Softw Syst Model
DOI 10.1007/s10270-013-0328-6

REGULAR PAPER

Environment modeling and simulation for automated testing of soft real-time embedded software

Muhammad Zohaib Iqbal · Andrea Arcuri · Lionel Briand

Received: 5 March 2012 / Revised: 6 February 2013 / Accepted: 26 February 2013

Abstract Given the challenges of testing at the system level, only a fully automated approach can really scale up to industrial real-time embedded systems (RTES). Our goal is to provide a practical approach to the model-based testing of RTES by allowing system testers, who are often not familiar with the system's design but are application domain experts, to model the system environment in such a way as to enable its black-box test automation. Environment models can support the automation of three tasks: the code generation of an environment simulator to enable testing on the development platform or without involving actual hardware, the selection of test cases, and the evaluation of their expected results (oracles). From a practical standpoint—and such considerations are crucial for industrial adoption—environment modeling should be based on modeling standards (1) that are at an adequate level of abstraction, (2) that software engineers are familiar with, and (3) that are well supported by commercial or open source tools. In this paper, we propose a precise environment modeling methodology fitting these requirements and discuss how these models can be used to generate environment simulators. The environment models are expressed using UML/MARTE and OCL, which are international standards for real-time systems and constraint modeling. The presented techniques are evaluated on a set of three artificial problems and on two industrial RTES.

Keywords Environment modeling · Environment simulation · Automated testing · Model-based testing · Real-time embedded systems · Search based software engineering

Communicated by Dr. Juergen Dingel.

This paper is an extension of a conference paper "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies" published in Proceedings of ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), 2010 [1].

M. Z. Iqbal (✉)
Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad, Pakistan
e-mail: zohaib.iqbal@nu.edu.pk

A. Arcuri
Simula Research Laboratory, Fornebu, Norway
e-mail: arcuri@simula.no

L. Briand
SnT Centre for Security, Reliability, and Trust, University of Luxembourg, Walferdange, Luxembourg
e-mail: lionel.briand@uni.lu

1 Introduction

Real-time embedded systems (RTES) are widely used in many different domains, as for example from integrated control systems to consumer electronics. Already 98 % of computing devices are embedded in nature and it is estimated that, by the year 2020, there will be over 40 billion embedded computing devices worldwide [2]. Testing these systems such that they are functionally correct and do not lead their environment into critical states (e.g., unsafe) is vital. RTES environments typically comprise a number of physical components (e.g., sensors and actuators) and possibly other RTES systems (e.g., in systems of systems). Typically, there is a large number and variety of stimuli to the RTES with different patterns of arrival times. These characteristics make the testing of RTES challenging and increase the need for automated, systematic testing strategies.

Because RTES are developed for diverse domains presenting different constraints (e.g., different timing, safety, security requirements), different testing approaches are required to handle the varying set of characteristics required by these domains [3]. Our main target RTES in this paper are soft real-time systems with time deadlines in the order of hundreds of milliseconds with an acceptable jitter of a few milliseconds in response time. Our testing approach (black-box system level testing) not only encompasses functional correctness of the system under test (SUT), but also enable to focus testing on particularly critical aspects of the RTES, i.e., potentially hazardous situations.

Typically, large scale testing of RTES software in real environments and on actual deployment platforms is not a viable option. It would be expensive, the consequences of failures might be catastrophic (e.g., in safety critical systems), and the number of variations in the environment that can be exercised within a reasonable time frame are small. Moreover, some of the environment components might not be available at the time of testing, since hardware and software components are typically developed concurrently. To test RTES software in this kind of situations, a common strategy is to develop a simulator for these environment components.

When testing RTES, the simulation of three concepts (or their combinations) is typically considered: the SUT, its hardware platform, and the environment with which the SUT interacts. Depending on the goal of testing, different combination of these three concepts can be simulated [3]: (i) at early stages of the development process, a typical approach is to model and simulate the SUT, its hardware and its environment to ensure that the specifications of the SUT are not incompatible with the environment assumptions; (ii) the embedded software is tested on the development platform with a simulated environment to ensure that the developed software works correctly and can handle possible environment failures. This is done with either an adapter for the hardware platform that forwards the signals from the SUT to the simulated environment or a simulation of the hardware platform; (iii) another level of simulation is when the actual software is deployed on the hardware platform (or part of the platform, e.g., only the processor) and testing is done with a simulated environment.

The focus of this paper is on the second type (ii) of modeling and simulation in which the actual SUT is used, the environment is simulated, the hardware platform is simulated or bypassed through an adapter communicating with the environment simulator. In our experience of working with two industrial organizations, which were developing RTES for different domains (seismic acquisition systems and automated bottle recycling machines), this form of testing was highly critical as it enabled early verification of the RTES.

To address the above objective, in this paper, we propose an automated methodology for RTES based on environment behavioral models developed using software modeling standards: Unified Modeling Language (UML) [4], UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) [5], and Object Constraint Language (OCL) [6]. The main contributions of this paper include an environment modeling methodology and an approach to generate a simulator of the environment from the environment model in a way to enable the automated testing of industrial RTES. As further discussed below, our focus is to devise a *practical* approach in a *system (level) testing* context, and we evaluate both the modeling methodology and simulation generation on two industrial case studies.

Environment 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. These models can also be used to obtain test oracles, which are typically modeled as "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 select test cases and sophisticated heuristics are used to automatically do so 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. Incidentally, by using this automated Model-Based Testing (MBT) technology, one of our industrial partners was able to find new critical faults in their RTES.

To support environment modeling in a practical fashion, we have selected standard and widely accepted notations for modeling software systems, the UML and its standard extensions. We use the MARTE [5] extensions for modeling real-time features and OCL for specifying constraints. We have also provided lightweight extensions to UML to ease its use 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. 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, using commercial or open source tools, and at a level of precision required to support automated MBT. The importance of relying on standards for modeling was confirmed on the two industrial case studies across entirely different domains.

Although code generation from models has been widely studied, the context of black-box RTES system testing poses specific challenges and problems that are not fully discussed and addressed in the literature. For this purpose, we present extensions to the *state pattern* [7] specifically aimed at enabling environment simulation for system testing and define rules for transforming environment models to Java code (the simulator).

To summarize, the fundamental motivation here is that system testers, in many industry sectors, are usually application domain experts but have a little or no knowledge of the system design and implementation. Our approach is therefore black-box and does not require the RTES itself to be modeled. It only requires its environment to be modeled at the right level of abstraction and in such a way as to enable effective test automation. The reliance on software modeling standards offers significant advantages, such as the possibility of using (1) different commercial and open source modeling tools (e.g., IBM Rational Software Architect (RSA),[1] Papyrus,[2] or Enterprise Architect[3]), (2) notations that many software engineers—including system testers—might already be familiar with and that can be used to also model the SUT, and (3) existing analysis tools (e.g., [8]) that can take such models as input.

The paper is organized as follows: Sect. 2 sheds light on the practical motivations and aspects of the work presented in this paper, setting the context to better justify our approach; Sect. 3 discusses the related work. Section 4 presents the motivating example that we use throughout the paper to explain various concepts. Section 5 discusses the proposed environment modeling methodology. Section 6 goes into the details of the most important decisions regarding the transformation of models to simulation code, whereas Sect. 8 presents the case studies. Section 9 discusses the limitation of the proposed work and finally, Sect. 10 concludes the paper.

2 Practical aspects

The work discussed in this paper was motivated by the problems faced and practices followed by two industrial organizations that we worked with, namely WesternGeco AS, Norway and Tomra AS, Norway. These two organizations were developing RTES for two different domains; WesternGeco was developing a seismic acquisition system and Tomra was developing automated recycle machines. Both the RTES were developed to run in an environment that enforces time deadlines in the order of hundreds of milliseconds with an acceptable jitter of a few milliseconds in response time. In one of the organizations, testing the SUT on the development platform with a simulated environment was considered to be mandatory before deploying the software on the operational hardware. To achieve this, software engineers were writing application specific simulators directly in Java. Test cases for system level testing were written by hand by the software test engineers and were executed on the SUT with the environment simulator. The research presented in this paper was strongly driven by our investigation of the practical needs of our industry partners which, based on our experience, are shared by many others in numerous industry sectors. Our understanding of these needs is presented in the remainder of this section.

Manually writing an environment simulator using a programming language (e.g., Java or C) appeared to pose a number of issues, the main one being that software engineers have to develop such simulator at a low-level of abstraction while simultaneously focusing on the logic of the simulator, complex programming constructs (e.g., multiple threads, handling timers), and the handling of test case configurations (when the simulator is used for testing). Making this problem even more acute, over the course of the RTES development, these simulators frequently change due to changes in the specifications of the hardware components.

Typically, modeling and simulation (M&S) approaches focus on simulating hardware components, execution platforms, and natural phenomena in the RTES environment using various simulation tools, such as DEVS [9] and Modelica [10]. These M&S tools support precise simulations of both discrete and continuous system behaviors and are typically based on mathematical models. However, in our context, such M&S tools are not practical, for a number of reasons: (i) *Software engineers*, who are typically in charge of system testing at this level, are often not familiar with such simulation languages. To enable technology transfer in industrial practice, it would be more convenient for them to develop or generate the simulator using a language that they are familiar with, as for example the languages used to program and model the SUT; (ii) These simulation tools do not support automated environment-based testing of RTES software. A number of features must be modeled to enable this kind of testing. For example, the models need to provide information for the automated generation of oracles (to verify whether test cases pass or fail). Furthermore, the simulator needs to interact with a test harness to get appropriate values for various non-deterministic events. The exact occurrences of such events in the environment cannot be determined. These events may follow different probability distributions (e.g., probability of failure of a sensor) or may occur at any time within a given time interval (e.g., a gate at a railroad intersection may take from 5 to 7 s to close); (iii) Another issue is that in simulation languages, such as ModelicaML [11] and DEVS [12], since they were developed for a different purpose, there are limitations regarding the interactions of the simulator with the production code of the RTES (e.g., handling of operating system resources, such as inter-process communication with the production code over

[1] http://www.ibm.com/developerworks/rational/products/rsa/, accessed on 05/02/2012.

[2] http://www.papyrusuml.org/, accessed on 05/02/2012.

[3] http://www.sparxsystems.com.au/, accessed on 05/02/2012.

TCP/UDP). Such an interaction is a requirement for the type of testing we deal with in this paper, since the environment simulator has to interact with the *actual RTES production code* to receive stimuli and to send responses. In dealing with such interactions, we do not want any constraint regarding the programming language in which the RTES is written.

The modeling methodology presented here provides an automated model-based approach to derive environment simulators, test cases, and test oracle, taking into consideration all the practical aspects described above, which are common place in many industrial environments. The only major input required are the environment models describing the structure and behavior of the environment as well as the test oracles (i.e., the *error* states). Since the intended users are software engineers, we chose standard software modeling languages for environment modeling with the aim to make the modeling methodology as simple as possible. This paper discusses the methodology for modeling the environment based on the selected modeling standards and a specific profile and furthermore describes the process of simulation generation from those environment models. It does not, however, address in detail test generation and test oracles.

3 Related work

In this section, we discuss the related literature in the areas of (1) modeling and simulation for RTES Testing, (2) environment modeling and environment model-based testing of RTES, and finally (3) code generation approaches from UML state machines and class diagrams.

3.1 Modeling and simulation for RTES testing

As discussed earlier, based on the goals of testing of RTES, the SUT, the hardware platform, the environment, or their combinations can be modeled and simulated.

At the early stages of the development process for RTES, a typical approach is to model and simulate the SUT, its hardware and its environment. The aim is to ensure that the model of the SUT complies with the requirement specifications and is compatible with environment and hardware assumptions. This approach is sometimes also referred as "model-in-the-loop" simulation [3,13,14].

Another level of simulation for testing is when the actual executable software is deployed on the real hardware platform (e.g., electronic control unit) and their combination is tested with a simulated environment (e.g., with the simulation of plant model [3]). This approach is generally referred to as hardware-in-the-loop testing [15,16]. Typically, a prototype of the hardware platform is used at this stage. A variation to hardware-in-the-loop testing is the case where only the actual processor is used during testing and rest of the hardware and environment are simulated. This variation is referred to as processor-in-the-loop testing [17].

Before the hardware or the processor is available, the embedded software can also be tested on the development platform (e.g., Linux or Windows-based machine) with a simulated environment and hardware platform. This is typically done to ensure that the developed software is not violating any of the environment assumptions and behave appropriately in hazardous or abnormal situations. This is mostly referred to as software-in-the-loop simulation [3,13].

Existing modeling and simulation languages have been developed and are widely used for the first three types of simulations. In these cases, the environment simulation needs to interact with the actual hardware or its simulation. In such cases, precise simulation of both discrete and continuous phenomena is required and is typically based on mathematical models. The existing simulation approaches (e.g., simulation generation from ModelicaML [11], Simulink [18], DEVS [12]) require the modeler to have sufficient knowledge of the target simulation language, that is different from the languages used to develop (e.g., C/Java) and model (e.g., UML) RTES software. Other languages, such as Mason [19] and SimJava [20], are developed for modeling and simulation of RTES to overcome these constraints. The existing simulation languages do not cater for the type of modeling and simulation that we require for black-box system testing of RTES (as discussed in Sect. 2), such as non-determinism (a common feature of the environment) and test-specific behavior (e.g., modeling of error and failure states to guide search-based testing). Moreover, as discussed in Sect. 2, the target users of our approach are software testers, who are software engineers. We made a conscious effort to select a modeling language that most of the software engineers are already familiar with. Moreover, in our context, it is highly important that we use the same models to generate both the simulator and test cases.

In this paper, we target a slight variation to the typical software-in-the-loop simulation. We only model and simulate the environment and use an adapter for the hardware platform that forwards the signals from the SUT to the simulated environment. Our research problem definition is motivated primarily by the practical needs of our industrial partners (Sect. 2) but it is expected to be relevant in many other industrial environments developing similar RTES. Other approaches that do testing with a similar focus are discussed next.

3.2 Environment modeling and environment model-based testing

In this section, we discuss various environment modeling and model-based testing approaches reported in the literature.

3.2.1 Environment modeling

There are a number of works in literature focusing on environment modeling of real-time or embedded systems. Kishi and Noda [21] present an approach for modeling the environment of an embedded system using an aspect-oriented modeling technique. Karsai et al. [22] propose a new language for modeling the environment of an embedded system. Choi et al. [23] use annotated UML class and sequence diagrams for modeling and simulation of environment. Kreiner et al. [24] present a process to develop environment models for simulation of automatic logistic systems and its environment. Burmeister [25] discusses the importance of environment modeling in order to have separate models for devices and control software of RTES. Ubayashi et al. [26] present a UML profile for context-based requirement analysis. The models developed using the profile identify the relationships between hardware components and the system context. Petit and Street [27] discuss the use of a context diagram to model the interfaces of input devices in a system. Axelsson [28] 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 [29] discusses the use of a context diagram for modeling the relationship between an RTES and its external entities. Friedentahl et al. [30] use the concept of SysML block diagram and activity diagrams to represent the system and its interfaces with environment components.

To summarize, there are works reported in the literature that deal with modeling the environment of a system for various purposes. Most of these works [21–30] do not focus on test automation and hence lack the corresponding modeling constructs required for this purpose (such as modeling of *error* and *failure* states for modeling oracles and failure scenarios, as done in our work). Moreover, approaches discussed in some of these works are only limited to modeling the static structure of the environment [25,27,29].

3.2.2 Environment model-based testing

There are a few works reported in literature that discuss testing of RTES based on its environment. Auguston et al. [31] discuss the development of environment behavioral models using Attributed Event Grammar for testing of RTES. The behavioral models contain details about the interactions with the SUT and possible hazardous situations in the environment. These models are then traversed to obtain various test scenarios. The approach is applied on a simulation of the RTES specifications. Heisel et al. [32] propose the use of a requirement model and an environment model using UML state machines along with the model of the SUT for testing. Adjir et al. [33] discuss a technique for testing RTES based on a model of the system and intended assumptions in the environment in Labeled Prioritized Timed Petri Nets. The works in [34,35] discuss an approach for conformance testing of RTES based on their specifications and environment assumptions. They propose an approach for generating test cases on the fly during testing. The specifications are modeled using UPPAAL timed automata. In [36], the UPAAL automata based testing approach is applied to an industrial case study. Krichen and Tripakis [37] discuss and approach based on timed automata for conformance testing of RTES. Bousquet et al. [38] present an approach for testing of synchronous reactive software by representing the environmental constraints using temporal logic. Another work is presented by Lindlar et al. [14], in which the tester needs to provide annotated FSMs specifying the search space for test cases and a fitness function to assess the outputs of the test cases. Peleska et al. [39,40] discuss the RT-Tester tool that enables SUT and environment modeling for test-case generation. The tool supports modeling in various notations including UML. David et al. [41] discusses an approach for modeling a partially observable SUT using Timed Game Automata for testing. In [42], the approach allows the modelers to capture various usage scenarios using Markov Chain models. These scenarios are then used to generate test-cases.

Among the works that discuss environment model-based testing, a number of approaches do not consider the SUT as a black-box [32,33,38–40]. Keeping the SUT as a black-box is a common approach for system-level testing where the test engineers are not familiar with the SUT internal design and is the one of the requirements in our context as discussed in Sect. 2. A simulator is essential to execute different possible behaviors of the environment based on its interaction with SUT for system testing. Most of the approaches discussed in literature are restricted to generating restrictive sets of interaction sequences with the environment rather than simulating the environment. Generating such test sequences or test scenarios from the environment models, as done by Auguston et al. [31], can only provide a limited coverage of the environment models. A simulator is also essential to simulate the failure behavior of environment components. Our approach allows the simulation of nominal as well as exceptional environment behavior (e.g., hardware breakdown of environment components). In our context, a test case is a setting of the environment simulator. To the best of our knowledge, the environment model-based testing approaches discussed in the literature, except the one by Heisel et al. [32], use non-standard languages for modeling the environment. Even Heisel et al. provide UML extension for modeling the concepts of probabilities and time without relying on UML extension mechanisms. Another major limitation of most of the relevant works is the application of the testing approach is on toy problems or simulations of actual industrial RTES (e.g., [31–35,37]). We believe that application on industrial case(s) is a requirement to assess the credibility and applicability of any MBT approach.

In our previous work, we evaluated and reported [43,44] the fault detection effectiveness of testing strategies based on the models developed using the modeling methodology proposed in this paper. We evaluated Random Testing (RT), Adaptive Random Testing (ART), and Search-based Testing (SBT) with Genetic Algorithms (GA) and (1+1) Evolutionary Algorithm (EA). The results indicate that SBT shows significantly better performance over RT for a number of cases and significantly worse performance than RT for a number of other cases. The testing strategy were further improved by devising a hybrid approach that combined ART and (1+1) EA [45]. The current paper focuses on the environment modeling and simulation aspect of our overall approach.

3.3 Code generation from UML classes and state machines

To the best of our knowledge, there is no reported approach in the literature generates RTES environment simulators from UML models or their extensions. A number of model-based testing techniques do generate test sequences, which contain interactions between the environment and SUT. Generating such test sequences or test scenarios from the environment models, as done by Auguston et al. [31], can only provide a limited coverage of the environment models. On the other hand, generating a simulator, as in our case, allows for a more realistic set of interactions between the environment and the SUT. Such a simulator also enables testing in nominal as well as exception environment situations. As we discussed in Sect. 3.2, though modeling and simulation languages and environments have been proposed, they do not fit our purpose of black-box system testing (Sect. 2).

Generating code from state machines is not a new problem. Even though a number of works are reported in the literature (e.g., [46,47]) and a number of tools are available that generate code from state machines (e.g., SmartState [48], IBM Rhapsody [49]), their purpose is not environment-model based testing. The use of standards is an important requirement for our modeling methodology, as discussed earlier. The modeling standards that we selected for our methodology are supported by a wide range of tools and support is available for training. The existing code generation tools and techniques discussed in the literature are focused on generating system code and not environment simulators for testing. They are not directly applicable for our purpose as already discussed earlier in Sect. 2.

