckTarget
Positioned Left
when [target = "Center" or (target.oclIsUndefined() and default = "Center")]
when [target = "Right" or (target.oclIsUndefined() and default = "Right")]
Positioned Right
Positioned Center
after(timeoutDelay) /target = default move(target)
itemReceived(item) /resetTimeoutDelay()
«error» Item Arrival Error
itemReceived(item)
move(target) [target <> current]
after (movingTime)
Moving
<<gaStep>> {prob = 0.05}
after (60, s)
after (60, s)
«failure» Sorter Stuck
<<gaStep>> {prob = 0.02}

Fig. 2. State Machine of the *SortingArm* component in the sorting machine case study

Modeling Non-Determinism. Non-determinism is a particularly important concept for environment modeling and is one of the fundamental differences between models for system modeling and models for environment modeling. Following we discuss different types of non-determinism that we have modeled for our case studies.

Specifying exact value for timeout transitions might not always be possible for RTES environment components. To model their behavior in a realistic way, it is often more appropriate to specify a range of values for a possible timeout, rather than an exact value. Moreover, the behavior of humans interacting with the RTES is by definition non-deterministic. For modeling this behavior, we can add an attribute in the environment component and use OCL to constrain the possible set of values of the attribute and then use this attribute as a parameter of a timeout transition. In the sorting machine case study, the *SortingArm* may reach a sorting location from its center between 5 sec and 6 sec, depending on various physical conditions. This is modeled through the attribute *movingTime*, which is passed as a parameter to the change event on the transition from *Moving* to *Not Moving*. Legal values for the attributes are constrained using OCL.

Another important form of non-determinism is to assign probabilities to the transitions of state machines. In an RTES environment, we sometimes only know the probability of a component to go into a particular state over time and we are not sure about the exact occurrence of such conditions. For example, we can say that the probability of a car engine to overheat after running continuously for 10 hours is 0.05, but we cannot be certain about the exact instance in time when this situation will happen. We can model this in the engine state machine with a transition going from *Normal Temperature* state to *Overheated* state, during an interval of 10 hours, with probability of 0.05. For modeling these scenarios, we assigned a probability on the transitions using the property *prob* of the MARTE *GaStep* concept. Whenever a timeout transition has the *gaStep* stereotype applied with a non-zero value of *prob*, the combination will be comprehended as the probability of taking the transition over time of timeout transition. In the sorting machine case study, a *SortingArm* can get stuck in a position (e.g., because of a bottle blocking it or the arm jamming) with a probability 0.02 in a minute if it is not moving and a higher probability when it is moving. This can be modeled as shown in Fig. 2 by the transitions from *Not Moving* and *Moving* to *Sorter Stuck*. The sending of non-deterministic signals can also be modeled using this type of transitions, by placing them in the actions of such transitions.

Another type of probability that we modeled in our case studies is for the situations where one event can lead to multiple possible scenarios, but all of them are mutually exclusive. For example, if we want to represent the fact that during the communication with the SUT (e.g., UDP), there is a chance that a signal from it is received without distortion or with distortion. To make the models more realistic, we assigned probabilities to each of such scenarios in the environment component. In terms of UML state machines, this means that multiple transitions are outgoing from one state based on the same event (maybe with identical guard). For modeling these scenarios, we assigned the MARTE *gaStep* stereotype to each of the multiple possible outgoing transitions. The example of communication with the SUT can be modeled by having two transitions going out of the environment component state on receiving of a signal, one labeled with a probability that the signal was corrupted and the other with the probability that the signal was fine. Modeling the distribution of event arrivals and timeout transitions can be useful for validation purposes, but is out of the scope of this paper, since our goal is verification of the SUT. Nevertheless, this type of information can be easily expressed in the model using the MARTE profile.

Modeling Error & Failure States. In the environment models, two types of states play a particularly important role: the *error* states and the *failure* states.

Environment error states are those states that the environment goes into because of unwanted response(s) (or lack of) from the SUT. Every component in the environment may have error states. If any component of the environment reaches one of these error states, then it means that the SUT is faulty. We use the stereotype <<*error*>> for such states in the environment model. For a *SortingArm*, an *Item* should not arrive while the arm is moving. This is an error state of the environment and can be caused if arm is not made to move on time by the *SortingBoard*. In Fig. 2, this has been modeled with the *Item Arrival Error* state.

Failure states model possible failures of environment components. A component may fail in several different ways with different consequences for the SUT. The SUT should appropriately behave under known, failing conditions. A failure can happen at any time during the execution of a component, e.g., a sensor may break at any time, and is modeled as non-deterministic behavior (as discussed). We use the stereotype <<*failure*>> for these failure states. The *Sorter Stuck* state discussed earlier, in which the *SortingArm* is stuck and cannot change its position, is a failure state of the environment.