The original state pattern, discussed in [7], provided a design pattern to implement state-driven behavior but did not address a number of important features present in UML 2.x state machines, e.g., concurrency, time events, change events, and actions. A number of extensions for the pattern have been discussed over time to handle missing features (e.g., [50,51]). Most of these extensions are focused on increasing the understanding and usability of the code obtained using the state pattern to support programmers (e.g., [51]) and are not very useful in our context where the code is automatically generated from the models. Chin and Millstein [52] propose an extension to the state pattern for handling state behavior in inherited sub-classes. Holt et al. [53] propose an extension for handling state and transition actions and report its application on an industrial case. Palfinger [54] provides an extension to the state pattern that allows extension of object's behavior at runtime by using a mapper class. In our approach, this was not applicable as we do not require the addition of new behavior during the execution of environment components. The work in [55] extends the state pattern to support hierarchical state machine and time events. We handle time in a similar way, i.e., by using a separate timer class that calls `timeout()` on the context object in case of a timeout. The approach in [55] does not handle parallel regions and change events, does not provide details on handling actions, and does not provide support for the test-related features required by our approach. We provided a number of extensions to the original state pattern in order to meet the needs for RTES environment simulation to support system testing.

There has been some work on generating code from UML/MARTE models. Among them, Qureshi et al. extend MARTE to generate code for Systems-on-Chips (SoC) [56], Rodrigues et al. discuss an approach to generate code in OpenCL for SoCs by using Allocation, Generic Component Model, and Hardware Resource Modeling packages of MARTE [57]. Piel et al. present an approach for generation of SystemC code from MARTE models for simulation of Multiprocessor SoCs [58]. Another work discussing code generation in SystemC from MARTE models is discussed in [59]. Vidal et al. discuss code generation in VHDL from MARTE models [60]. Mraidha proposes an approach to translate MARTE models, specifically RtUnits to Accord [61]. Our approach automates the test oracle by modeling *error* states in the environment, which is a requirement for automated environment model-based testing. The work reported in [62] discusses code generation for multiple clock domains modeled using the Clock Constraint Specification Language (CCSL). The concepts discussed can be used in our simulation approach, but for both our industrial case studies, we only used one clock (which is based on CPU clock, see discussion in Sect. 6.5.2).

We could have used some optimizations to improve the ease of understanding and modification, and cleaner code generation, but these would not have had a large impact as the generated source code is not visible to the end-user and is only provided as an executable archive. Furthermore, there are optimizations in the literature to improve the performance of the generated code (e.g., minimizing the number of running threads to avoid overheads due to context switching). However, in our framework (as we have explained in details throughout the paper), we do not need to optimize performance. The generated simulators are used only for testing purposes, and each test case runs on a different process, lasting from a few seconds to a few minutes (depending on the RTES). As long as the environment simulators can behave as expected (e.g., providing the right stimuli at the right time), this would be sufficient for our testing purposes. This was the case for all our case studies.

3.4 Summary

To summarize, this paper differs from existing works in several of the following ways: (1) it provides an environment modeling methodology based on international software engineering modeling standards (UML 2.x, MARTE, OCL) that is dedicated to black-box, RTES system testing. The targeted RTES have complex environments and have soft-real time constraints in the order of hundreds of milliseconds pertaining to the response time of the SUT and operations of the environment. This is the first work focusing on such methodology, allowing the modeling of important concepts for testing such as modeling non-determinism and oracle information, while relying only on light-weight, standard extensions of UML (i.e., by defining a UML profile); (2) it provides an approach to generate simulators, based on environment models developed using the proposed environment modeling methodology, addressing the specific needs of black-box system testing; (3) regarding simulator generation, the paper provides extensions to the state pattern that handle time-related features and various UML 2.x features that were not previously discussed in the literature, including handling of change events and concurrency; (4) Unlike most of the works reported in the literature, this paper assesses the proposed methodology and simulator generation on two industrial RTES, which we believe is a requirement to assess the applicability of any test automation approach.

4 Motivating example

To motivate and explain our modeling methodology and simulator generation strategy for RTES, we take as example a subset of one of our industrial case studies. Note that we have sanitized the information due to confidentiality restrictions. This example is an automated bottle recycling system developed by Tomra AS, Norway. It is representative of the type of RTES we are targeting in this paper; it has soft real-time constraints in the order of hundreds of milliseconds pertaining to the response time of the SUT and operations of the environment (e.g., the sorting of item should be done within a couple of minutes after an item is inserted).

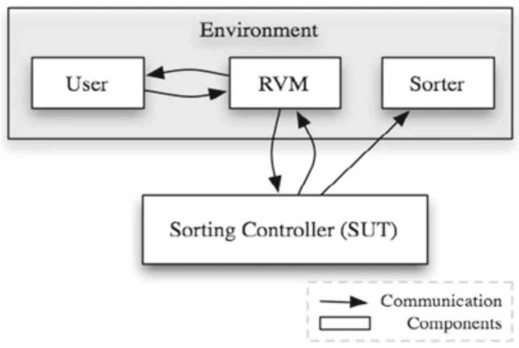

Fig. 1 Layout of the automated bottle recycling machine

The portion of the case study considered in this paper is focused on the important functionality of sorting the recycled items to their proper storage locations (or destinations). The layout of this portion is shown in Fig. 1. Users insert the items to be recycled inside the front-end of the recycling system, called the *Reverse Vending Machine (RVM)*. The items can be of three different types for the subset we are discussing: plastic bottles, cans, or glass bottles. The *RVM* forwards the items to the *Sorter*, which is a sorting arm (we only consider a simplified backroom with a single sorting arm). On its way from an *RVM* to *Sorter*, an item can be lost if it is not detected in time or if it falls from the moving belt. The *Sorter* can move in three directions (each leading to a specific destination) and its movement is controlled by a *Sorting Controller*.

The *Sorting Controller* is the system under test in our case study. The *Sorting Controller* receives information of the type of the item inserted from the *RVM* and when it is supposed to reach the *Sorter*. The *Sorting Controller* is responsible for moving the *Sorter* in a position that leads the items to their appropriate destinations. There can be different destinations based on the type of items. Plastic bottles and cans are placed in their appropriate bins, whereas the glass bottles are placed in the crates. The *Sorting Controller* should prevent certain erroneous situations from happening. For the subset of the case study discussed in this paper, we consider two such situations: (i) when an item is not correctly sorted and it goes to a wrong destination (for example, a plastic bottle going into a bin of cans) (ii) when an item reaches the *Sorter* while it is still moving.

5 Environment modeling methodology

If environment models are to be used for testing RTES, they should not only be sufficiently detailed, but should also be easy to understand and to 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 be well-defined 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 our industrial cases, we abstracted the functional details of the environment components to such an extent that only the details visible to the SUT were included. An environment of a RTES typically features a number of non-deterministic events (e.g., breakdown of a sensor), which must be modeled. Such events are not common when modeling the internal behavior of a system.

To model RTES environments, we have developed a profile that provides support for modeling various concepts central to our methodology and highlights the subset of UML/MARTE that is required for such modeling. For testing the system based on its environment, the behavioral 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 annotated with our defined profile. 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 and our defined profile.

In the kind of testing this paper addresses, the focus is on the interactions of the RTES with the components in its environment, i.e., what are the possible inputs/outputs to/from the RTES from/to these components at any given point in time? How does the RTES behave in abnormal situations, such as a hardware failure in any of the environment components? A test case for a RTES would typically consist of a sequence of actions from the user(s), signals from/to sensors/actuators, and possibly hardware component breakdowns. This would correspond, in our context, to non-deterministic events that can happen during the environment simulations.

In the following subsections, we discuss the environment modeling profile that we developed (Sect. 5.1) followed by the guidelines for modeling domain (Sect. 5.2) and behavioral models (Sect. 5.3) that were developed based on our experience of modeling two large-scale industrial RTES—a marine seismic acquisition system and an automated bottle recycling system. Guidelines presented in Sects. 5.2 and 5.3 are explained by examples related to the motivating example.

5.1 Environment modeling profile

Our goal was to model the environment based, to the extent possible, on the standard UML and its existing extensions. We applied the standard notations and based on our needs for those case studies, where required, we provided light weight extensions to UML. UML is a general purpose, standard modeling language that is meant to cater for different application domains and problems, and it is, as a result, quite large. The entire language is not meant to be used to solve a particular problem in a particular domain. Therefore, one of the key requirements to make UML successful in industry is to select a proper subset of the language matching the needs. MARTE is a comprehensive UML profile covering different aspects for modeling RTES. Similar to UML, the set of concepts provided by MARTE is fairly large and caters to a wide variety of analysis needs. However, it is important to clearly identify the required subset of MARTE for a specific problem and domain.

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.

Developing UML profiles is a way to provide lightweight extensions to UML that do not conflict with its original semantics. An alternate to this is extending UML semantics by adding new concepts (meta-classes). Even though the latter approach allows more customization of UML, due to a number of reasons discussed in [63] and easier industrial adoption, we opted for a profile. To model an RTES environment, generate its simulator, test cases, and obtain test oracle from these models, we need more specific notations than what the standard UML provides. We provided extensions to the standard UML class diagram and state machine notations in the form of a profile. The profile also resolved various semantic variation points left open by the standard (discussed later in Sect. 6.6) to address our specific needs. Figure 2 depicts a profile diagram for our proposed RTES environment modeling profile. The profile defines a set of stereotypes for modeling our methodology specific features on UML classes and state machines. It also shows the subset of MARTE that the profile is using, i.e., the Time package, the concept of *GaStep* from the Generic Quantitative Analysis

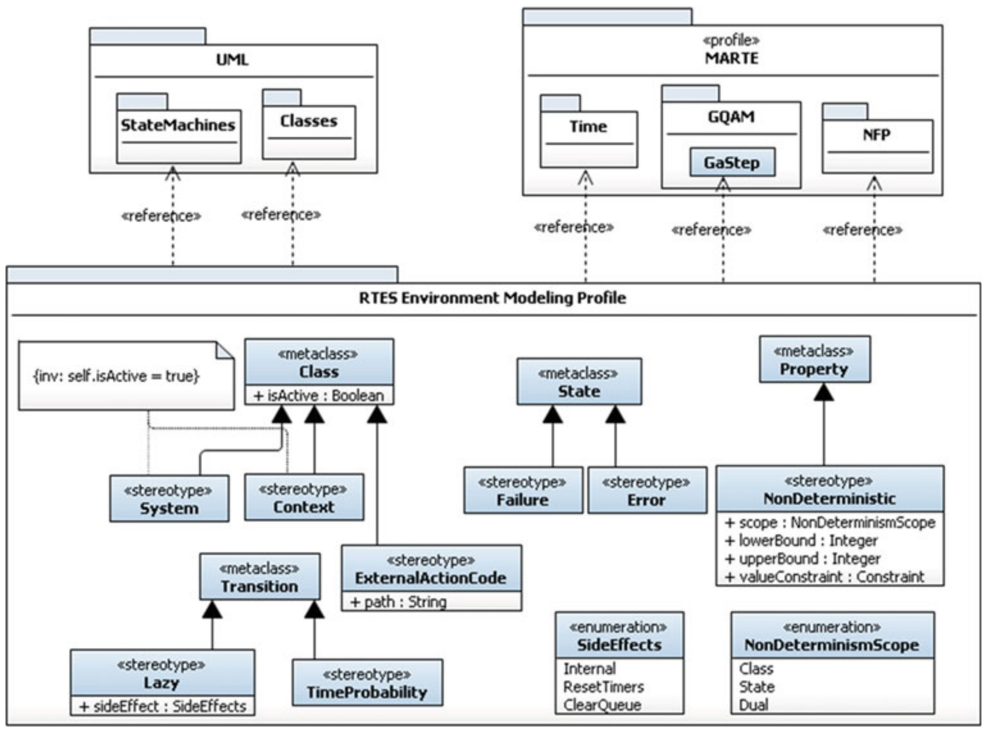

Fig. 2 RTES environment modeling profile

Modeling (GQAM) package, and the Non-functional Properties (NFP) package. The NFP package allows selecting more appropriate types for properties of components. The Time package allows the software engineers to model various time related features, such as timed events and action durations [5]. This small subset of UML and MARTE was sufficient for modeling our two industrial case studies for the purpose of automated black-box testing.

5.2 Domain modeling

Our environment modeling methodology for system testing requires the modeler to create an environment domain model that captures relevant structural details of the environment including the various components of the environment, their cardinalities, characteristics, and relationships. The domain model is developed using the UML class diagram notation.

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. The domain model represents various possible forms that the environment of RTES can take. 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. To restrict the possible forms an environment of an RTES can take, OCL constraints can be specified. These constraints can for example be used to restrict the possible combinations of environment components or to restrict the possible values of attributes.

The domain model for the Sorting Machine case study is shown in Fig. 3. *Sorting Controller* in the domain model is the SUT and the components *RVM*, *User*, *Sorter* and *Item* are the environment components. All the environment components are considered to be active objects, i.e., having their own thread of execution, and communicate with each other through signals. Each environment component in the domain model can have multiple instances. For example, in the domain model, shown in Fig. 3, Item is represented as one environment component, but during simulation it can have multiple instances. The number of instances to be created, which we refer to as an ‘environment configuration’, is determined based on the cardinality of relationships, i.e., in this case the cardinality of the association between *User* and *Item* with the role name *itemCollection* and the OCL constraints restricting the possible combination of environment components. In the motivating example we have restricted the possible number of items a user can enter to be less than 100. This is shown as an OCL constraint in Fig. 4. A valid environment configuration for this example is a single *RVM,* a single *Sorter* and a *User* with three *Items*. A test case in our context is a combination of a setting of the simulator for the

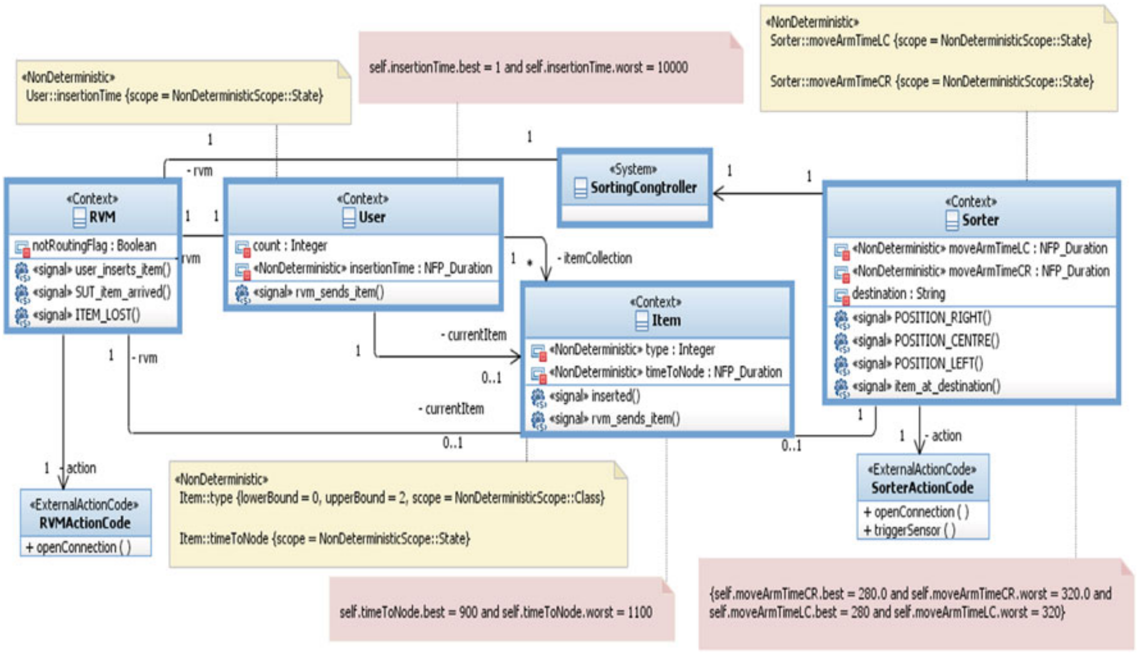
«NonDeterministic» User::insertionTime {scope = NonDeterministicScope::State}
self.insertionTime.best = 1 and self.insertionTime.worst = 10000
«NonDeterministic» Sorter::moveArmTimeLC {scope = NonDeterministicScope::State}
Sorter::moveArmTimeCR {scope = NonDeterministicScope::State}
«System» SortingCongtroller
«Context» RVM
notRoutingFlag : Boolean
«signal» user_inserts_item()
«signal» SUT_item_arrived()
«signal» ITEM_LOST()
«Context» User
count : Integer
«NonDeterministic» insertionTime : NFP_Duration
«signal» rvm_sends_item()
«Context» Item
«NonDeterministic» type : Integer
«NonDeterministic» timeToNode : NFP_Duration
«signal» inserted()
«signal» rvm_sends_item()
«Context» Sorter
«NonDeterministic» moveArmTimeLC : NFP_Duration
«NonDeterministic» moveArmTimeCR : NFP_Duration
destination : String
«signal» POSITION_RIGHT()
«signal» POSITION_CENTRE()
«signal» POSITION_LEFT()
«signal» item_at_destination()
- rvm
- itemCollection
- currentItem
- action
«ExternalActionCode» RVMActionCode
+ openConnection ( )
«NonDeterministic» Item::type {lowerBound = 0, upperBound = 2, scope = NonDeterministicScope::Class}
Item::timeToNode {scope = NonDeterministicScope::State}
«ExternalActionCode» SorterActionCode
+ openConnection ( )
+ triggerSensor ( )
self.timeToNode.best = 900 and self.timeToNode.worst = 1100
{self.moveArmTimeCR.best = 280.0 and self.moveArmTimeCR.worst = 320.0 and self.moveArmTimeLC.best = 280 and self.moveArmTimeLC.worst = 320}
1
1
1
*
0..1
0..1

Fig. 3 Domain model of sorting machine case study

context User **inv:**
self.itemCollection→size() > 0 and self.itemCollection→size() < 100

Fig. 4 An example OCL constraint

non-determinism in the environment models (e.g., a specific time at which a sensor stops working) (which we call 'simulation configuration') and an environment. A test case uses these settings for a particular simulation run. During testing, the selected test strategy decides the way these configurations for a test case are generated by the 'Test Framework' (e.g., random values for simulation and environment configurations when using random testing).

Note that the domain model that we develop is different from the ones commonly discussed in literature (e.g., [64]). 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.

5.2.1 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 is getting too complex, when possible, we can decompose the component and divide 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. Other components interacting with already included components can also be added in the domain model to have more control during simulation. For example, in the motivating example, we added *User* to model that items can be inserted in the *RVM* periodically (e.g., after every few seconds) with a time delay. The environment components in the domain are stereotyped with "Context". The environment components are modeled as active objects and can communicate with each other and the SUT through signals.

5.2.2 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 *SortingController* with the *Sorter*, which is controlled by the board.

5.2.3 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. For example, in Fig. 3, the attribute *type* of *Item* is used by the *SortingController* to move the *Sorter* in appropriate position before the items arrive.

By using the profile, it is also possible to leave the decision of selecting exact values for properties of the environment components till the time of testing (where it is decided by the simulation configurations). This concept is modeled by assigning «NonDeterministic» to the properties of environment components. This stereotype has three properties: an 'upper bound', a 'lower bound', a 'valueConstraint', and a 'scope'. The upper and lower bound specify the possible range of values that an integer property of an environment component can take during simulation. This is provided to ease the modeling of time events' bounds. Alternatively, an OCL constraint can be provided as a valueConstraint that restricts the possible values that an environment property can take. This constraint can, for example, be used to restrict a string property to certain specific values.

As shown in Fig. 2, the scope property can have three possible values: 'class', 'state', or 'dual'. If the value is set to 'class', the properties of the environment component instances are initialized with a value obtained from simulation configuration only once when the instances are created. If the value is set to 'state', the values are obtained whenever there is a state change in an instance. If the value of scope is set to 'dual', then a value is obtained for this environment component's property from the *simulation configuration* when an instance is created and the property is reassigned a value when there is a state change in the instance. For example, in Fig. 3, the property *type* of *Item* is a non-deterministic variable with the scope 'class' and its value is initialized based on a simulation configuration when an instance of *Item* is created.

5.2.4 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». For example, the *SortingController* in Fig. 3 is the SUT. The stereotype was used initially by Gomaa [29] to refer the system in a context diagram. Since, our goal is *software-in-the-loop* testing, the SUT modeled in the domain model represents the SUT and its execution platform as a single component.