3.3. Modeling the Constraints

To apply constraints on the relationships and restrictions on various value combinations (or state combinations) of objects, we have used the Object Constraint Language (OCL). We have also used OCL for representing the guards on the state machines, various state invariants and general constraints on the relationships of environment components.

An RTES environment consists of a number of components including some real-world concepts (e.g., temperature, air pressure). If we consider all the various components of environment together, it is important to restrict the possible state combinations of these components to avoid infeasible situations (e.g., reverse and forward movement of motors is not possible at the same time). In our methodology, we have used OCL to specify constraints for such scenarios. For example, for the sorting machine, if a *SortingArm* is moving then only one *DiskMotor* and *PositionMotor* should be running at a given time. If the arm is not moving, both the motors should not be running. There can be a number of such constraints and it is important to model them to have a realistic simulation and testing based on the models. Otherwise, the models would end up in states that are not practically possible.

State invariants in the environment also play a significant role. Based on the values of the attributes of the component, the state invariants are used to evaluate the current state of the environment and derive state oracles (i.e., is the environment in the expected state?). We have used OCL to specify the state invariants. We also used OCL to specify the overall set of values that an attribute of an environment component can take. Last, the OCL constraints were also used for modeling non-determinism as discussed earlier.

3.4. Environment Modeling Profile

Our goal was to model the environment based on the standard UML and its existing extensions as much as possible. We applied the standard notations and based on our needs for those case studies, where required, we provided light weight extensions to UML. In this section we will discuss the subsets of UML and MARTE that we used and the lightweight extensions that we have provided for environment modeling. From a practical standpoint, it was important to identify these subsets for the methodology, since the UML and MARTE standards are very large and most organizations would be reluctant to adopt such large notations.

We used the concept of Context, System, Error, and Failure under the form of UML stereotypes. Context is used to represent an environment component and is applied on the classes of the domain model. Similarly, System is also applied on the classes of the domain model and represents the SUT. Error represents the states of environment component that are only taken if there is an error in the SUT. Failure is also applied on the states and represents a failure in the environment. Within UML, we used the concept of Class diagram, State Machines. From MARTE, we only used the Time package and the *GaStep* concept from the GQAM package as shown in Fig. 3. This small subset of UML and MARTE was sufficient for modeling our two industrial case studies for the purpose of automating black-box testing.

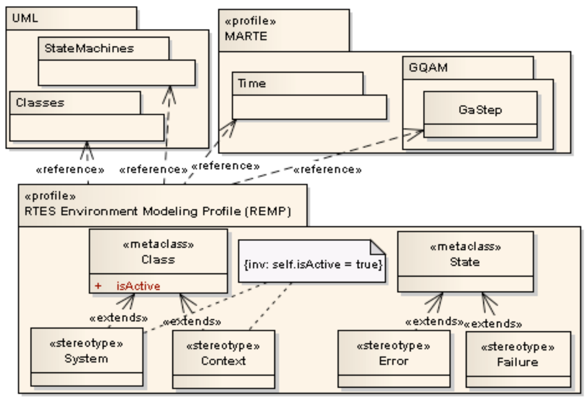

Fig. 3. Profile diagram showing various stereotypes and references

3.5. Simulation of Environment Models

Due to size constraints, we cannot go into the details of the simulation and only briefly discuss it. An environment simulator can be used to simulate the RTES environment. Such a simulator is generally used to test a RTES in conditions similar to its real environment. The environment models developed using our methodology with UML and the MARTE profile are transformed into a simulator in Java using a model to text transformation. Since, the standard for a concrete syntax of the UML Action Language is still not finalized, we made use of Java. Once there is a standard UML Action Language, the actions can be written in that language and then translated into the target language of the RTES. For our case studies, the actions are written in Java and are converted into Java method calls.

4. Model-based Testing based on Environment Models

In this section we briefly discuss how our modeling methodology is used to achieve automated system testing. Further details can be found in [19].

The UML/MARTE models of the environment are used to automatically generate a simulator for it. The simulator is used to test the RTES. In our methodology, a test case is the setting used for the simulator. The information of what to configure in the simulator is automatically derived from the models and it is given as input to the test engine. Two types of setting are necessary:

- Number and relations of the environmental components. For example, given a state machine representing a sensor, the Domain Model is used to determine how many sensors can be connected to the RTES (and so, we would know how many running instances we need for that state machine). Several different combinations are possible.

- Each state machine can have non-deterministic events. The models are used to specify them and to provide details of their type. When the simulator is running, every time it requires a value to calculate a non-deterministic event, it then queries the test engine to obtain such values.