5.3 Behavior modeling

For each environment component in the domain model that has a behavior affecting the SUT, our methodology requires to create a state machine representing this behavior. The state machine captures such behavior at the level of abstraction that is visible to the SUT. The state machines are developed using UML 2.x state machine notation and concepts, MARTE real-time extensions, and our profile to assist in modeling the environmental aspects of RTES. The MARTE profile is used to model the features related to time and a form of non-determinism. As discussed earlier, during simulation, the instances of the environment components run in parallel to form the environment of the RTES. They 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. Figures 5, 6, 7, and 8 show the state machines of the four environment components of our motivating example. Note that the diagrams are developed in IBM RSA v8.x, which adds some additional symbols to the triggers and effects in the state machines. A change event is not shown with a 'when' keyword as for example in the transition from *On_Hold* to *No_Item* in the *ItemInside* region of *RVM* state machine shown in Fig. 6. The version does not support value specification language of MARTE and the time events are entered as strings with quotes. All the guards in the state machines have the corresponding environment components as the context for OCL constraints. Following, we discuss the details of the methodological guidelines for modeling behavior of the environment components.

5.3.1 Identifying stateful components

Components whose states either affect the SUT or are affected by the SUT should be modeled with state machines. Overall, the environment should be modeled in a way that enables, after the initialization and provision of simulation and environment configuration*s*, the full simulation of the interactions with the SUT. All the environment components shown in Fig. 3 are stateful components of the sorting machine case study. For example, the *Sorter* component was modeled as stateful since it receives signals from the *Sorting-Controller* and reacts differently based on its current state.

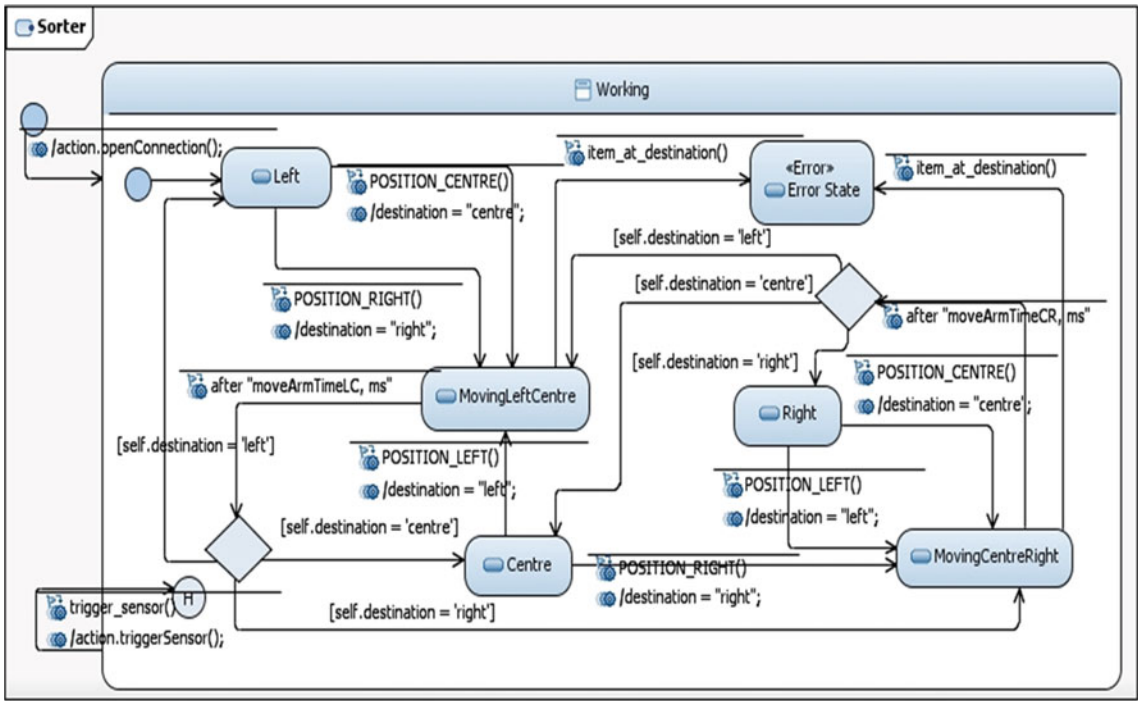

Fig. 5 State machine for the sorter component

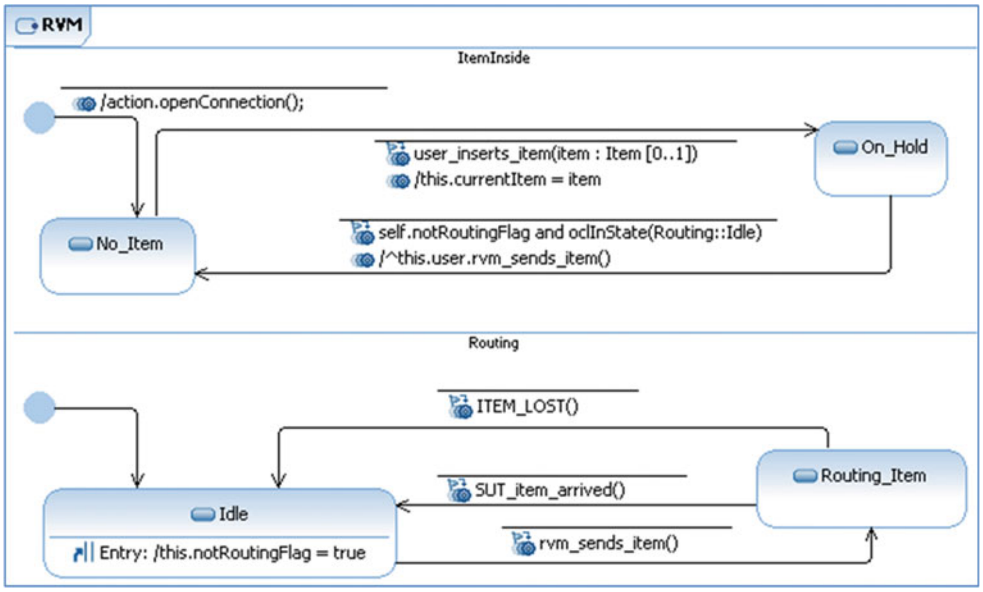
RVM
ItemInside
/action.openConnection();
On_Hold
user_inserts_item(item : Item [0..1])
/this.currentItem = item
No_Item
self.notRoutingFlag and oclInState(Routing::Idle)
/^this.user.rvm_sends_item()
Routing
ITEM_LOST()
Routing_Item
SUT_item_arrived()
Idle
Entry: /this.notRoutingFlag = true
rvm_sends_item()

Fig. 6 State machine of the RVM component

5.3.2 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 states of *Item* that we 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 *SortingController*.

A state in a UML state machine can be a simple state, a composite state (i.e., containing substates) or it can be a submachine state. UML state machines can also have multiple

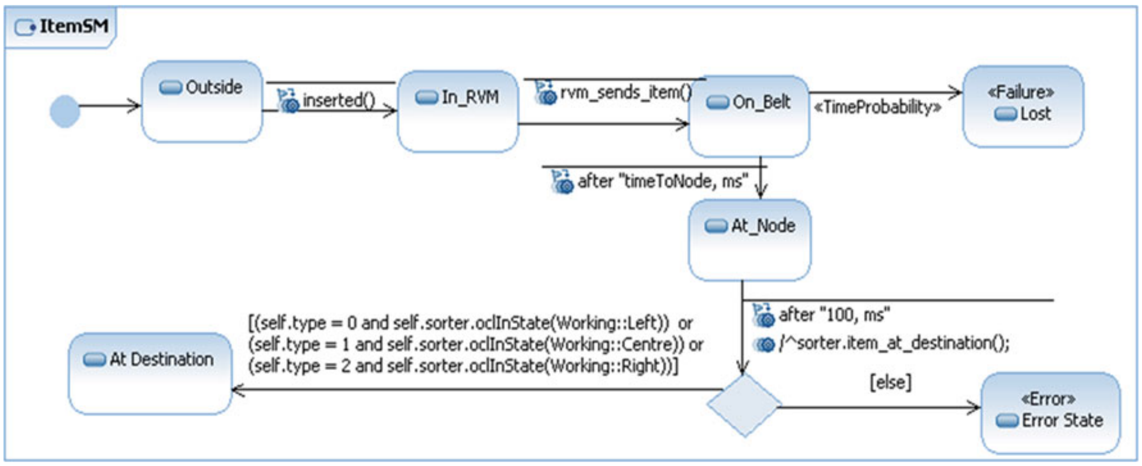
ItemSM
Outside
inserted()
In_RVM
rvm_sends_item()
On_Belt
«TimeProbability»
«Failure»
Lost
after "timeToNode, ms"
At_Node
after "100, ms"
/^sorter.item_at_destination();
[(self.type = 0 and self.sorter.oclInState(Working::Left)) or
(self.type = 1 and self.sorter.oclInState(Working::Centre)) or
(self.type = 2 and self.sorter.oclInState(Working::Right))]
At Destination
[else]
«Error»
Error State

Fig. 7 State machine of *Item* environment component

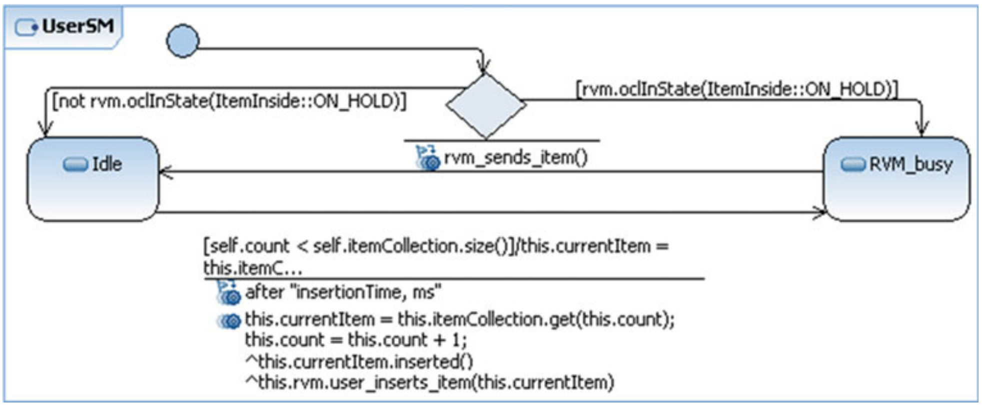
UserSM
[not rvm.oclInState(ItemInside::ON_HOLD)]
[rvm.oclInState(ItemInside::ON_HOLD)]
Idle
rvm_sends_item()
RVM_busy
[self.count < self.itemCollection.size()]/this.currentItem =
this.itemC...
after "insertionTime, ms"
this.currentItem = this.itemCollection.get(this.count);
this.count = this.count + 1;
^this.currentItem.inserted()
^this.rvm.user_inserts_item(this.currentItem)

Fig. 8 State machine of *User*

orthogonal regions. The concept of orthogonal regions is particularly useful in environment modeling as one environment component can in reality be composed of multiple sub-components. For example, *RVM* in our motivating example is composed of two sub-components: an item feeder that handles item insertion and a conveyer that is responsible for routing the items. From the perspective of the SUT (*Sorting Controller*) it is not important to distinguish these two components as it sees *RVM* as a single component. For the *RVM,* to completely simulate the behavior visible to the *Sorting Controller*, it must manage the movement of items on the conveyer in parallel to handling items in the feeder. From the *RVM* point of view, functionality must be provided for both of these components, conveyer and feeder. Therefore this information is modeled as two orthogonal regions of the *RVM* (named *ItemInside* and *Routing*) in the state machine shown in Fig. 6. In addition, according to our modeling methodology, failure behavior of a component that is independent of its nominal behavior can also be modeled as a separate orthogonal region (see Sect. 5.3.6 for details).

5.3.3 Modeling users in the environment

Generally, for software system modeling, users are only modeled as sources of inputs and data. 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 motivating example, we modeled the persons who enter the items for sorting as a *User* environment component (state machine shown in Fig. 8). In certain cases it can be interesting to model both the expected and unexpected behavior of users using the proposed methodology. Overall, the behavior of a user in an RTES environment is modeled using the same notations as any other environment component.

5.3.4 Modeling events

When using UML 2.x state machines [4] for environment modeling, only three types of events are required to be modeled: signal events, time events, and change events. Call events are not required since the components in the environment represent active objects and communicate asynchronously. OCL is used to model guards on transitions and conditions in the change events. For example, Fig. 5 shows the state machine of the *Sorter* component. As discussed earlier, a *Sorter* can be at three different positions. This is represented by the three states, *Left*, *Centre*, and *Right*. Movement between these states is represented by the outgoing transitions from these states to the two movement related states: *MovingLeftCentre* and *MovingCentreRight*. For the *Sorter* to move from *Left* to *Center* it needs to transition first from *Left* to *MovingLeftCentre*, which is triggered on receiving a signal event *POSITION_CENTRE*() from the SUT (*Sorting Controller*). A transition from *MovingLeftCentre* to *Centre* state is triggered by the time event *after "movingArmTimeLC, ms"*, where *movingArmTimeLC* is the name of a non-deterministic property of *Sorter* and *ms* is the unit of time, milliseconds. This transition is only triggered if the guard on the transition, written in OCL (*self.destination = "centre"*), is true. An example of a change event can be seen in the state machine of the *RVM* component (Fig. 6) in the *ItemInside* region on a transition from *On_Hold* to *No_Item*. The transition has an effect `^this.user.rvm_sends_item()` written in Java, which we chose as the action language as further discussed later.

The MARTE *TimedEvent* concept is used to model all timeout transitions, so that it is possible for them to explicitly specify a clock (if needed). 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. If no clock is specified (as in the case of motivating example), then by default the notion of time is considered to be according to the physical time. 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.

The proposed environment modeling profile allows the modeler to apply three stereotypes to transitions in the state machines: «Lazy», «TimeProbability», and «gaStep» (defined by MARTE profile). Following, we discuss the stereotype «Lazy», whereas the other two stereotypes are related to non-determinism and will be discussed later (Sect. 5.3.7).

By default whenever a component transitions from one state to another (i.e., transitions that are not internal), its event queue is emptied. To control the effect of a transition on the event queue and timers of the context component, we defined the stereotype «Lazy». In other words, this stereotype is a way to give more control to the modeler over the internal handling of the queues and timers. The stereotype has a property called 'sideEffect', which can have three possible values: (1) 'Internal', to denote that the transition should have no side effects on the source state. A transition with this stereotype will result in no alteration of the event queue and the various timers in the environment component; (2) 'ResetTimers', which will result in no alteration of the event queue, but will reset the timers; (3) 'ClearQueue' where the queue is emptied, but does not reset the timers.

5.3.5 Modeling actions and action durations

In our methodology, we chose Java as the action language for writing actions. The decision to choose Java as the action language at the model level is due to the current lack of tool support for the UML action language (ALF) [65] at the time of our tool development. Moreover, the testers of the SUT are expected to be more familiar with Java (consistent with our experience of applying the approach in two industrial contexts), rather than with a newly accepted standard language.

In the environment models, actions can be written in two places. Simple actions can be written inside the models, e.g., in the *RVM* state machine (Fig. 6), the simple assignment action of the transition from *ItemInside::No_Item* to *ItemInside::On_Hold* (i.e., `this.currentItem = item`) is placed directly as an effect. Relatively complex actions and communication related details are written in a separate source file and are referred to as the *external action code*.

External action code is the code that is to be written manually by the tester in a separate source code file, to communicate with the SUT (i.e., the test adapter) and compute complex effects. An example for the type of external action code is signals transmitted to the SUT over a UDP/TCP communication layer. If certain environment state parameters can only be computed by modeling continuous phenomena, then external action code can be used to invoke the corresponding code of other modeling and simulation tools, such as code generated from Simulink [18] using Simulink Coder [66]. As discussed earlier in Sect. 3.1, the code of other modeling and simulation languages does not support the testing constructs required by our methodology (e.g., *error* states). Excessive use of action code for this purpose can reduce the performance of the testing algorithms. For example, in the case of search-based testing, the search will not get any guidance from such code and therefore it may not be able to effectively guide the environment towards the error states. In both our industrial case studies, we did not need to model such continuous phenomena, therefore, we have not evaluated the practical implications of integrating our simulation code with such languages.

The classes containing the action code are stereotyped as «ExternalActionCode» and the path to the actual Java class containing the action code is provided with the *path*

```
1. import simula.embt.commons.*;
2. public class SortingMachineActionCode implements ExternalCode
3. {
4.      private int port;
5.      private String address;
6.      private Connection con;
7.      private IActiveObject sorter;
8.      public SortingMachineActionCode(Object[] args)
9.      {
10.             port = (Integer) args[0];
11.             address = (String) args[1];
12.     }
13.     public void openConnection() throws Exception
14.     {
15.             con = TCPConnection.openConnection(address, port, 1000);
16.             String msg = con.readMessage();
17.             sorter.receiveSignal(msg, null);
18.     }
19.     public void triggerSensor() throws Exception
20.     {       con.sendMessage(Signals.TRIGGER_SENSOR);      }
21.     @Override
22.     public void startExecution(IActiveObject ao)
23.     {       sorter= ao;                                   }
24.     @Override
25.     public void stopExecution() throws Exception
26.     {       con.stopExecution();                          }
27. }
```

Fig. 9 Excerpt of ExternalActionCode for the sorter component

property of the stereotype. Figure 3 shows an example of this where two action classes for *Sorter* and *RVM* are modeled. For the other two environment components, no action class was required. An object of this class is accessed in the models by using the role name of the association between the action class and context class. Calls to methods of external action code classes are simply made by using the role name, as shown in Fig. 5 in the transition action of the initial transition (with the role name *action*). An excerpt of the external action code for the *Sorter* component is shown in Fig. 9 (line # 13 and line # 19). The action code for the two messages sent to the action object in the state machine of *Sorter* (Fig. 5)—`openConnection()` and `triggerSensor()`—can be seen in the excerpt shown. Class `TCPConnection` is part of the communication library that we used. The method `triggerSensor()` simply forwards the signal to the SUT over the TCP connection. The decision to keep such action code separate was made to avoid cluttering the models with unnecessary details and to allow developers to write this code in a familiar programming tool. It was also important to keep the communication related information separate to avoid changing the models in case of changes in the communication mechanism. For example, if we want to change the communication from TCP to UDP, the only change will be in the external action code classes. Given a communication layer, even if the simulator is generated in Java, there is no particular restriction on the programming language in which the SUT is implemented.

Specifying a time threshold 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. Though in our case studies we did not face a situation where we needed to model action durations, the methodology supports this feature.

5.3.6 Modeling error and failure states

Two of the important features that are modeled in the state machines of environment components are the *Error* and *Failure* states. Failure states represent possible failures in the environment of SUT, e.g., hardware failure in the components. These states are required to test the robustness of the SUT when confronted to failures in the environment components. The failure states are modeled with the "Failure" stereotype. Failures that are independent of any specific aspect of an environment component's behavior (e.g., a hardware failure that can occur in any component state) are typically modeled using separate parallel regions within the state machine of the environment component.

Error states are the states of the environment that can only be reached due to faulty behavior of the SUT. These states are conditions that should never happen in the environment, otherwise indicating that the SUT is faulty. For example, a *Sorter* should never receive an *item* while it is moving and when there are no simulated failures in the hardware of the environment components, all items should always be delivered to the correct destinations based on their types. It is the responsibility of the *Sorting Controller* (SUT) to make the *Sorter* reach the appropriate position before an *item* reaches it. Otherwise, this would mean that there is a fault in the implementation of the *Sorting Controller*. This behavior of *Sorter* is modeled in the state machine shown in Fig. 5 as an error state, which is labeled with "Error". Error states are key oracle information that is used during the test execution of the SUT. By modeling erroneous situations as states, the methodology allows modeling of erroneous situations due to violation of temporal constraint (modeled as time transitions leading to error states), due to illegal change in the state of the environment (modeled as transitions leading to error states triggered by change events), and due to erroneous signal receptions (modeled as a transitions leading to error states triggered by signal events). Note that the error states are abstract states and not concrete states, because we are only interested in conditions that should never happen. One error state can in fact capture many concrete error states. In a large realistic system, we cannot possibly address all concrete error states, and we therefore only focus on the most important and hazardous situations according to what is possible with given test budgets. For both our industrial partners, we modeled error states referring to hazardous situations in a way that each of them could cover multiple concrete error states.

5.3.7 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. In the following, we discuss different types of non-determinism that we have modeled for our case studies.

For a number of RTES environment components, specifying the exact values for timeout transitions is not always possible. 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. Similarly, there can be properties of the environment components whose values can be between a specific range of possible values (e.g., items can be of different types). The modeler may require that the exact value for such properties is decided by the testing framework during testing because of their possible impact on testing (e.g., the position of sorter varies based on the *type* of item). For modeling these behaviors, the modeler can add an attribute in the environment component and label it with the stereotype «NonDeterministic». One option to specify the range of values is by providing upper and lower bound values for the corresponding properties of the stereotype. Another option is to write an OCL constraint as the 'valueConstraint' of the stereotype. In the latter case, the properties' upper and lower bounds will not be evaluated. If the properties being modeled are only used for non-deterministic time transitions, then another way of modeling is to set the type of the non-deterministic property to NFP_Duration and specify the 'best' and 'worst' properties of the NFP. The non-deterministic property can then be used as a parameter of a time event. If the type of a «NonDeterministic» attribute is NFP_Duration, then other properties will not be evaluated and the range will be obtained from the best and worst properties of NFP_Duration. In the state machine of the *User* (Fig. 8), the transition between the state *Idle* and *RVM_Busy* is modeling the behavior that this transition is non-deterministic and that the user can insert the next item with a delay ranging from 1 to 10,000 ms. The information constraining the values is provided in the domain model by applying the stereotype «NonDeterministic» on the *insertionTime* attribute (see Fig. 3). The attribute is then used as a parameter to the time event on the transition. The actual value (between the range specified) to be used during simulation is obtained from the simulation configuration. Since the scope property of the stereotype is set to 'state', the value for this attribute will be obtained from the simulation configuration every time the *User* enters *Idle* state, i.e., every time a new item is inserted.

There can be situations in which the modeler wants to restrict that a non-deterministic value is either only obtained once for each instance (i.e., assigned at the time of instantiation) or is obtained at the time of instance creation and every state change. As discussed earlier, these restrictions can be modeled by setting the property scope of the stereotype «NonDeterministic» to 'class' or 'dual' respectively. For example, The attribute *type* for an *Item* (see Fig. 3) on the basis of which the *Item* is sorted is modeled as «NonDeterministic» with scope set to 'class' and values constrained between 0 and 2 representing different types of items. This means that for each instance of *Item*, the attribute *type* is given a value by a simulation configuration when an instance of *Item* is created.

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 can assign 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 the test case execution. In the sorting machine case study, a *Sorter* can get stuck in a position (e.g., because of a bottle blocking it or the arm jamming) for example with a probability 0.02 in a minute if it is not moving and a higher probability when it is moving. The sending of non-deterministic signals can also be modeled using this type of transitions, by placing them in the actions of such transitions.

If the goal is *validation*, for example based on reliability estimation, then these probability values can be used as a sort of *operational profile* of the SUT [67]. On the other hand, if the goal of testing is the *verification* of the SUT, then the actual values of these probabilities are not important (test framework decides if an event happens, as long as its probability is not zero). For example, if the goal was validation, the above discussed scenario of a *Sorter* getting stuck could have been modeled with the *gaStep* stereotype to provide an exact value, range, or a probability distribution of occurrence of this failure.

For verification purposes, typically we only require modeling of such situations without specifying an exact probability value or distribution and leave the decision of exact value to the test framework. The stereotype «TimeProbability» on a transition is used to model such a non-deterministic trigger, whose occurrence is decided by the test framework and obtained from the simulation configuration. Such a transition is very useful to represent failures in environment components. For example, in the Sorting machine case study, an item can fall of the belt and be lost at any time while it is moving inside the machine. This is modeled as a transition from *On_Belt* state to a failure state named *Lost* in the *Item* state machine shown in Fig. 7. An alternate way of modeling this is by using unconstrained clocks. Whenever there is a time event that refers to an unconstrained clock, its value (i.e., when the event is going to be triggered) will be decided by the test framework.

Another type of non-determinism 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, we might want to represent the fact that during the communication with the SUT there is a chance that signals are received with or without distortion. For modeling such scenarios in UML state machines, we can use choice nodes. Whenever, there is a choice node with multiple outgoing transitions without any guards, the decision of taking one of the outgoing transitions from such a choice node is made at the time of execution by the test framework.

If the modeler wants to provide precise probabilities for such scenarios, she can assign 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. As mentioned earlier, 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.

5.4 Summary

For the purpose of modeling the environment to support black-box and automated RTES testing, we have defined a UML profile and a detailed modeling methodology. One of the major aims while developing the profile was to keep it as simple as possible to facilitate industrial adoption. The methodology describes guidelines for modeling both the structural and behavioral details of the environment. The structural details are modeled as an environment domain model, which captures the information of various environment components, their properties, and their relationships. For the domain model, we used the UML class diagram notation and annotated class diagram elements with the proposed profile. We define guidelines for identifying various components of the environment, their properties, and relationships. The behavioral details of the environment were modeled using the state machine notation annotated with once again with the proposed profile. Each environment component has one associated state machine. We defined guidelines for identifying states, modeling events, modeling non-determinism, and modeling error and failure states. Such state machines contain information of the nominal behavior of the components, their robustness behavior (e.g., break down of a sensor), and *error* states that should never be reached (e.g., hazardous situations). If any of these error states is reached, then it implies a faulty RTES. Error states act as the oracle of the test cases, i.e., a test case is successful in triggering a fault in the RTES if an error state of the environment is reached during testing.

6 Simulator generation

The environment models, comprising a domain model (UML class diagram) and behavioral models (UML state machines), are converted into a Java-based simulator using model to text transformations. The transformations are based on an extension of the state pattern [7], which is a well-known way of implementing state machines. The transformations proposed

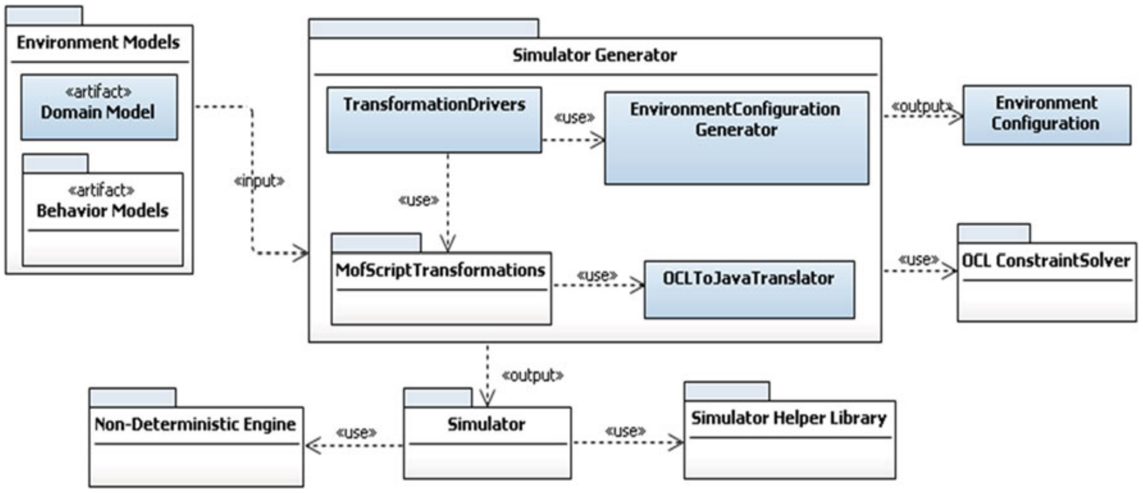

Fig. 10 Architecture diagram of simulation framework

here are defined to address the specific requirements for environment simulation and RTES system testing. In this section, we first provide an overview of the overall simulation framework (Sect. 6.1). Then we discuss our extended state pattern (Sect. 6.2) followed by a discussion on detailed transformation rules for domain model (Sect. 6.3) and behavior models (Sect. 6.4) to simulator code, thus providing a more thorough description of the pattern. This is followed by a discussion on the various important design decisions that we made for the transformation in Sect. 6.5. Section 6.6 discusses the resolution of all the related UML semantic variation points, whereas Sect. 6.7 discusses the tool support for the transformations.

6.1 Simulation framework

Figure 10 shows the architecture of the *simulation framework*. Components marked with the stereotype «artifact» represent the artifacts that are provided by the software testers to use the framework. The only input is the Environment Models that are developed according to the methodology discussed in Sect. 5.

The package named `Simulator Generator` contains the core components required for simulator generation. The sub-package `TransformationDrivers` contains driver classes provided with the framework that are responsible for configuring and running the model transformations. The `MofscriptTransformations` package contains the transformations we wrote in MOFscript [68] to translate the environment models to Java classes representing the environment simulator. Class `EnvironmentConfig uration Generator` is responsible for generating an environment configuration representing one possible setting of the environment. The class `OCLToJavaTranslator` is used by the MOFScript transformations to translate the OCL expressions in the model representing guards and change events to their Java equivalent. More details on how the simulator generator handles change events are provided later in Sect. 6.4.2. The components inside the Simulator Generator package generate a set of classes in Java corresponding to the environment models given as input. This is represented as a `Simulator` package in Fig. 10. The generated simulator is statically linked to classes from two packages: the `Simulator Helper Library` and the `Non-Deterministic Engine` which we discuss below.

The `Simulator Helper Library` is developed to support a number of features required by the generated simulator. The library is independent of the case studies and hence is developed as a separate library. The library contains generic features required by active objects (event queue, event handling mechanism, etc.), time related functionalities (including features for handling clocks and timed events), collection classes (providing facility for sending broadcast signals to all elements in the collection), and support for implementing the defined extension of the state pattern, as discussed further in Sect. 6. The core package of the library is shown in Fig. 11. The class `ActiveObject` represents the UML concept of an *active object*. The class provides an event queue for the active objects in the form of a `java.util.PriorityBlockingQueue`. The queue is a blocking queue and enables setting the priority of elements in the queue. The queue holds instances of class `EventInvocation`. An `EventInvocation` instance represents an event that is ready to be executed and is placed in the event queue of an `ActiveObject`. `EventInvocation` has an attribute `methodToInvoke` that is of type Method from the Java reflections package

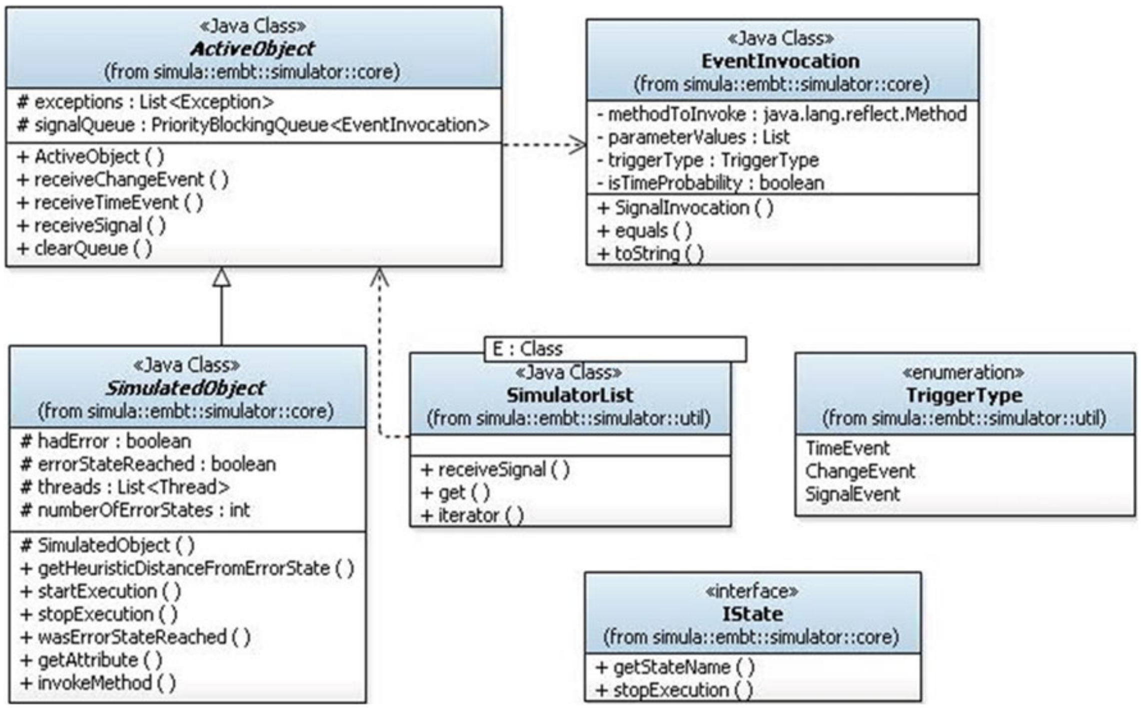

Fig. 11 Important classes in the SimulatorHelperLibrary

and contains the method of the `ActiveObject` to be invoked along with its parameter types, an attribute `parameterValues` containing the values for the parameters of the method to be invoked, an attribute `triggerType` representing the type of the trigger (signal event, change event, or time event), and an attribute `isTimeProbable` that indicates whether the method to be invoked represents a time probability trigger. The class `ActiveObject` is an abstract class containing the generic behavior of an active object. The behavior provided by the class is used both for the environment components (extending the further generalized class `SimulatedObject`) and implementation of parallel regions (since each region has its own thread of execution and an event queue). The interface `IState` is implemented by all the classes representing UML states. The class `SimulatorList` is used to implement relationships having a multiplicity greater than 1. The class provides facility to broadcast events to all the elements it contains at any given time. For example `SimulatorList` is used to implement the relationship (*itemCollection*, Fig. 3) between *Item* and *User* for the Sorting Machine case study, so that signals to items can be easily broadcasted.

The `Non-Deterministic Engine` is responsible to provide a link between the simulator and various simulation configurations produced by the test framework. The `Non-Deterministic Engine` is called by the simulator each time a non-deterministic occurrence needs to be produced, which in turn queries the current simulation configuration and returns the value generated by the test framework corresponding to the non-deterministic occurrence. This is handled by assigning a unique id to each non-deterministic occurrence during the entire simulation, based on the formula: <NDID> = <INSTANCE_ID> * <MAX_ND_COUNT> + <LOCAL_ND_ID>, where NDID is the non-deterministic occurrence id to be calculated, INSTANCE_ID is the unique id assigned to instances of environment components, MAX_ND_COUNT is the maximum number of non-deterministic occurrences that any component in the domain model can have, and LOCAL_ND_ID is the unique id for the occurrence for each environment component. For example, in the *Item* component state machine (Fig. 7), the transition from *On_Belt* to *Lost* is stereotyped as «TimeProbability», which is a non-deterministic event. The actual value for the time when this transition is to be taken is obtained by the simulator through the `Non-Deterministic Engine`. As discussed earlier in Sect. 5.3.7, non-determinism can be of multiple types. An excerpt of generated code in Fig. 12 shows a call to the `Non-deterministic Engine` in order to initialize the value of the attribute *type* of the *Item* environment component. The statement `instanceId` * 3 + 0 will result in a unique id for this non-deterministic occurrence during simulation.

6.2 An extended state pattern for environment simulation

The original state pattern, discussed in [7], provides a design pattern to implement state-driven behavior in an

Fig. 12 Code snippet showing call to non-deterministic engine

```
public Item(int id){
        super(1);
        this.instanceId = id;
        type = (Integer)TestCaseHandler.getTestCase().
                getNextNonDeterministicValue(instanceId*3 + 0);
}
```

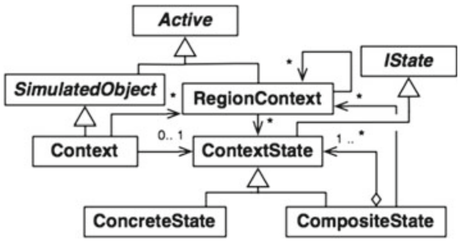

Fig. 13 Extended state pattern meta-model

object-oriented programming language. The idea of the pattern was to provide a clean way to implement state-based behavior and make it easy to add new states or transitions by confining the code related to states in separate classes and by providing a mechanism to change the state class at runtime. The original state pattern did not, however, specify a number of important features present in UML 2.x state machines, such as concurrency, time events, change events, and effects. A number of works have provided extensions to the basic state-pattern for various purposes. We discuss these extensions in the related work section and explain why these extensions do not entirely match our needs. In this section, we describe how we extend the basic state pattern for various features required by our environment modeling methodology.

Figure 13 shows the meta-model of the proposed extensions to the state pattern. The meta-model is included here for ease of understanding. The actual transformation is from UML models directly to Java code (without an explicit target meta-model). The abstract meta-classes `Active`, `IState` and `SimulatedObject` represent the classes with the same names in the `Simulator Helper Library`. The core package of `Simulator Helper Library` is shown in Fig. 11. These classes were required for the environment simulation and are not defined in the original state pattern.

The concept of Active is similar to the notion of an active object in UML. Instances of this class hold an event queue and run as a separate thread. All state classes extend the `IState` class. The class is implemented as a Java interface in Simulator Helper Library. `SimulatedObject` holds a list of threads that have been created so far for the instance of an environment component and whether or not the simulation has resulted in reaching an error state. The classes `Context`, `ContextState` (called *State* in state pattern), and `ConcreteState` perform similar roles as in the state pattern. The class `Context` corresponds to a «Context» component of the environment. It holds a reference to all the states and to the current state of the environment component instance and forwards the incoming events to the current state object. A `ContextState` can be a `ConcreteState` or a `CompositeState`. The `ConcreteState` class represents the simple states where actual implementations of triggers are defined. According to the modeling methodology, all events that are not explicitly defined on a state are ignored (Sect. 6.4). To implement this behavior, the `ContextState` class provides an empty implementation of all the operations that correspond to signals defined for the `Context` class. The operations have the same signature (i.e., the same name and the same parameters) as the signal. Since all the concrete state classes extend the `ContextState` class, if a signal is not accepted in a state, then its implementation is not provided in the corresponding `ConcreteState` class. This results in executing the empty implementation and the event is ignored. Since the original state pattern does not handle parallel regions or composite states, we have added the `CompositeState` and `RegionContext` classes for this purpose. An instance of `CompositeState` class is created for each composite state in the state machine. A `CompositeState` holds substates that can be either a `ConcreteState` or `CompositeState` (implemented as composite pattern [7] in the meta-model). Instances of `RegionContext` are generated for each parallel region in the state machines. All the states within a region are created as instances of `ContextState`. A `RegionContext` can have further links with other `RegionContext` objects in case of further parallel regions within a region.

Further extensions to the state pattern are related to handling the required simulation details for RTES system testing. These extensions include the support of non-determinism that is a common feature in RTES environments. We also provide support for change events, time events modeled using the MARTE profile, and handling actions written in Java. We also provide a generic way to develop and integrate a communication layer in the generated code for communication with the SUT (via the external action code as discussed earlier in Sect. 5.3.5). The generated code also supports the generation of code for error and failure states and various heuristics necessary to apply search-based testing techniques (e.g., for calculation of branch distances [44]).

The overall event processing in the simulator for active environment components is based on run to completion processing as defined by UML semantics. The active object waits on the event queue, performs a dequeue operation, processes the received events, and waits on the event queue again if the queue is empty.

6.3 Transformation of the domain model

The domain model is used to obtain information regarding various environment components, their relationships and possible cardinalities of associations, attributes, non-deterministic attributes, signals, and signal receptions. This information is used throughout the transformation process and is also included in the generated simulator classes. Since the transformation rules are based on an extension of the state pattern, a number of Java classes are generated for every stateful component in the domain model, e.g., *RVM, User*. As discussed earlier, every environment component is translated into a Java class which is an instance of the `Context` meta-class. A list of the important auto-generated methods in such context classes along with their descriptions is provided in Table 1 and a list of generated attributes is given in Table 6 of Appendix. The various methods listed in the table are further discussed in Sect. 6.4 (since most of them relate to the behavior models). Every context class instance for each environment component will be assigned a unique id, called `instanceId`, during simulation. The `instanceId` is decided by the environment configuration and is passed to the constructor of context classes when the instances are created, as shown in Table 1. Each environment component holds a reference to instance of a state object that represents the current state of the component. A method `oclInState(stateName)` is provided for every component that returns true if the component is in a state with name equal to `stateName`. The method corresponds to the `oclInState` method defined by the OCL specifications and is called during evaluation of various OCL expressions that need to check the state of a component. Whenever a component changes its state, the method `changeState(fromState, toState)` is called, which updates the state object referring the current state. The method `startExecution()` is called when the simulation is started and `stopExecution()` is called when the simulation is stopped. These methods call methods with the same name in the *external action code* (shown in line # 22 and line # 29 in Fig. 9). Code related to acquiring or releasing resources can be placed in these methods. Effects defined on transitions where either the constructor of the environment component is the trigger or the transitions are initial transitions (i.e., their source is the initial pseudo-state) are implemented in the `startExecution()` method. For example in the *Sorter* state machine (Fig. 5), the effect of initial transition (`action.openConnection()`is implemented in the `startExecution()` method of context class `Sorter`).