At the current moment, we have not investigated different configurations based on the Domain Models. We have focused on testing the behavior of the RTES given a single configuration. The goal of the testing is to provide a valid setting for the non-deterministic events such that an environmental error state (Section 3.2) is reached during the simulation, if any fault is present.

The simplest testing technique would be to provide (valid) random values each time the simulator queries the test engine for values to use in non-deterministic events. But more sophisticated techniques that exploit the information in the models can be used. For example, reaching the error state during simulation can be represented as a search/optimization problem, so Search Based Testing (SBT)[20] can be used. From the models we can automatically generate a *fitness function* to guide the search. Common heuristics such as *approximation level* and *branch distance* of the OCL constraints would be used for the fitness function. Due to size constraints, the investigated testing strategies are reported in [19], where we also proposed a *novel* fitness function that exploits the time properties of the UML/MARTE models.

The use of models for SBT in the case of RTES system testing is essential. In fact, to have effective heuristics (i.e., the fitness function) we need to have precise knowledge of the error states. This information is easily added in the models using stereotypes (Section 3.4). All the relevant states/transitions that lead to those error states can be exploited for the automatic derivation of the fitness function. On the other hand, if we have a simulator but no model, it is unlikely that it would be possible to automatically reverse-engineer all this useful information from the code alone. Therefore, the fitness function would be necessarily written by hand, with all the related downsides that this choice brings.

In some relevant cases [19], it is possible to automatically derive very precise fitness functions. This happens when time constraints need to be satisfied (a typical case in RTES), e.g., a signal should be received within 10 milliseconds. A test case for which that signal is received after 9 milliseconds gives more information than a test case in which the same signal is immediately received after 1 millisecond (notice that in both cases the constraint is satisfied). SBT can automatically exploit this information by focusing the search on simulator configurations that are more likely to yield a deadline miss. A tester does not need to write these heuristics, they are in fact automatically derived from the environment models. This is essential, because in general software testers do not have the expertise to write proper fitness functions for search algorithms.

In the state machine depicted in Fig. 2, the transition to the error state is not based on a time deadline. The error arises if an *Item* arrives while the sorter is in the *Moving* state. Still, heuristics can be automatically derived. For example, we can reward in the fitness function the test cases that lead that sorter to stay in the state *Moving* for as long as possible, such as to increase the possibilities that this erroneous transition will be taken. In general, in a fitness function we would reward test cases that lead the machine to stay longer in these states from which a transition to error states can take place. This information can be automatically derived from the environment models. In our particular example, the search algorithms would be guided by the fitness function towards test cases in which consecutive arriving items have different destinations (so that each time an *Item* is arriving, the *SortingArm* has to move).

The results in [19] show that our modeling methodology can be used for a fully automated system testing that is effective in revealing faults in industrial RTES. Although different testing strategies can be design (e.g., Random Testing and SBT), the environment modeling methodology described here would still remain the same.

5. Case Studies

To evaluate the proposed methodology for environment modeling, we applied it on two industrial RTES. The application domains of the systems were entirely different. Though we cannot provide full details of the systems because of confidentiality restrictions, we are providing a brief description of the systems. One of the systems was a marine seismic acquisition system. One of the responsibilities of the system was to control the movement of seismic cables, where each cable had a large number of sensors and seismic vibrators, among other equipments. The system regularly communicated with these components and was responsible for managing the life cycle and connections for these components (among other things). The second RTES was a sorting system, which was part of a recycling machine. The system communicated with a number of sensors and actuators to guide a recycled item through the recycling machine to its appropriate destination. We provide a summary of the environment models developed for both the case studies in Table 1.

For Case A, the RTES was configurable as three different types of systems; therefore the number of environment components was large. But most of the components' behavior could be modeled with a couple of states. The highest number of states was 18. Many components inherited a parent component behavior, i.e., its state machine. That was the case for example for *DiskMotor* and *Motor* in Fig. 1.

Though the number of components for Case B was more limited than for Case A, the number of instances for some of the components in the environment was very large (e.g., thousands of sensors of the same type communicating with the SUT), thus leading to many instances of executing state machines during simulation. The complexity of component state machines was also on average much higher than for Case A.

Application of the proposed methodology to these case studies for simulator generation and effective automated testing shows the scalability and effectiveness of our approach.

One important conclusion is that, in both cases, we were able to model the RTES environments with the subset of UML and MARTE that we identified and the lightweight extensions that we proposed.

For both case studies, the number of components identified at the time of domain modeling was larger than what was finally required. During successive revisions and

Table 1. Summary of the environment models of the two industrial RTES.