Relationships are implemented following the standard class diagram conversion rules [69]. Signal receptions are translated into Java methods in the context class. The associations with an environment component at the navigable end, with a multiplicity above one, are implemented using a `SimulatorList` collection from the `Simulator Helper Library` (Fig. 11). If, in the action code, a signal is sent to a role name having multiplicity more than one, then it is sent to all the elements of the collection. The attributes of classes that are stereotyped as «NonDeterministic» (e.g., *moveArmTimeLC* in *Sorter*) are used to generate an output file, called `NonDeterministicOccurrences` that contains the range of values specified in the model for these attributes. This file is used by the test framework to identify the domain of valid test cases. Initial values for the environment configuration are randomly generated based on the OCL constraints defined on the attributes.

6.4 Transformation of behavioral models

In this section, we will discuss some of the important rules for transformation of environment behavioral models (i.e., UML/MARTE state machines). As discussed earlier, depending on the type of the states in state machines of environment components (i.e., simple, orthogonal, and composite states), instances of corresponding sub-classes of meta-class `ContextState` are generated. Various methods that are generated for the instances of meta-class `ContextState` and its subclasses are discussed in following sections. A summarized list is provided in Table 7 in Appendix.

6.4.1 Handling hierarchical state machines

We have already discussed how a simple state is handled by the code generator (in Sect. 6.2). Since a submachine state is semantically equivalent to a composite state [4, p. 566]), both of them are treated in the same way for code generation. In the remainder of the section, we describe in detail how the code generator handles orthogonal regions and composite states.

To translate parallel regions into code, we introduced the concept of a `RegionContext` class (Fig. 13). The class acts as the state pattern context class for various states in that region (i.e., it holds the current state object and forwards the messages to it). The `Context` class in these cases has a reference to this `RegionContext` class. Each `RegionContext` class extends the *Active* class and thus holds a queue of events. This was important for simulation in order to allow each region for separate processing and execution of events in its queue. Figure 14 shows the static structure

Table 1 Automatically generated methods in instances of the context meta-class

Method name	Description
«Constructor» (int instanceId)	The constructor is passed with the unique `instanceId` for this instance during the simulation. Actions of constructor are included in `startExecution()`
`startExecution`	This method is called at the start of execution and all the initialization of attributes, regions, and states is done here. Threads for concurrent regions are started in this method
`stopExecution`	This method is called at the end of execution. Various threads are stopped, and resources are released in this method. For example in the sorter case, a call to action code to close the opened sockets is sent from this method
`evaluateChangeEvents`	This method is called from setter methods to evaluate whether any of the change events is affected by the current change
`executeChangeEvent(condition)`	This method is called when the condition of a change event has been satisfied. The call is forwarded to `executeChangeEvent()` of the current state object
Setter methods	Setter methods are generated for every attribute and association of the environment components
Getter methods	Getter methods are generated for every attribute and association of the environment components
`oclInState(stateName)`	Evaluates whether the component is in the specified state. The Semantics is similar to OCL method `oclInState`, except that the parameter `stateName` is of type `String`
timeout	Called whenever a timeout has occurred (i.e., a timer of a time event has expired). The method calls the timeout method in the current state object
Signal methods	These methods are generated for every signal that is accepted. The method forwards the call to the method implementing the signal in the current state
State getters	A getter method is generated for every `ContextState` class
`executeCompletionEvent`	This method is executed when the entry actions of the current state object have been executed. The call is forwarded to current state object
`changeState(fromState: IState, toState:IState)`	The `fromState` object refers to the source state and the `toState` refers to the target state. `onStateExit()` in `fromState` and `onStateEntry()` in `toState` are called. Current state is changed to `toState`

of the code (without operations and attributes) produced by the code generator for the *RVM* state machine shown in Fig. 6. Classes name `ItemInsideStateContext` and `RoutingStateContext` are the two `RegionContext` classes corresponding to the two regions of the state machine. Both these classes have an association to their corresponding `ContextState` object, `ItemInsideState` and `RoutingState`, respectively. These `ContextState` classes are specialized by the `ConcreteState` classes, for example, in the case of *RVM*, `RVMIdleState` and `RVMRouting_ItemState` are instances of meta-class `ConcreteState` (for the states *Idle* and *Routing_Item* in Fig. 6 respectively) and specialize the `RoutingState` class, which is an instance of the `ContextState` meta-class.

For every composite state, at least two classes are added to the state hierarchy. If a composite state does not contain parallel regions, then an instance of `CompositeState` class is added and, for all the sub-states of this composite state, instances of `ConcreteState` are added. If there are parallel regions, then an instance of a `RegionContext` class is added and the `CompositeState` class has an association with it (Fig. 13). The rest of the handling is similar to other states as defined by the UML semantics (e.g., when a sub-state is entered, the entry actions of composite states are executed first followed by the entry actions of the sub-state).

6.4.2 Event handling

As discussed earlier (in Sect. 5.3.4), the environment modeling methodology allows three types of events to be modeled: signal events, change events, and time events. Since environment components represent active objects, each of them contains an event queue that holds the events that have been dispatched, but have not been executed yet. An environment component instance is considered to be busy when it is executing behavior corresponding to an event. When an event is triggered on a busy instance, the event is kept in the event

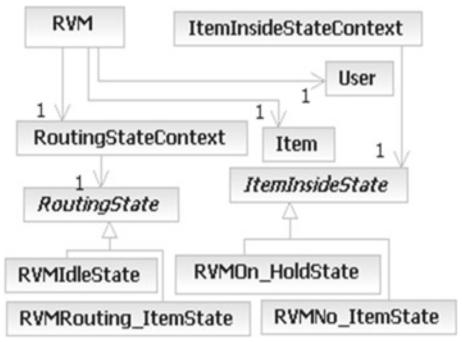

Fig. 14 Generated code structure for RVM environment component

queue and is processed after the current execution is finished. To avoid race conditions and to make environment simulation repeatable, each event is assigned a unique id in the context of an environment component at the time of code generation. For time events, the time expression and the guard of an event are used to identify a time event. For change events, the change condition is used to identify a change event uniquely, and signal events, are uniquely identified by using the name of the signal and the constraint of the guard. In the following, we discuss how the events of different types are translated into simulator code.

Handling Signals The rules for handling signals are defined according to UML semantics. Here, we discuss how they are realized using the proposed extensions of the state pattern. Sending of a signal from one component to another is done in the generated code by calling a `receiveSignal` method of the target context instance, which extends the *Active* class in `Simulator Helper Library` where `receiveSignal` is defined. The name of the signal being sent (a Java String) and the arguments with which the signal is to be invoked (an array of Java Object) are passed as parameters of the `receiveSignal` method. The method places the received signal in the queue as an `EventInvocation`, which represents the method to be invoked and the parameters. This behavior is shown as a sequence diagram in Fig. 15. When the context object is ready to process the signal, then the method of the `EventInvocation` is invoked on the context object using Java Reflection API. Whenever an instance exits a state, its event queue is emptied (except for «TimeProbability» events, discussed later).

Signals to a SUT are sent through the action code. All the signals towards the SUT are first forwarded to a corresponding method (with the same signature) of the action code. For the reasons discussed earlier (in Sect. 5.3.5), the low level details for sending the signal to the SUT over a communication medium are written in the external action code manually by the developer. For example, in the state machine of *Sorter* shown in Fig. 5, as a result of receiving a signal `trigger_sensor()`, the *Sorter* sends a signal to the SUT. This is modeled as a signal to the action class `action.triggerSensor()`. The implementation of the action class corresponding to the *Sorter* is shown in Fig. 9. The action class contains `triggerSensor()` (line # 19 in Fig. 9) which implementation is to forward this signal to the SUT over the TCP connection (implemented as `sendMessage(..)`). Similarly, it is the responsibility of an action code developer to define a mechanism by which the signals are received from the SUT and are forwarded to the context objects. For this purpose, every action class has an access to its corresponding context object. For example, in the external action code of the *Sorter* shown in Fig. 9, an object of type *Sorter* is passed as an `IActiveObject` to the method `startExecution()` (line # 22 in Fig. 9). This method is called at the start of environment simulation and is a way to allow the action code writer to provide initializations required at the start of execution (e.g., opening of sockets). As shown in the implementation of the action code in the figure, the reference of the instance is kept in a local variable of the class (line # 23), which is then used to send messages to the *Sorter* component (see line # 19 in Fig. 9) by invoking message `sorter.receiveSignal()`. In the state machine of *Sorter* shown in Fig. 5, the signals `position_left(), position_centre(), and position_right()` are sent by the SUT to the *Sorter* and are handled in the above mentioned way.

Handling Change Events A special mechanism was implemented for handling change events. In the generated code, setter methods are generated for the attributes of environment components. All action code statements that are assigning values to attributes are converted to corresponding setter calls of state machines. Within the code of the setter method of every attribute there is a call to `evaluateChangeEvents()` in the context object. The method forwards the call to the current state object. Within the state object, `evaluateChangeEvents()` evaluates whether the change in the attribute value has an impact on any of the possible change events. If this is the case, then the corresponding condition which was evaluated to true is returned by the method. In the case where multiple change events are true, the condition corresponding to the event that has a minimum corresponding id value will be selected (see details on calculation of id values for events in Sect. 6.6.1). In the context object's setter method, if the condition returned is not null, then a call to `executeChangeEvent()` with the condition as parameter is placed in the event queue of the context object. This mechanism is similar to handling signals in the queue. The only difference is that the change events have a higher priority than the signal events (reasons discussed later in Sect. 6.6.1). The mechanism was adopted in order to execute change events asynchronously for active objects. As an example, the behavior of what happens when a setter method is called for the `notRoutingFlag` attribute of the *RVM* component is shown in Fig. 16.

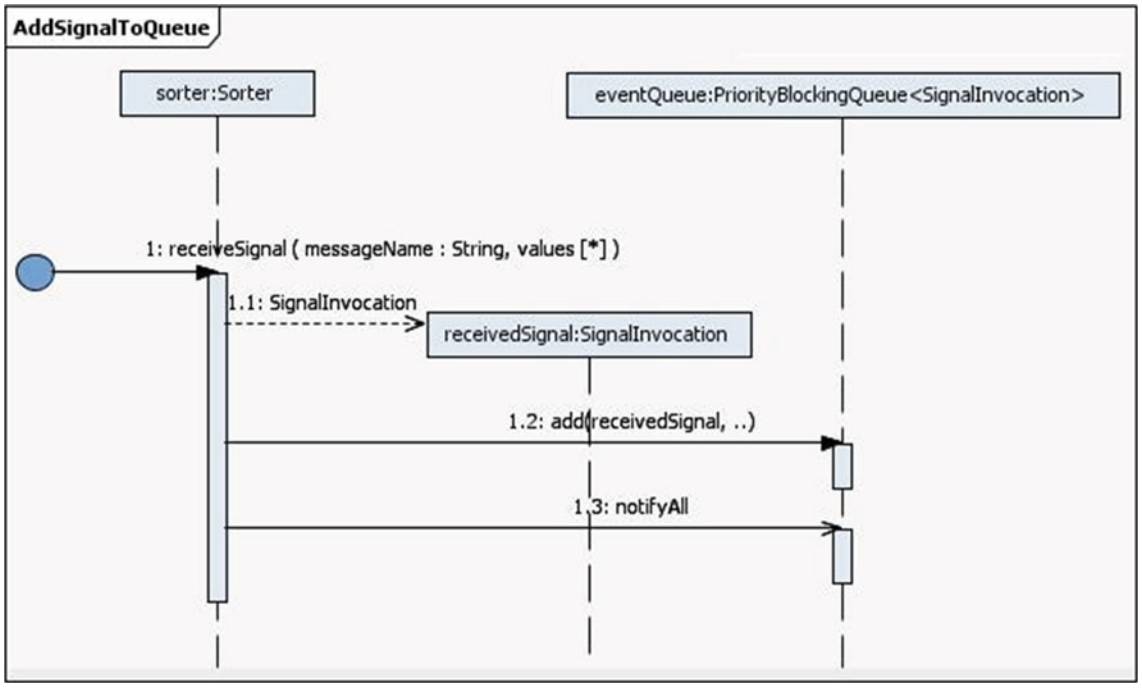
AddSignalToQueue
sorter:Sorter
eventQueue:PriorityBlockingQueue<SignalInvocation>
1: receiveSignal (messageName : String, values [*])
1.1: SignalInvocation
receivedSignal:SignalInvocation
1.2: add(receivedSignal, ..)
1.3: notifyAll

Fig. 15 Sequence of message calls on receiving a signal

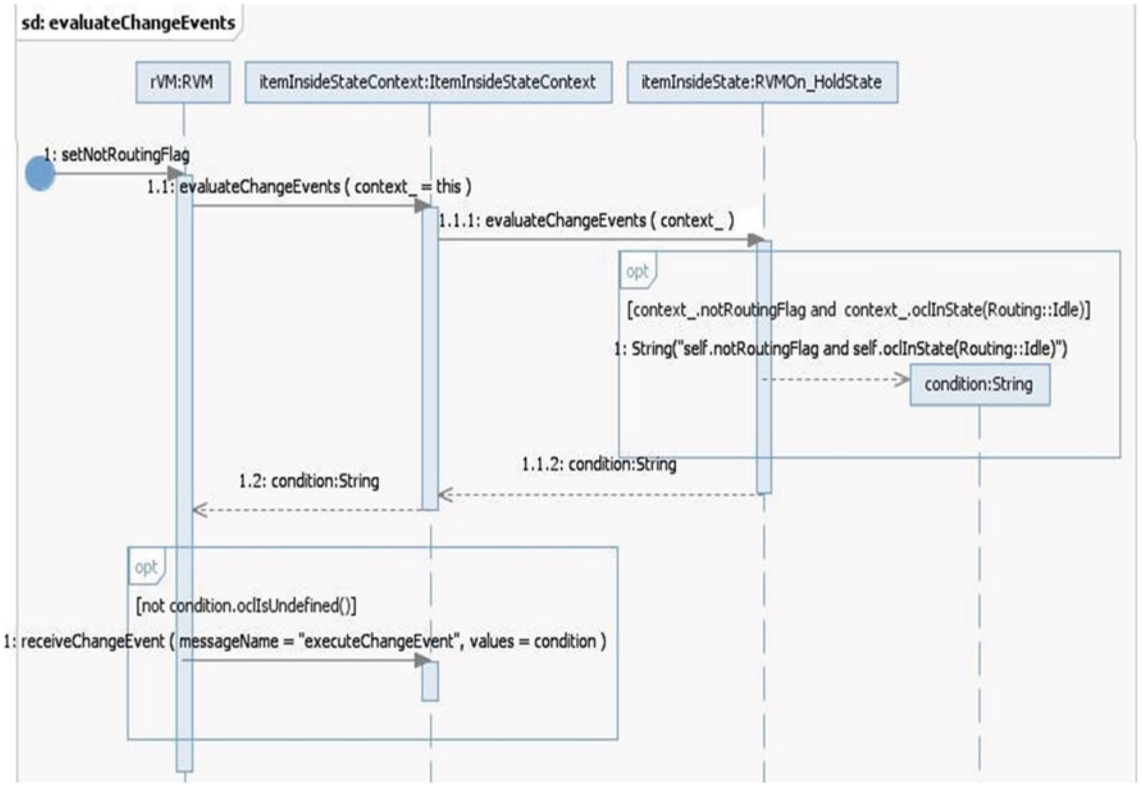
sd: evaluateChangeEvents
rVM:RVM
itemInsideStateContext:ItemInsideStateContext
itemInsideState:RVMOn_HoldState
1: setNotRoutingFlag
1.1: evaluateChangeEvents (context_ = this)
1.1.1: evaluateChangeEvents (context_)
opt
[context_.notRoutingFlag and context_.oclInState(Routing::Idle)]
1: String("self.notRoutingFlag and self.oclInState(Routing::Idle)")
condition:String
1.1.2: condition:String
1.2: condition:String
opt
[not condition.oclIsUndefined()]
1: receiveChangeEvent (messageName = "executeChangeEvent", values = condition)

Fig. 16 Sequence diagram showing the behavior for evaluating change events

A change event depending on the value of this attribute is shown in Fig. 6 (i.e., `self.notRoutingFlag` and `oclInState(Routing::Idle)`). When the `execute ChangeEvent` method is executed, it forwards the call to the current state object, which in turn evaluates the condition again and executes the transition corresponding to the change event. If the condition is not satisfied, then nothing happens and the event will be considered as lost.

Handling Time Events Another non-trivial transformation is for the time events in state machines. We have created a TimeService Library, as part of Simulator Helper Library that is responsible for managing time-related operations (e.g., clock handling, event scheduling). In every state class that has an outgoing transition with time trigger(s), an object of type TimeService is added, which is responsible for handling time-related operations. For each time event, a corresponding TimeInstance object is included that is initialized to the value of the time event. A method afterT<i> is also included for every time event, where <i> is the unique index associated with each time event. For such state classes, a timeout() method is also implemented, which has the logic of forwarding the call to the correct afterT<i> method.

The time events are scheduled according to their corresponding clock. If no specific clock is associated, then they are scheduled on first entry into the state class and are reset on every next entry into the state. Non-deterministic time events (where times of occurrence are determined by the simulation configuration) are handled as discussed in a specific section below (Sect. 6.4.5). When a timer associated with a scheduled time event expires, a signal timeout() is sent to the context object with the information of the time instance. The context object forwards the call to the *timeout* method of its current state.

6.4.3 Handling guards and actions

Since the environment models are translated to Java code, the OCL guards on the models also need to be translated to Java. For search-based testing we need to evaluate the OCL guards and the corresponding branch distance [44]. Therefore, the OCL guards are translated to their Java equivalent along with instrumentation code to support testing heuristics at run time [44].

As mentioned earlier (in Sect. 5.3.5), we have used Java as an action language. Complex action code and the code related to the communication with the SUT (e.g., handling UDP/TCP sockets, writing to the file system) are written in a separate action code file developed as part of the modeling activity as discussed in Sect. 5.3.5. Recall that the mapping between the *Context* class and its action code class is provided using the stereotype «ExternalActionCode». The actions on the transitions are placed inside the body of the corresponding events in the instance of ConcreteState class corresponding to the state from which the transition is outgoing (e.g., the action discussed above is placed in the method for the signal user_inserts_item() in the class RVMNo_ItemState). Internal state activities (entry, do, and exit) are handled as defined by the UML semantics. Their implementation is provided in the onStateEntry() method of the state classes. On completion of the do activity, executeCompletionEvent() in the state class is called.

6.4.4 Handling Oracle information

As discussed earlier (in Sect. 5.3), error states represent the states of the environment that are reached due to a faulty implementation of the SUT. For example, in the Sorting machine case study, there are two possible errors: the items are not sorted correctly according to their type or an item reaches the Sorter while it is moving. The first error scenario is modeled in the state machine of Item (Fig. 7) and the second scenario is modeled in the state machine of the Sorter (Fig. 5). For the purpose of verification, the testing that we performed in [44] aimed at reaching the error states of the SUT. Note that the approach does not require any sophisticated technique for oracle generation. The error states are modeled with similar syntax as the regular states and represent the test oracle in our context. The oracle consists in checking that any environment component instances do not traverse any of the error states during test execution. Each error state for each instance of the environment components is assigned a unique id during the simulation and the search heuristics (to help the generation of test cases, discussed in Sect. 7.1) use this id to report information relevant to the selected test heuristic, e.g., the distance from the error state for search-based testing.

During the simulation, a number of components in the environment might fail, but with a correct SUT, the environment should never enter in an error state, although the SUT would likely operate with degraded functionalities. In terms of code generation, the execution is stopped once an error state is reached, a complete log showing the execution trace is generated, and a *JUnit* test case is generated corresponding to the values used in the simulation in order to enable the re-execution of the test case.

6.4.5 Handling non-determinism

Non-determinism can be of five types in the environment models, as discussed in Sect. 5.3.7. In the following, we discuss how each of the five types of non-determinism are handled for simulating the environment. Note that, as discussed earlier in Sect. 6.1, a unique id is assigned to each non-deterministic occurrence during the entire simulation, which is used by the Non-deterministic Engine to select an appropriate value for this occurrence from the simulation configuration.

The first form of non-deterministic occurrence is due to a trigger accessing a class attribute that has a «NonDeterministic»stereotype. On entering a state, when one of the outgoing transitions contains a trigger with such an attribute, the generated code passes the unique id of the non-deterministic occurrence to the `Non-deterministic Engine` and obtains an appropriate attribute value from the simulation configuration. Handling of time events accessing such variables was discussed earlier in Sect. 6.4.2.

The second form of non-determinism is when a variable of an environment component (modeled by assigning a stereotype «NonDeterministic») needs to be initialized at the time of instance creation during simulation. This information will be implemented by having the component constructor query for a value from the `Non-deterministic Engine`. For example, for the domain model in Fig. 3, the code that initializes the value for the attribute `type` of `Item` is shown in Fig. 12.

The third form of non-determinism is the explicit representation of the probability to take a transition, which is specified by the «gaStep» MARTE stereotype. During simulation whenever the code corresponding to such transitions is reached, based on the probability it is decided whether or not to take the transition.

The fourth type of non-determinism can be due to the «TimeProbability» stereotype on a transition. The time value specifying the delay in taking the transition is obtained from the Non-deterministic Engine when the state is entered for the first time or when the instance is reentering the state after this transition has been executed.

The fifth type of non-determinism is possible when we have a choice node with multiple outgoing transitions without a trigger. During simulation, whenever the code corresponding to such a choice node is reached, the option of which branch to select is obtained from the `Non-deterministic Engine` (again, by passing the id of this occurrence).

6.5 Important design decisions and their rationale

In the following, we discuss the two important design decisions regarding the concurrency model and time semantics of simulation being followed.

6.5.1 Object concurrency model

We used the Active object model [4] to handle the concept of a concurrent object. In our case, most of the environment components are considered as active objects. This is because they operate independently in the RTES environment and can communicate asynchronously with each other and the SUT. These objects have their own thread of execution and receive asynchronous messages that are handled using an event queue. For example, *Sorter* shown in Fig. 3 is implemented as an active object and executes independently from the other environment components, such as *RVM* and *Item*.

An active object simulating an environment component can have multiple internal threads associated with it. These threads correspond to the parallel regions of the state machines. In our motivating example, *RVM* (Fig. 6) has two internal threads, each for a parallel region (*ItemInside*, *Routing*). This was required to simulate the behavior of an *RVM* when routing and handling item insertion at the same time. In other cases, the parallel regions were used for modeling component failures that could happen at any time independently of the current state of the component.

6.5.2 Time semantics

Typically, in simulation approaches, the aim is to simulate and analyze the behavior of a system or environment before it is actually built. For the type of simulator that we have developed, the aim is to simulate the environment in order to test the RTES in diverse situations, without involving actual hardware or people. The SUT in our case is always the actual executable production code and is seen as a black-box. We have no control over the SUT behavior and its definition of time. Therefore, there is no point in simulating the SUT clock, unlike the case in typical simulation approaches such as SystemC.[4] From its point of view, the SUT interacts with the simulator as it would interact with the actual environment. The time it takes for SUT to process a message will be same in both cases. The notion of time for the environment is therefore based on the software implementation of the physical time. In our case, we used the implementation provided by Java time semantics which are based on the CPU clock.

A typical issue in using the CPU clock is the jitter that might be introduced because of computation overhead on the processor (e.g., garbage collector, other operating system processes). This jitter can range up to a few milliseconds. Fortunately, for the type of environments for the systems that we tested, as in many embedded applications, a delay of few milliseconds was not a major issue as the time events were generally in the magnitude of seconds (for example the time of a bottle traveling on a conveyor). To be on the safe side, we explicitly executed the Java garbage collector before and after the simulation. Moreover, for the experiments that we conducted, the garbage collector never executed during simulation and we never faced synchronization issues due to jitter. For the type of industrial systems that we were focusing on (see Sect. 2), using Java was sufficient. Figure 17 shows the plot for the jitter (difference between actual value of time event with the simulated value) corresponding to 16,000 time events obtained after a number of simulator executions of an artificial problem (details about the various artificial problems are discussed in Sect. 8.1). The time events are plotted on x-axis, whereas the y-axis shows the jitter in terms of

[4] http://www.systemc.org/, accessed on 04/01/2012.

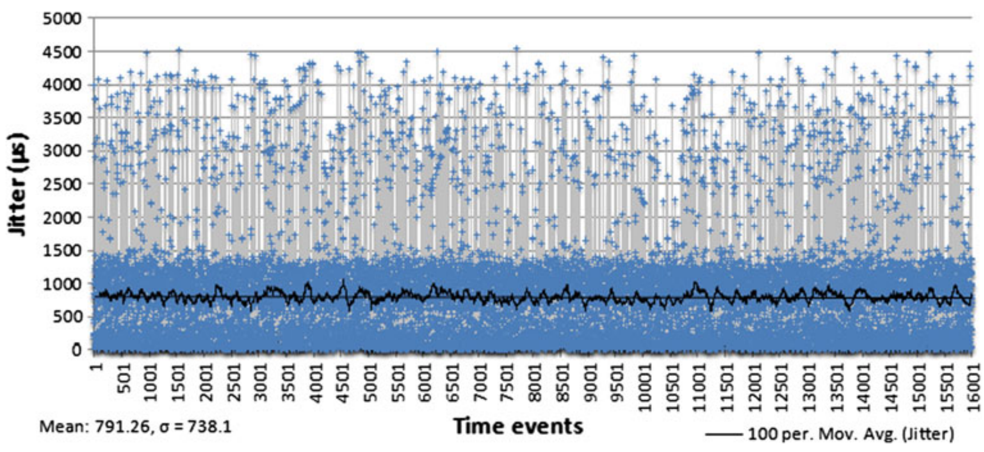

Fig. 17 Jitter in microseconds for time events during simulation

microseconds. The values of the time events were selected randomly. The graph shows that the jitter is less than 4.5 ms (4,500 µs) at most and less than 1 millisecond on average. For this experiment we used a Dell Precision M4600 with Intel Core I7 (2.50 GHz, 8 MB cache, Quad Core), 16 GB RAM, and running Ubuntu Linux 10.0.4. This is the same machine that we used for testing one of our industrial RTES.

For the environments which have hard time constraints from the system, for magnitudes of milliseconds or less, the jitter can be a critical problem. A possible solution to address this problem is to use real-time Java virtual machines (e.g. Sun Java Real-Time System [70]) running over a real-time operating system (e.g. SUSE Linux Enterprise Real Time Extension [71]), which will result in nanosecond level accuracy. The code that our tool generates is in theory compatible to run with real-time Java virtual machines since this is one of the requirements of such virtual machines. Though, the practical implications of doing this for testing hard real time systems still remain to be investigated.

6.6 Resolution of UML semantic variation points

A number of decisions regarding the semantics of implementations in UML are kept open to interpretation by tool developers, and are referred to as semantic variation points (SVP). In this section, we discuss how we resolved the explicitly mentioned UML SVPs (in the UML specification document [4]) in our context. These include the SVPs corresponding to state machines, classes, and behavior modeling.

6.6.1 Execution semantics and order of events in queue

The state machines of environment components are implemented using the run-to-completion semantics as specified by the standard UML [4]. For handling asynchronous messages, active objects need to implement event pools. The implementation of such event pools is a UML SVP. In our context, we used the `PriorityBlockingQueue` Java class to implement these queues. They are priority queues and hold various events during the life cycle of an environment component.

The decision of ordering of events in the event queue is also a SVP in UML (see the discussion on `Behaviored Classifier` in UML specifications [4, p. 449]). In our context, we resolved this by assigning the time events in the queue the highest priority, change events the second highest, and the signal events the lowest priority. Time events have the highest priority since the behavior that they trigger is explicitly related to time, so it should be executed as close to the event occurrence time as possible. Change events have higher priority than the signal events since it is important to execute the corresponding behavior at the earliest opportunity once the change condition turns to true, as it may become false again. For the events of the same type, the order in the event queue is determined based on the ids assigned to each event (discussed in Sect. 6.4.2).

6.6.2 Default entry and handling conflicting triggers

Another UML semantic variation point is the decision about the default behavior of state machines when there is no explicit initial pseudo state defined in an enclosed region (e.g., in sub states) (see the discussion on *State* in UML specifications [4, p. 566]). Our environment modeling methodology requires the modeler to put at least one initial pseudo state in every enclosed region. A region without an explicit initial transition will be considered ill-formed.

There can be situations during simulation when one outstanding event satisfies multiple triggers in an environment component. This issue is left as a semantic variation point in UML (see the discussion on BehavioredClassifier in UML specifications [4, p. 449]). For our methodology, when such a case arises, the event that has the minimum id (calculated as discussed in Sect. 6.4.2) among the satisfied events is selected and triggered. Our environment modeling methodology recommends avoiding such situations as they indicate imprecise or incomplete environment models.

6.6.3 Event not satisfying any trigger

The behavior in the cases when the occurring events do not specify triggers on active states are left as semantic variation points in UML (see the discussion on `Behaviored Classifier` in UML specifications [4, p. 449]). In our case, all the events that do not satisfy any trigger are simply ignored. The modeling methodology requires the modeler to only model those triggers that have a significant behavior, e.g., they have a corresponding effect. If accepting a signal coming from the SUT in some state represents a faulty behavior (i.e., that signal should not have been sent), then it should be modeled with a transition leading to an «Error»state. For example, in the sorting machine, a *Sorter* should never accept a signal `item_at_destination()` when it is in *MovingLeftCentre*, and so we modeled this with a transition from the moving state to an error state, as shown in Fig. 5.

6.6.4 Event evaluation time

In UML semantics, the time it should take for a component to dispatch an event after it was received is not defined and is left as a semantic variation point to be decided by specific methodologies (see section on *Triggers* on p. 472 of UML specifications [4]). In our methodology, this time is dependent on the event queue size of the receiver, the priority of the event, and the time to enqueue and dequeue the events before consumptions.

For the specific case of time events (corresponding semantic variation point discussed in `TimeEvent`, UML specifications [4, p. 468]), as discussed earlier (Sect. 6.4.2), when a time event is received by the environment component, it is placed in the event queue. If there are more time events ready to be dispatched in the queue, then the event received may never be executed. If a time event already at the front of the queue is executed, it will result in a time transition and hence, the queue will be emptied from all its events, unless for specific cases already mentioned in Sect. 5.3.4 (e.g., for transitions with «Lazy»). If the time event belongs to a parallel region (or a substate of a parallel region) then it will again be placed in the region's queue (see Sect. 6.4.2). Thus, the dispatch time of a time event depends on the time events that are already in the queue(s) and the time to enqueue and dequeue in the queue(s), and the number of internal orthogonal regions. Overall this time will only be within a few milliseconds, which is not a major obstacle for the type of system testing that we deal with and for many embedded applications (as discussed in Sect. 6.5.2).

6.6.5 Signal transmission

The mechanism of how a signal is transmitted from the source to target component is also left as a SVP in UML (See pp. 437, 447, 466 of UML specifications [4]). As discussed earlier, in our context, signals are transmitted from the source to the target by calling the target's `receiveSignal()` method with the signal parameters. The `receiveSignal` method creates an instance of `EventInvocation` and puts it in the event queue of the target component. The sender and receiver will be running on the same process since all the environment components run on a single process. This decision was made because intra-process communication is easier and faster than inter-process communication. Since SUT is a black-box in our testing approach, the SUT runs on a separate process (or even a separate machine) and the communication between environment components and SUT is handled through the external action code.

6.6.6 Other variation points

In this subsection, we discuss the remaining important semantic variation points, which have already been resolved as a result of our extension of the state pattern and have already been discussed in that context. The decision of how and when change events are to be evaluated is a SVP (see Change Events, p. 452 of UML Specifications). We have already discussed this in Sect. 6.4.2. Another SVP related to change events is what to do with them in cases where between the time of dispatch and execution, the condition of the change event becomes false again (Change Events [4, p. 452]) . As discussed in Sect. 6.4.2, such events will be considered as lost events and will be ignored.

The details on how the run-to-completion semantics are implemented are also left open as SVP in UML [4, p. 452]. The discussion of the state pattern extensions in 6.2 and the discussion in Sect. 6.4 regarding code generation for various state machines, elaborate how these semantics are implemented in our context.

The mechanism by which a mapping between a signal and corresponding operation is performed is also a SVP [4, p. 447]. In our context, we generate operations with the same signature (i.e., same name and parameter types) for the signals (as discussed in Sect. 6.2). The sender of a signal sends the name of the signal to be invoked as a String value to the receiver. As discussed in Sect. 6.2, the receiver object uses Java Reflections to identify the operation to be invoked. This resolves another UML SVP about how to identify the invocation corresponding to an event [4, p. 438]. The decision of whether to pass the arguments by reference or value [4, p. 438] is also resolved as we pass the arguments of the signal as Java Objects, which are passed by reference in Java.

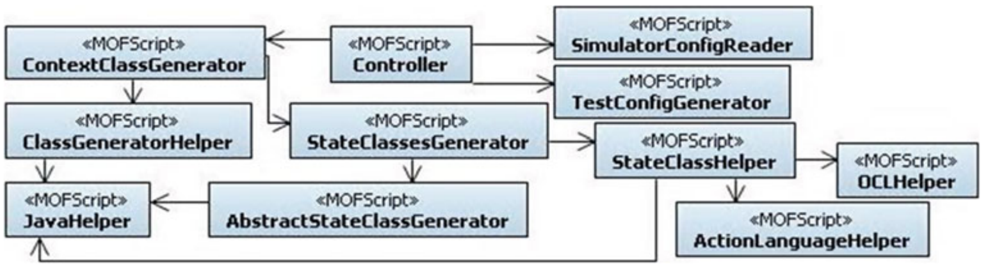
«MOFScript» ContextClassGenerator
«MOFScript» Controller
«MOFScript» SimulatorConfigReader
«MOFScript» TestConfigGenerator
«MOFScript» ClassGeneratorHelper
«MOFScript» StateClassesGenerator
«MOFScript» StateClassHelper
«MOFScript» OCLHelper
«MOFScript» JavaHelper
«MOFScript» AbstractStateClassGenerator
«MOFScript» ActionLanguageHelper

Fig. 18 MOFscript transformations used for generating simulators

6.7 Automation

We implemented the rules mentioned earlier in Sect. 6 by using MOFScript model to text transformations [68]. Figure 18 shows various MOFScript transformations that we developed for transformation from environment models to Java code. These transformations are contained in the package named `MOFScript Transformations` shown in Sect. 6.1. Stereotype «MOFScript»denotes the MOFscript m2t files containing the transformation rules.

The control of transformations is handled by `Controller`. The `ContextClassGenerator` transformation is responsible for transforming the domain model with the help of `ClassHelper`. `ContextClass Generator` also calls `StateClassesGenerator`, which is responsible for transforming the state machine of an environment components to Java code with the help of `StateHelper`. `JavaHelper` contains the rules specific to Java and are used by various other transformations; `OCLHelper` uses the `OCLToJavaTranslator` class discussed in Sect. 6.1 to convert OCL expressions to their equivalent Java code. `ActionLanguageHelper` is responsible for modifying the action language code in the models according to the generated code structure. `SimulatorConfigReader` reads the configuration file given as input and the `TestConfigGenerator` generates a configuration file that is read by the test framework.

The tool requires models exported as standard EMF format for UML (.uml file). This is a widely accepted format for interchanging UML models and is supported by a number of modeling tools, including Rational Software Architect and Papyrus. Our tool reads the .uml file and generates a simulator corresponding to the models. The software engineer then needs to provide a test driver that specifies the testing strategies to be used and that initializes the SUT and the generated environment simulator.

7 Interaction with test framework

In this section, we first discuss the details about how the environment simulator interacts with the framework, which is followed by discussions on how the information from the models is used for search-based testing (Sect. 7.1), more details on simulator configuration (Sect. 7.2), the OCL constraints solver that we used (Sect. 7.3), and finally details about test drivers and JUnit test cases (Sect. 7.4).

The test framework queries the simulator to obtain information required to generate the simulation configuration (e.g., number of non-deterministic variables and their value domain and type). The test framework then generates valid simulation configuration*s* based on the testing strategy in use (e.g., at random for random testing) and uses the *Test Driver* to run the SUT with the environment simulator. The test framework uses a set of heuristics based on the previous test case executions (e.g., rewarding test case diversity and Genetic Algorithms) to choose new test cases to run and evaluate (for details see Sect. 7.1 and [7]).

The goal of the test framework is to find a simulation configuration for which, once executed with the simulator, an error state is reached (if any fault is present in the SUT). Once such a configuration is found, it automatically generates a *JUnit* test case. In [7], we show how the models developed using the methodology described above can be used for automated system testing of RTES. A test case is used to define two important components of the simulation: (1) the configuration of the environment, e.g., number of sensors/actuators and their initialization; (2) the non-deterministic events in the simulation, e.g., variance in time-related events such as physical movements of hardware components, occurrence and type of hardware failures and actions of the user(s). In [7], we investigate three strategies to automate these choices: Random Testing, Adaptive Random Testing and Search-Based Testing (using a Genetic Algorithm). The results of the experiments (on three artificial problems and one large industrial RTES) showed that environment model-based testing is able to automatically find faults in all the considered case studies. In particular, previously uncaught critical faults were automatically found in the industrial RTES [44].

Test data generation can be reformulated as a *search problem* [72,73], in which for example the goal can be to find test data for which failures are triggered (if any fault is present in the SUT). To achieve such goal, it is important to have a heuristic to evaluate how good test data are, even if they do not trigger any failure. For example, test data that lead to execute most of the code of the SUT would a priori be more useful than test data for which the computation finishes very quickly after executing few lines of code. A *search algorithm* would use such information to focus on areas of the search space (i.e., test data) that are more likely to contain test data for which the SUT fails. Since it is not feasible to evaluate and run all possible test data, a search algorithm has to focus only on some promising areas. Such type of test data generation is referred to as *search-based software testing* [72], and there are many search algorithms and *fitness functions* (i.e., functions to evaluate how good the test data are). For details regarding how to use search algorithms to system testing of RTES, see [44].

However, to use such search algorithms, it is important to obtain information on the execution of test cases after they are run (e.g., which parts of the SUT code they execute). Such information would be used to compute the fitness function. Which type of information is needed depends on the selected testing strategy.

Typically, in white-box testing, when information regarding source code execution is needed for the heuristics, the code of the SUT needs to be *instrumented*. Instrumentation means that the code is augmented with probes to collect execution data (e.g., to check which branches of "if" statements are executed). Since our system testing approach is based on environment models where the SUT is treated as a black-box, the probes are inserted only in the code of the environment simulator. The models contain valuable information to guide search-based testing, and such information would be lost or difficult to reverse-engineer after the simulator is generated. For practical reasons, the probes are automatically inserted when the code is generated from the models. This is another advantage of using simulators generated from models rather than manually coding a simulator, as manually inserting those probes would be likely tedious and time consuming. The rest of the section is organized as follows: Sect. 7.1 discusses various search heuristics that are used to guide search-based testing strategies, which is followed by a discussion on simulation configuration (a fundamental part of a test case in our context) in Sect. 7.2. Section 7.3 discusses the OCL constraint solver that is used by the test framework, whereas Sect. 7.4 discusses test driver (an input of the test framework) and JUnit test cases (the key output of the test framework).

7.1 Search heuristics

In the following, we discuss four types of instrumentation to collect test case execution information used in fitness functions in the context of environment-based system testing for RTES [44], namely *approach level*, *branch distance*, *time distance*, and *risky states*. Because the goal of such type of testing is to find simulation configurations for which error states are reached, we collect information regarding *how close* the execution was from reaching any of these error states.

The *approach level* is a common heuristic [72] in white-box, search-based software testing, where executions that get close to the testing target (e.g., branches in branch coverage) are rewarded. When a test case is run, several states in the environment state machines are reached, while others are not. The approach level calculates the *minimum* number of transitions required to reach any error state from any visited state of the environment component. To obtain this value, we consider the state machine as a graph and perform a breadth first search on each state to obtain the minimum distance (in number of transitions) to reach the error states. This calculation is done only once, when the simulator code is generated, and then hard-coded directly in the simulator code to ease fitness computations during simulation. When a test case is executed, the approach level for all reachable error states is calculated and reported. For example, consider the environment component *Sorter* (Fig. 5): the distance for the *Sorting::Working::Error* state from *Sorter::Working::Left* state is two whereas such distance is one from *Sorter::Working::MovingCentreRight*. If both these two non-error states are reached during test case execution, then the approach level would be the minimum value among those two values (i.e, 1).