Industry Case	# of env components	Stateful components	Average # of states	Max states in a component	Max transitions in a component
Case A	55	43	~3	18	40
Case B	5	4	~12	19	29

based on insight obtained through behavioral modeling, some components turned out to be unnecessary and were removed from the domain model. One practical challenge is that it was not easy in practice to identify the right level of abstraction to model the behavior of environment components. Sub-machines were widely used to incrementally refine the behavioral models until the right level of detail was achieved to simulate the behavior of component from the viewpoint of the SUT.

6. Conclusion

In this paper, we have discussed a methodology for modeling the environment of a Real-Time Embedded System (RTES) in order to enable black-box, system test automation, which is usually performed by test engineers who are not informed of the design specifics of the RTES. For practical reasons and to facilitate its adoption, the methodology is based on standards: UML, MARTE profile, and OCL for modeling the structure, behavior, and constraints of the environment. We, and this is part of our methodology, made a conscious effort to minimize the notation subset used from these standards. We briefly discussed how the environment models are used to generate automated system test cases and a simulator of the environment to enable testing on the development platform. One advantage is that the methodology also allows more focus on the testing for critical and hazardous conditions in the RTES environment as environment failures and possible error states due to faults in the RTES implementation are explicitly modeled.

We modeled the environment of two industrial RTES in order to ensure that our methodology and the notation subsets selected were sufficient to fully address the need for automated system testing. Our experience showed that was the case.

7. Acknowledgements

The work presented in this paper was supported by Norwegian Research Council and was produced as part of the ITEA 2 VERDE project.

References

1. OMG: UML Profile for MARTE Beta 2. Object Management Group Inc (2008)

2. Zave, P., Yeh, R.T.: Executable requirements for embedded systems. In: Proceedings of the 5th international conference on Software engineering pp. 295 - 304. IEEE Press (1981)
3. Ubayashi, N., Seto, T., Kanagawa, H., Taniguchi, S., Yoshida, J., Sumi, T., Hirayama, M.: A context analysis method for constructing reliable embedded systems. In: Proceedings of the 2008 International Workshop on Models in Software Engineering pp. 57-62. ACM (2008)
4. Burmeister, C.: Real-Time Environment Modeling. In: IEEE Workshop on Real-Time Applications pp. 142-146. (1993)
5. Pettit IV, R.G., Street, J.A.: Lessons Learned Applying UML in the Design of Mission Critical Software. UML 2004 Satellite Activities, Vol. 3297. Lecture Notes in Computer Science, Springer Berlin / Heidelber (2004) 129 - 137
6. Kishi, T., Noda, N.: Aspect-oriented Context Modeling for Embedded Systems. In: Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design pp. 68-74. (2004)
7. Karsai, G., Neema, S., Sharp, D.: Model-driven architecture for embedded software: A synopsis and an example. Science of Computer Programming 73, 26-38 (2008)
8. Choi, K.S., Jung, S.C., Kim, H.J., Bae, D.H., Lee, D.H.: UML-based Modeling and Simulation Method for Mission-Critical Real-Time Embedded System Development. In: IASTED International Conference Proceedings pp. 160-165. (2006)
9. Kreiner, C., Steger, C., Weiss, R.: Improvement of Control Software for Automatic Logistic Systems Using Executable Environment Models. In: EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO pp. 20919. IEEE Computer Society (1998)
10. Axelsson, J.: Unified Modeling of Real-Time Control Systems and Their Physical Environments Using UML. In: Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01) pp. 18. (2001)
11. Gomaa, H.: Designing Concurrent, Distributed And Real-Time Applications With UML. Addison-Wesley Educational Publishers Inc (2000)
12. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. Elsevier (2008)
13. Auguston, M., B, M.J., Shing, M.: Environment behavior models for automation of testing and assessment of system safety. Information and Software Technology 48, 971-980 (2006)
14. Du Bousquet, L., Ouabdesselam, F., Richier, J.L., Zuanon, N.: Lutess: a specification-driven testing environment for synchronous software. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering pp. 267-276. ACM New York, NY, USA (1999)
15. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online Testing of Real-time Systems Using Uppaal. Formal Approaches to Software Testing, Vol. 3395. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2005) 79-94
16. Heisel, M., Hatebur, D., Santen, T., Seifert, D.: Testing Against Requirements Using UML Environment Models. In: Fachgruppentreffen Requirements Engineering und Test, Analyse & Verifikation pp. 28-31. GI (2008)
17. Adjir, N., Saqui-Sannes, P., Rahmouni, K.M.: Testing Real-Time Systems Using TINA. Testing of Software and Communication Systems, Vol. 5826. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2009) 1-15
18. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall PTR Upper Saddle River, NJ, USA (2001)
19. Arcuri, A., Iqbal, M.Z., Briand, L.: Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing. Technical Report, Simula Research Laboratory (2010)
20. McMinn, P.: Search-based Software Test Data Generation: A Survey. Software Testing Verification and Reliability 14, 105-156 (2004)