The second piece of information used in the fitness functions is the *branch distance*. To reach an error state, it is necessary to follow some specific paths in the state machine. A path would be a sequence of state transitions, driven by triggers. However, state transitions often have *guards* (e.g., logical predicates expressed in OCL), which need to be satisfied (i.e., their predicates need to be evaluated to true) to take such transitions. The predicates in these guards depend on variables, which values cannot be directly manipulated [73] by the test framework and depend on the entire test case execution carried out so far until the guard is evaluated. Some guards can be difficult to satisfy (i.e., only few simulation configurations lead to it) and, because the variables in the guards cannot be directly manipulated, it is not possible to use external *constraint solvers* to satisfy them. The *branch distance* is a heuristic to reward simulation configurations that brings the guards closer to satisfaction. Consider for example the guard "x==0", and two test cases for which "x=1" and "x=100". None of the test cases satisfy that guard but the case "x=1" is heuristically closer. The branch distance is a common and effective heuristic in search-based software testing for structural coverage [72]. In previous work, we have developed a search based constraint solver, in which we extended the branch distance functions for white-box testing to support all the constructs of OCL constraints [73]. In this paper, when we generate Java code to represent the OCL guards, we instrument such predicates to calculate their branch distance each time they are evaluated. For the details of how these branch distances are calculated, see [73].

The third type of information used in the fitness functions is the *time distance*. In some cases, a transition is taken only after a timeout, and this type of transitions can appear on the paths that lead to the error states. For example, assume that, in a particular state, the environment expects a signal from the SUT within one second.

If such a signal is not received, then an error state is reached. In a state machine, this would be modeled as a transition to an error state with trigger *after (1, s)*, whereas receiving the signal from the SUT would trigger a transition toward another state. The *time distance* calculates how much longer it would have taken to get a given time trigger fired. Taking the example above, if we receive the signal after one millisecond, it would be worse than receiving the signal after 900 ms, although in neither of the cases the error state is reached. Each time there is a time transition, during code generation such transition is instrumented to calculate its time distance.

The fourth type of information used in the fitness functions to guide the search is about risky states. The states that have a direct transition to error states are considered to be *risky* states. For the search, this information is important as these are the closest states to the error states. For example, in the *Sorter* component, the state *Sorter::Working::MovingCentreRight* is a risky state. How often a risky state has been reached, and for how long the environment was in such risky states, can be used by the search algorithms to reward test cases that keep the environment in these risky states as long as possible.

7.2 Simulation configuration

The simulation configuration is generated by the *Test Framework*. During the simulation, the simulator queries the simulation configuration to obtain the values of non-deterministic occurrences, e.g., exact time in time event. For this purpose each non-deterministic occurrence is assigned a unique id during the simulation. Notice that, once a configuration is defined, the simulation becomes deterministic. In other words, executing again the simulator environment with the same simulation configuration should result in the same behavior. However, this latter point is not strictly correct, because a simulation would still be affected by non-deterministic components such as the thread scheduler and other operating system resources. Fortunately, this is not a serious problem for the type of system level testing done here where, for most environments, variances of few milliseconds in the interactions between the environment and SUT are simply negligible as they have no impact on the resulting states of the environment and SUT. When this is not the case, as further discussed in Sect. 6.5.2, the modeling methodology and code generated are still valid but a real-time operating system and Java RT would need to be used. Therefore, for all practical purposes, a test case is uniquely characterized by a simulation configuration.

As an example consider Table 2 that shows a random generated simulation configuration. The simulation configuration shown is based on an environment configuration having 1 *RVM* instance, 1 *Sorter* instance, 1 *User* instance, and 3 *Item* instances. This environment configuration results in a total of twelve non-deterministic occurrences, 2 for *Sorter*, 1 for *User*, and 3 for each of the three instances of *Item*. Each instance has a unique id during the simulation (“Instance Id”). Each of the non-deterministic occurrences has a unique id during simulation as shown by the “Nondeterministic Occurrence Id” column. The “Related Property” column in the table shows properties of the environment components that are related to the non-deterministic occurrences. When a non-deterministic occurrence is based on a transition with «TimeProbability», the stereotype is mentioned in the column. The values for these occurrences selected by the test framework are shown in the column labeled “Value”. In the case of «TimeProbability», each value pair specifies the choice of value as 1 or 0 referring to whether or not to take the transition and the time in milliseconds at which the transition is to be triggered (irrelevant when the transition is not to be taken). Other values in the column are assigned by the test-engine based on the ranges (e.g., the upper and lower bounds) of «NonDeterministic» environment component properties. The ranges are shown in the domain model as stereotype properties of the environment component properties (Fig. 3). For example the lower and upper bounds for *moveArmTimeLC* are 280 and 320. The value in the simulation configuration decided by the test framework is 292.

7.3 OCL constraint solver

According to the modeling methodology, the domain model captures the different forms the SUT environment can take (see Sect. 5.2). For a given test execution, we need to select one possible environment configuration. We use an OCL constraint solver to generate a possible configuration from the domain model. This consists of selecting appropriate initial values for the association multiplicities and attributes that are not labeled with the «NonDeterministic» stereotype, as the latter are determined by the simulation configuration. Figure 4 shows an example of OCL constraint on the *User* component of the domain model of the Sorting machine case study (Fig. 3). The constraint specifies that a *User* instance is always associated with a non-empty collection of items. To obtain an appropriate value of an instance according to such constraints, we have developed a search-based OCL constraint solver [73], since current OCL solvers in the literature do not scale up to the complexity of real constraints

Table 2 A simulation configuration for the sorting machine

Environment component	Instance id	Nondeterministic occurrence id	Related property	Value
User	0	0	insertionTime	500
RVM	1	None	None	N/A
Sorter	2	6	moveArmTimeLC	292
Sorter	2	7	moveArmTimeCR	303
Item	3	9	type	1
Item	3	10	timeToNode	1,062
Item	3	11	«TimeProbability»	0, 0
Item	4	12	type	0
Item	4	13	timeToNode	970
Item	4	14	«TimeProbability»	1,300
Item	5	15	type	0
Item	5	16	timeToNode	1,011
Item	5	17	«TimeProbability»	0, 0

found in industrial systems. The generated simulator code calls this solver to generate values for which the OCL constraints are satisfied.

7.4 Test driver and JUnit test case

A Test Driver needs to be written by the tester, which is used to start the execution of the simulator and SUT, and to stop them after a timeout (a set of predefined libraries are provided to help the tester in this task, wrapper by the `SimulaVerdeDriver`). In the case studies for this paper, the environment simulator and the SUT are run on different processes on the same machine. Figure 19 shows an example of a test driver for the *SortingMachine* case study. The `SimulaVerdeDriver` class provides a number of methods for configuring various testing options.

Whenever the test framework leads the simulation to an error state, this is due to a faulty implementation of the SUT. The specific simulation configuration of the simulator at that time is embedded in a *JUnit test case* and a source file representing the test case is generated by the *Test Framework*. This is done so that the simulation configuration is saved and can be executed later for debugging purposes. The *JUnit test case* calls the *Test Driver* based on a simulation configuration. Assert statements are automatically added to check whether any error state has been reached during the simulation. Once generated, these *JUnit test cases* do not need the test framework for their execution. Figure 20 shows an auto generated test case for the Sorting Machine case study. The test case is based on the simulation configuration shown in Table 2. The class `ProblemData` used in the JUnit test case holds information of various non-deterministic occurrences.

The method `getTestCaseData()` creates and returns an object of class named `TestCase` based on a simulation configuration decided by the *Test Framework*. The implementation of the method shows the various values being assigned to non-deterministic occurrences. Class `ProblemData` used in the JUnit test case (see line number 9 in Fig. 20) is supposed to hold information related to various settings of the *Test Framework*, such as the environment configuration and the total time for simulation. Objects of class `Environment` represent environment configurations for the simulator. The execution trace for the test case along with its comments can help guide the testers to the source of the problem.

8 Case study

In this section, we discuss the case study we conducted to evaluate the proposed modeling methodology and simulator generation. Note that we followed the guidelines of reporting case studies in software engineering presented by Runeson et al. [74,75]. First we discuss the design of the case study in Sect. 8.1, which is followed by case study procedure (in Sect. 8.2) and results (in Sect. 8.3).

8.1 Case study design

In this section, we discuss the design of the case study, which includes the objectives of the case study and a description of various cases used.

The objective of the case study is to evaluate whether (1) the transformation rules are sufficient to convert environment models of different complexity levels, and belonging

```
import simula.embt.commons.SUT;
import simula.verde.drivers.SimulaVerdeDriver;
import simula.sorter.sut.SortingController;
public class TestDriver {
 public static void main(String args[])
 {
    //helper class provided with framework
    SimulaVerdeDriver svd = new SimulaVerdeDriver();
    /* configuration for testing */
    svd.setTestStrategy(SimulaVerdeDriver.ART_STRATEGY); //selecting the test strategy
    svd.setSeed(0);        //initial seed for the strategy
    svd.setRunTimeMinutes(7200);  //how long testing should continue
    svd.setTestCaseTimeoutMillis(10000);  //duration for each test case
    svd.setJunitOutputDir("User_Driver_Demo/");  //output directory for test cases
    SUT sut = new SortingController();    // System under test
    sut.setVerbose(true);  //whether SUT should be verbose
    sut.setPort(1249);     //initial port on which SUT is listening
    svd.setSUT(sut));
    /* setting environment details */
    svd.setInputModel("res/Sorter.uml");  //.uml file containing environment models
    svd.setSrcDirectory("generated_code/"); //output directory for simulator code
    svd.setPackageName(“simula.sorter.env”); // package name for the simulator classes
    svd.generateCodeAndTest(); //call to generate code and test cases
 }
}
```

Fig. 19 An example of a test driver for sorting machine case study

```
public class Test_Sorter {
 @Test
 public void testCase(){
    ProblemData pd = new sorter.embt.SorterProblemData();
    TestCaseRunner runner = new TestCaseRunner();
    runner.init(pd);
    TestCase tc = getTestCaseData(pd);
    Environment env = pd.getEnvironment();
    try{    runner.runTestCase(tc);       }
    catch(Exception e){    fail(e.toString());    }
    assertEquals(false, env.hadError());
 }
 protected TestCase getTestCaseData(){
    RingTestCase tc = new RingTestCase();
/*Item (id: 5): TimeProbability, source: On_Belt, target: Lost*/
    tc.setVariable(17, 0, 0);
/*Item (id: 5): timeToNode, source: On_Belt, target: At_Node*/
    tc.setVariable(16, 1011);
/*Item (id: 5): type*/
    tc.setVariable(15, 0);
/*Item (id: 4): TimeProbability, source: On_Belt, target: Lost*/
    tc.setVariable(14, 1, 300);
/*Item (id: 4): timeToNode, source: On_Belt, target: At_Node*/
    tc.setVariable(13, 970);
/*Item (id: 4): type */
    tc.setVariable(12, 0);
/*Item (id: 3): TimeProbability, source: On_Belt, target: Lost*/
    tc.setVariable(11, 0, 0);
/*Item (id: 3): timeToNode, source: On_Belt, target: At_Node*/
    tc.setVariable(10, 1062);
/*Item (id:  3): type*/
    tc.setVariable(9, 1);
/*Sorter (id: 2): movingArmTimeCR, source: MovingCentreRight, target: Choice*/
    tc.setVariable(7, 303);
/*Sorter (id: 2): movingArmTimeLC, source: MovingLeftCentre, target: Choice*/
    tc.setVariable(6, 292);
/*User (id: 1): insertionTime, source: Idle , target: RVM_busy*/
    tc.setVariable(0, 500);
    return tc;
 }
}
```

Fig. 20 An auto generated JUnit test case for sorting machine case study

Table 3 Environment models for artificial problems and industrial cases

Case	EC	States			Trans	Guards	Signal events	Time events	Change events	Profile elements			
		Simple	Orth	Comp						NDV	TP	Error	Failure
AP1	1	3	0	0	5	4	2	1	0	1	1	1	1
AP2	1	6	2	0	7	0	0	3	1	1	1	1	1
AP3	2	9	0	0	13	2	7	6	0	6	0	1	0
IC-A	4	20	2	1	38	11	16	5	1	5	1	2	1
IC-B	3	23	3	1	46	8	20	7	3	3	7	3	4

EC environment components, *Orth* orthogonal, *Comp* composite, *Trans* transition, *NDV* non-deterministic variables, *TP* time probability

to various domains, to simulator code (completeness of the transformation rules), (2) the automated generation of simulators is likely to significantly reduce development effort (effect on development effort), (3) the generated simulators enables the detection of failures in RTES system testing (effectiveness in test automation), and (4) the transformations implemented are correct (correctness of transformations). Note that we are not providing a proof of correctness or completeness; rather we are just reporting empirical data to increase the confidence in correctness and completeness of transformation.

We selected five different RTES as part of our study. Two of the cases were industrial RTES. One industrial RTES, Industrial Case A (IC-A) is the sorting machine system that we have discussed as a motivating example throughout the paper.[5] As mentioned earlier, in this paper, we are only considering a subset of the case study focusing on the sorting functionality, having four environment components and an average of five states per component.

The second industrial system, Industrial Case B (IC-B), is a marine seismic acquisition system, which has five environment components with an average of 12 states per component. Even though only looking at the number of components, this may suggest that the case studies are simple, at the time of testing, there can be hundreds of instances of each component, as it was the case with IC-B where hundreds of sensors were communicating with the SUT at run time. The aim was to select two industrial systems that belong to different domains, with different functionalities, to study diverse environment models. We also developed three artificial problems of varying complexity that also belong to different domains. Two of the artificial problems (AP1 and AP2) were inspired by one of our industrial case studies and deal with RTES interacting with multiple sensors in different situations. The third artificial problem (AP3) is inspired by a train control gate system discussed in [76]. The RTES for these artificial problems were developed in Java. For the industrial case (IC-A), a hardware interface layer was not separated from rest of the code while the RTES was being developed (as it was for IC-B). As a refactoring task to improve the testing processes, the software engineers are currently working on separating out the code that deals with hardware components and providing a standard mechanism of communication for the SUT. Since the adapter was not yet available at the time of writing, we manually developed the portions of SUT that are related to the subset of the case being used in this paper. However, this has *no effect* on the code generation discussed in this paper, as the environment models would be *exactly* the same for the actual SUT. To develop the environment models for the two industrial case studies, we conducted several interactive sessions with our industry partners. More details on lessons learned regarding the industrial application of the methodology are provided in [77]. Table 3 shows the statistics of various modeling elements used in the five cases.

To evaluate completeness and correctness of the transformation rules, we also created several test models. The models were not linked to any SUT and were only developed in order to cover different sets of modeling elements according to our environment modeling profile. In total the test models comprised of 50 components, with each component having on average of 4 states and 12 transitions. They covered various state machine and class diagram constructs, and all the modeling features defined by the environment modeling methodology.

8.2 Case study procedure

This section describes how the case study was conducted and data was collected for each of the four evaluation criteria.

8.2.1 Completeness of the transformation rules

Evaluating the completeness of model transformations is still an open research question [78]. Note that in this paper, we have only discussed the most important rules for simulator generation. To evaluate completeness, we generated simula-

[5] Notice that in [1] we considered the entire sorting machine case study. In this paper, we only discuss the subset of the case study that we used for testing and simulator generation. Therefore, the data presented in this paper is not exactly the same as in [1].

Table 4 Summary of procedure to evaluate the correctness of transformations

Artifacts	Property				
	Syntact correctness	Semantic correctness	Conformance to models	Heuristics reporting	Failure detection
Input	Test models, AP1–AP3, IC-A, IC-B	Test models	Test models, models for AP1, AP2, AP3, and IC-A	Test models, models for AP1, AP2, AP3, and IC-A	Models for AP1, AP2, AP3, and IC-A
Execution procedure	Manual drivers	Manual inspection	Manual test cases	Manual test cases	Manual test cases
SUT	None	None	None for test models, stubs for artificial problems and IC-A	None for test models, stubs for artificial problems and IC-A	Buggy and Correct versions for AP1, AP2, AP3, and IC-A
Oracle	Java compiler	Checking compliance with extended state pattern	Comparison with source models	Manually added in the test cases based on source models	Failure reporting mechanism during simulation

tors for the five cases, each having different level of complexity and modeling different concepts. We also generated simulators for the different test models that covered different sets of modeling elements.

8.2.2 *Effect on development effort*

To evaluate this, we generated simulators for the five cases and obtained size and complexity information about the generated code. When compared to the size of the models, such data can help assess the potential amount of effort saved by generating simulators from models rather than developing the simulators manually. Of course, we can only provide quantitative insights because the qualitative results depend on the skills of developers with respect to modeling and coding.

8.2.3 *Effectiveness in test automation*

To assess this, we manually seeded non-trivial faults in the three artificial problems and industrial case A (one fault for each SUT). We could have rather seeded those faults in a systematic way, for example by using a mutation testing [79] tool. We did not follow such procedure because the SUTs are highly multi-threaded and use a high number of network features (e.g., opening and reading/writing from TCP sockets), which could be a problem for current mutation testing tools. Furthermore, our testing is taking place at the system level, and though small modifications made by a mutation testing tool might be representative of faults at the unit level, it is unlikely to be the case at the system level for RTES. For the SUT of Case B, a previously uncaught critical fault was found with our test framework [44] and used to assess the effectiveness of the simulator for test automation. We ran the simulators generated for the five cases (i.e., the three artificial problems and two industrial cases) with various testing strategies to evaluate whether the test cases that were expected to fail did and whether no other test cases failed.

8.2.4 *Correctness of transformations*

Correctness of the transformation rules is about how correct are the transformation rules in generating simulators that behave as expected according to the environment models. Evaluating the correctness of model transformations is still an open research question [78] and no standard mechanism or appropriate tool is available for testing them. To evaluate the correctness of our transformations, we focused on both structural correctness (the code is what it is supposed to be) and behavioral correctness (the code works as it is supposed to work) of the generated code. We evaluated the structural correctness of the code keeping in mind two properties: *syntactic correctness* (the generated simulators are syntactically correct, i.e., no compilation errors) and *semantic correctness* (the generated code is what it should be according to the extended state pattern). To evaluate that the behavior of the generated simulators is correct, we manually developed test cases keeping in mind three properties: (1) behavior of the generated simulators conform to what is specified in their environment models, (2) the simulators correctly report the information required to guide test heuristics, and (3) the simulators correctly detect and report failures in SUT.

To evaluate the transformation rules, we adopted a procedure that is summarized in Table 4. To evaluate syntactic correctness, we created a number of test models that contained different set of modeling elements for state machines and class diagrams, generated simulators for them, and used the Java compiler to compile the generated simulator code. For semantic correctness, we inspected the generated code for the developed test models to see if the code was conforming to the extended state pattern.

To evaluate the effectiveness of simulators to correctly detect and report failures in SUTs, we created two versions of SUTs for the three artificial problems and industrial case A: one version was a bug free version and for the other one we seeded a fault manually (same as we did for evaluating effectiveness in test automation). We created two test cases for each artificial problem and industrial case A, one test case was supposed to detect and report the failure corresponding to the seeded fault. The other test case was not expected to detect the fault. We ran the two test cases on the two versions of the SUT and observed their behavior. Our assumption was that a faulty simulator might lead to falsely reporting a bug in the correct SUT or prevent the triggering of an expected failure in the faulty SUT.

The above strategy was iteratively applied to check correctness and achieve a stable version of the tool. At any rate, testing cannot prove the absence of defects, although it increases our confidence.

8.3 Results

In the following, we present the results of the case study for each evaluation criterion.

8.3.1 Completeness of the transformation rules

As far as generating simulators from environment models for the five cases and test models presented above, the transformation rules are complete. These test models along with the three artificial problems and two industrial cases covered all the modeling elements defined in the methodology. The MOFScript transformations developed were able to generate Java code for all of the UML/MARTE/OCL model constructs used in the case study artifacts and the test models.

8.3.2 Effect on development effort

When using our methodology, the only significant effort required by the software engineers is to create environment models for the environment of RTES. Once developed, these models are used for generating the simulator, executable test cases, and automated oracles. Table 3 shows relevant size data for the various environmental models and Table 5 summarizes the details for corresponding generated simulators for the five cases (three artificial problems and two industrial case studies) in terms of the total number of generated classes, number of methods, threads, and lines of code. The first three rows of the tables show data about the three artificial problems (AP) and the last two rows about the two industrial cases. Note that, even though the number of components in each case is small, during the simulator execution a number of instances are generated for each environment component. As discussed earlier the number of instances to be generated is decided based on the OCL constraints on the domain model and is specified in an environment configuration. For example, in industrial case B, as shown Table 3 the total number of environment components is three, but for one of the components, hundreds of instances were created for simulating the environment.

One of the reasons for the large number of generated classes, methods, and lines of code is the use of the state pattern, which requires a separate class for each state. For example, industrial case A has only four environment components, but because of 23 simple, 3 orthogonal, and 1 composite state over 5,000 lines of simulator code were generated with 35 classes and 386 methods. Even if the code were manually written, we would expect developers to follow a similar pattern (extension of state pattern) in order to facilitate changes, which would have resulted in a similar number of lines of code. This conjecture is supported by the fact that, in one of the industrial case studies, an existing, manually written simulator was of similar complexity. The simulator is a complex, multithreaded application and various components run in parallel. If the simulator were to be manually written, the developers would have had to resolve synchronization issues related to concurrency. For example, in IC-B, the generated simulator included 20 threads to handle active objects and various timers. With a large number of possible instances running during an environment simulation (as hundreds of instances for IC-B) the overall behavior of the environment is quite complex. In our approach, the simulator generator takes care of these issues and the software engineers only have to develop the environment models, which are expressed at a higher level of abstraction than the source code.

The statistics about the generated code are only used to provide an estimation of the complexity of the simulators.

Table 5 Details of generated simulators

Case	Classes	Methods	LOC	Threads	Manually written LOC
AP-1	8	67	975	3	123
AP-2	13	133	1,871	6	79
AP-3	16	174	2,396	8	137
IC-A	35	386	5,545	12	181
IC-B	37	573	9,209	20	360

The reason is that, the objective of our tool was on generating simulators that are correct and are usable for environment-based system level testing by utilizing reasonable level of computing resources. The generated simulator is given to the end-user as an executable archive so we did not focus on optimizing the code for better understanding or cleaner code generation. Therefore, there is room for further optimizations to reduce the number of lines of code, classes, and methods.

The column in Table 5 labeled 'Manually written lines of code' show the lines of code that the developers had to write by hand. These were mostly written for external action code dealing with communication. It is worth noting that, even if the simulator were manually developed, this communication-related code would have to be written in any case and would have been very similar to the code written using our methodology (as we have experienced in one of the industrial case studies). Therefore, the effort required to develop such communication layers is not something specific to our approach, rather it is required whenever environment-based simulations are run.

Overall, the automated generation of the simulator code can be expected to save significant effort to the developers. Though there is a considerable effort involved in developing environment models, given the amount and complexity of the source code generated, it is expected to be less than the effort required for manually developing and maintaining environment simulator code with concurrency and complex synchronization issues. However, to ascertain this claim with confidence, controlled empirical studies in industrial contexts are required.

8.3.3 Effectiveness in test automation

As discussed earlier, to evaluate the effectiveness of the generated simulator to help in test automation, we ran the simulator with various testing strategies on all the five cases, i.e., the three artificial problems and two industrial cases.

Overall, the testing framework was able to trigger system failures corresponding to all the seeded faults for these cases. For IC-B, we ran the testing framework for three different testing strategies: Random Testing, Adaptive Random Testing, and Genetic Algorithms and we were able to find a critical fault in the production code. The detailed results for the experiment conducted on the three artificial problems and this industrial case are presented in [44]. Taken together, the results of these experiments increased our confidence that the generated simulators are effective in detecting faults in the SUT when used in combination with various test automation strategies.

8.3.4 Correctness of the transformation rules

On the stable release of the transformations, we did not find any compilation errors for the test models. Inspecting the code generated revealed that the code was generated according to the extended state pattern. For conformance correctness, the transitions in the models were triggered correctly in the code and all the events that were not defined were ignored (as defined in the methodology). The heuristics reported also matched the desired results, except that the time distance had a jitter of 2 – 4 milliseconds due to the possible noise in timers. To evaluate failure reporting, for the sixteen test cases that we executed, the four that were supposed to trigger the failure in the buggy SUT reported the failure and the rest did not report any failure. This increases our confidence that the generated simulators report failures correctly.

9 Limitations

The focus of the work presented here was only on RTES with soft time deadlines in the order of hundreds of milliseconds, with an accepted jitter of a few milliseconds. However, the modeling methodology proposed in this paper is independent of this limitation and can be used to model systems with stricter deadlines. This limitation is due to the choice of Java as a target language for simulation. The choice of Java was made based on the needs of our industry partners but may obviously not be appropriate in other environments. To use the approach in the presence of stricter deadlines, a possible option is to use a virtual machine supporting Real Time Java (Java RT) (e.g., [70]) on a real-time operating system (e.g., [71]). We have not currently evaluated the practical implications of using Java RT, but the specifications claim that the standard java code is completely portable to a Java RT machine, which provides a precision in the order of nanoseconds.

One of the limitations of the proposed simulator generator is that we still cannot be completely sure about its correctness. As discussed earlier, to test the simulator generator, we wrote a number of test cases by hand. We also ran the generated simulator with the testing engine to test faulty RTES (artificial problems and industrial cases) and were able to trigger failures on test cases revealing seeded faults without triggering failures that were unwarranted. This increased our confidence in the correctness of the transformation rules. The evaluation of such transformations is still an open research question [78] and no suitable tool for testing model to text transformation is available yet.

Based on the experiments that we ran [44] to test different types of RTES (artificial problems and industrial cases), and as discussed above, the generated simulator seems effective in supporting test automation for fault detection. Because this is very time consuming, we however did so only on five RTES. One important question is whether our simulation rules are complete enough to simulate the environment of *any* RTES. To address this possible limitation, the industrial cases and artificial problems that we selected were from diverse domains. One case was of an automated bottle recycling system and the other was a marine seismic acquisition system. Both the case studies involve control oriented, embedded software, which we believe are the most suitable target applications of our methodology. Two of the artificial problems that we developed depicted the common scenarios of RTES interactions with sensors in the environment. The third artificial problem was selected from the domain of train control systems. The diversity of the domains of these RTES, which environment was simulated, increased our confidence in the completeness of the transformation rules and the simulation framework for the RTES developed using our modeling methodology.

We evaluated correctness and completeness of the transformation rules by generating simulators for several test models, three artificial problems, and two industrial cases. Though it cannot be guaranteed that the implemented transformation rules are complete and correct, note that this does not affect the general validity and applicability of the simulation generation approach based on environment modeling using extensions of UML. Such rules will be refined and augmented over time.

10 Conclusion

Black-box system testing of real-time embedded systems (RTES) on their development platforms is required to verify the correctness of these systems without involving the deployed hardware and other physical components of their environments. This approach typically involves simulations of the behavior of environment components in a way that is transparent to the RTES. Such a strategy allows early and fully automated system testing, even when the hardware is not yet available. It is also helpful in situations where testing RTES for critical failures in their actual environments is either not feasible, too costly, or might have catastrophic consequences.

This paper reported on a model-driven automated approach for such black-box system testing strategy based on environment simulation. We purposefully took a practical angle and our approach does not require software engineers to use additional, specific notations for simulation and testing purposes, but only involve slight extensions of existing software modeling standards and a specific modeling methodology. This paper focuses on environment modeling and rules for simulator generation to enable automated black-box system testing and only briefly discusses the test generation strategies, which are reported elsewhere.

As mentioned above, 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. Our modeling methodology entails the use of constructs (e.g., non-determinism, error states, and failure states), which are essential to enable fully automated system testing (i.e., choice, execution and evaluation of the test cases). We modeled the environment of three artificial problems and two industrial RTES in order to investigate whether our methodology and the notation subsets selected were sufficient to fully address the need for automated software testing. Our experience showed that this was the case.

Based on a careful analysis of the literature, we concluded that none of the existing code generation approaches in the literature supports the constructs required to support the testing of RTES through environment simulation. We implemented the code generation rules for the simulator using model-to-text transformations with MOFScript, thus producing a set of Java classes. Our empirical evaluation based on our five case studies shows that the developed rules are sufficient and that they are correct as far as fault detection is concerned. The automated simulator generation is expected to save a significant amount of effort, although controlled empirical studies in industrial contexts will be necessary to support such a claim with increased confidence. By using our environment models and the generated simulators, it was possible to automatically find new, critical faults in one of the industrial case studies using fully automated, large scale random and search-based testing.

Acknowledgments The work presented in this paper was supported by Norwegian Research Council and was produced as part of the ITEA 2 VERDE project. Lionel Briand was partly supported by a PEARL grant from the National Research Fund, Luxembourg (FNR/P10/03 - Validation and Verification Laboratory). We are thankful to Christine Husa, Tor Sjøwall, John Roger Johansen, Erling Marhussen, Dag Kristensen, and Anders Emil Olsen, all from Tomra and Bjørn Nordmoen, Petter Eide, Tore Andre Nilsen, and Sofia Wegger from WesternGeco for their technical support.

Appendix

The appendix provides a description of auto-generated attributes for instances of Context meta-class in Table 6 and a description of auto-generated methods for instances of `ContextState` meta-class of the extended state pattern in Table 7.

Table 6 Auto generated attributes in the context class

Attribute name	Description
`instanceId`: int	Refers to a unique id for every instance in the simulation
action: ? extends ExternalCode	The reference contains an object of action class associated with the context class
states:IState[*]	An array of possible states the environment component can be in
stateContext[*]:Region	The array has object(s) when the class has an orthogonal state machine associated with it.
currentState:IState [0.. 1]	Refers to the state object that the context object is currently in

Table 7 Auto generated methods in the state class

Method name	Description
Constructor	Generally empty unless the state has outgoing time events. In that case `TimeService` object is initialized
`evaluateChangeEvents`	Returns the condition corresponding to any of the change triggers of the outgoing transition from the state that is satisfied
`executeChangeEvents`	The condition that is satisfied is compared and actions corresponding to the change event that the condition relates to are executed.
`executeCompletionEvent`	If the only outgoing transition from the state is without a trigger, then that transition is taken (in this case `changeState` method is executed)
getStateName	Returns the name of the current state
`onStateEntry`	Timers corresponding to time events are initialized. For non-deterministic time events, a value for the timer is obtained from test-engine. Timers associated with «TimeProbability»transition are only initialized/reset if this is the first entry into the state after instance creation or the transition has been taken. Entry activity is executed, do activity is executed in a parallel thread, and completion event is triggered.
`onStateExit`	Exit activity of the state is executed
`stopExecution`	Stops the current thread
`afterT`<i> methods	An `afterT`<i> method is generated for every time event and is called by the timeout method if the corresponding time event is triggered. <i> is the automatically assigned id of the time event.
Signal methods	A signal method is generated for every signal that is defined to be accepted for the state. Action code corresponding to the transition is placed in this method along with a call to `changeState` method
`timeout()`	Calls the `afterT`<i> method whose time event has been triggered

References

1. Iqbal, M.Z., Arcuri, A., Briand, L.: Environment modeling with UML/MARTE to support Black-Box system testing for real-time embedded systems: methodology and industrial case studies. In: Model Driven Engineering Languages and Systems, pp. 286–300. Springer, Berlin (2010)
2. Artemis: (2011) Artemis Joint Undertaking—The public private partnership for R & D Embedded Systems (June 13, 2011). http://artemis-ju.eu/embedded_systems
3. Broekman, B.M., Notenboom, E.: Testing Embedded Software. Addison-Wesley, Boston (2003)
4. OMG: Unified Modeling Language Superstructure, Version 2.3 (2010). http://www.omg.org/spec/UML/2.3/

5. OMG: Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0 (2009). http://www.omg.org/spec/MARTE/1.0/
6. OMG: Object Constraint Language Specification, Version 2.2. Object Management Group Inc. (2010). http://www.omg.org/spec/OCL/2.2/
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software (1995)
8. Cheddar: (2011) http://beru.univ-brest.fr/singhoff/cheddar/
9. Wainer, G.A.: Discrete-Event Modeling and Simulation: A Practitioner's Approach: CRC, Boca Raton (2009)
10. Fritzson, P., Engelson, V.: Modelica–a unified object-oriented language for system modeling and simulation. In: ECOOP'98–Object-Oriented Programming, p. 67. Springer, Berlin (1998)
11. Schamai, W., Fritzson, P., Paredis, C., Pop, A.: Towards unified system modeling and simulation with ModelicaML: modeling of executable behavior using graphical notations. Presented at the 7th International Modelica Conference. Como, Italy (2009)
12. Mooney, J., Sarjoughian, H.: A framework for executable UML models. Presented at the Proceedings of the 2009 Spring Simulation Multiconference. San Diego, California (2009)
13. Kruse, P.M., Wegener, J., Wappler, S.: A highly configurable test system for evolutionary black-box testing of embedded systems. Presented at the Proceedings of the 11th Annual conference on Genetic and evolutionary computation. Montreal, Canada (2009)
14. Lindlar, F., Windisch, A., Wegener, J.: Integrating Model-Based Testing with Evolutionary Functional Testing. Presented at the Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (2010)
15. Lindlar, F., Windisch, A.: A search-based approach to functional hardware-in-the-loop testing. Presented at the Proceedings of the 2nd International Symposium on Search Based, Software Engineering (2010)
16. Short, M., Pont, M.J.: Assessment of high-integrity embedded automotive control systems using hardware in the loop simulation. J. Syst. Softw. **81**, 1163–1183 (2008)
17. Francis, G., Burgos, R., Rodriguez, P., Wang, F., Boroyevich, D., Liu, R. Monti, A.: Virtual Prototyping of Universal Control Architecture Systems by means of Processor in the Loop Technology. Presented at the Twenty Second Annual IEEE Applied Power Electronics Conference, APEC 2007 (2007)
18. Simulink. http://www.mathworks.se/products/simulink/
19. Mason. http://cs.gmu.edu/eclab/projects/mason/
20. SimJava. http://www.dcs.ed.ac.uk/home/hase/simjava/
21. Kishi, T., Noda, N.: Aspect-oriented context modeling for embedded systems. Aspect-Oriented Requirements Engineering and Architecture Design, Presented at the Workshop on Early Aspects (2004)
22. Karsai, G., Neema, S., Sharp, D.: Model-driven architecture for embedded software: a synopsis and an example. Sci. Comput. Progr. **73**, 26–38 (2008)
23. 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. Presented at the IASTED International Conference Proceedings (2006)
24. Kreiner, C., Steger, C., Weiss, R.: Improvement of control software for automatic logistic systems using executable environment models. Presented at the EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO, 1998
25. Burmeister, C.: Real-time environment modeling. In: IEEE (ed.) IEEE Workshop on Real-Time Applications, pp. 142–146. New York (1993)
26. 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 International Workshop on Models in Software Engineering, Leipzig, pp. 57–62 (2008)
27. Pettit IV, R.G., Street, J.A.: Lessons learned applying UML in the design of mission critical software. In: UML: satellite activities. Lecture Notes in Computer Science. Springer, Berlin, pp. 129–137 (2004)
28. Axelsson, J.: Unified modeling of real-time control systems and their physical environments using UML. Presented at the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01) (2001)
29. Gomaa, H.: Designing Concurrent. Distributed and Real-Time Applications with UML. Addison-Wesley Educational Publishers Inc, Boston (2000)
30. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. Elsevier, Amsterdam (2008)
31. Auguston, M., Michael, J.B., Shing, M.: Environment behavior models for automation of testing and assessment of system safety. Inf. Softw. Technol. **48**, 971–980 (2006)
32. 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 (2008)
33. Adjir, N., Saqui-Sannes, P., Rahmouni, K.M.: Testing real-time systems using TINA. In: Testing of software and communication systems. Lecture Notes in Computer Science. Springer, Berlin (2009)
34. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using Uppaal. In: Formal Approaches to Software Testing. Lecture Notes in Computer Science, Springer, Berlin (2005)
35. Hessel, A., Larsen, K., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using uppaal. In: Formal Methods and Testing, pp. 77–117 (2008)
36. Larsen, K.G., Mikucionis, M., Nielsen, B., Skou, A.: Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In: Proceedings of the 5th ACM International Conference on Embedded Software, pp. 299–306 (2005)
37. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods Syst. Design **34**, 238–304 (2009)
38. Du Bousquet, L., Ouabdesselam, F., Richier, J.L., Zuanon, N.: Lutess: a specification-driven testing environment for synchronous software. Presented at the ICSE '99: Proceedings of the 21st International Conference on Software Engineering, Los Angeles (1999)
39. Peleska, J., Lapschies, F., Vorobev, E., Loeding, H., Smuda, P., Schmid, H., Zahlten, C.: A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In: Testing Software and Systems. Springer, Berlin, pp. 146–161 (2011)
40. Peleska, J., Vorobev, E., Lapschies, F., Zahlten, C.: Automated Model-Based Testing with RT-Tester, University of Bremen (2011)
41. David, A., Larsen, K.G., Li, S., Nielsen, B.: Timed testing under partial observability. In: International Conference on Software Testing Verification and Validation, ICST'09, pp. 61–70 (2009)
42. MaTeLo Tool. http://www.all4tec.net/index.php/All4tec/matelo-product.html
43. Iqbal, M.Z., Arcuri, A., Briand, L.: Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In: International Symposium on Software Testing and Analysis (ISSTA) (2012)
44. Arcuri, A., Iqbal, M., Briand, L.: Black-Box system testing of real-time embedded systems using random and search-based testing. In: Testing Software and Systems, pp. 95–110. Springer, Berlin (2010)
45. Iqbal, M.Z., Arcuri, A., Briand, L.: Combining search-based and adaptive random testing strategies for environment model-based

testing of real-time embedded systems. In: Symposium on Search-based, Software Engineering (2012)
46. Pilitowski, R., Dereziñska, A.: Code generation and execution framework for UML 2.0 classes and state machines. In: Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, pp. 421–427. Springer, Netherlands (2007)
47. Chauvel, F., Jézéquel, J.-M.: Code generation from UML models with semantic variation points. In: Model Driven Engineering Languages and Systems, pp. 54–68. Springer, Berlin (2005)
48. SmartState: SmartState–UML statemachine code generation tool (2011). http://www.smartstatestudio.com/
49. IBM: IBM Rational Rhapsody (2011). http://www.ibm.com/software/awdtools/rhapsody/
50. Samek, M.: Practical UML statecharts in C/C++: event-driven programming for embedded systems: Newnes (2009)
51. Ferreira, L., Rubira, C.: The reflective state pattern. Presented at the Proceedings of the Pattern Languages of Program Design, Monticello, IL, USA (1998)
52. Chin, B., Millstein, T.: An extensible state machine pattern for interactive applications. In: ECOOP–Object-Oriented Programming, pp. 566–591. Springer, Berlin (2008)
53. Holt, N., Anda, B., Asskildt, K., Briand, L., Endresen, J., Frøystein, S.: Experiences with precise state modeling in an industrial safety critical system. Presented at the Critical Systems Development Using Modeling Lanuguages, CSDUML'06 (2006)
54. Palfinger, G.: State Action Mapper. Presented at the 4th Pattern Languages of Programming. PLoP), USA (1997)
55. Niaz, I.A., Tanaka, J.: An object-oriented approach to generate Java code from UML Statecharts. Int. J. Comput. Inf. Sci. **6**, 83–98 (2005)
56. Quadri, I.R., Meftali, S., Dekeyser, J.L.: Designing dynamically reconfigurable SoCs: From UML MARTE models to automatic code generation. In: Conference on design and architectures for signal and image processing (DASIP), pp. 68–75 (2010)
57. Rodrigues, W., Guyomarc'h, F., Dekeyser, J.L.: An MDE approach for automatic code generation from UML/MARTE to OpenCL. Comput. Sci. Eng. (2012)
58. Piel É., Atitallah R.B., Marquet P., Meftali S., Niar S., Etien A., Dekeyser J.L., Boulet P., Europe I.L.N.: Gaspard2: from MARTE to SystemC simulation. In: Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML profile DATE, vol. 8, p. 65 (2008)
59. Peñil, P., Medina, J., Posadas, H., Villar, E.: Generating heterogeneous executable specifications in SystemC from UML/MARTE models. Innov. Syst. Softw. Eng. **6**, 65–71 (2010)
60. Vidal, J., De Lamotte, F., Gogniat, G., Soulard, P., Diguet, J. P.: A co-design approach for embedded system modeling and code generation with UML and MARTE. In: Design, Automation and Test in Europe Conference and Exhibition (DATE '09), pp. 226–231 (2009)
61. Mraidha, C., Tanguy, Y., Jouvray, C., Terrier, F., Gérard, S.: An execution framework for MARTE-based models. In: 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 222–227 (2008)
62. Yu, H., Talpin, J.P., Besnard, L., Gautier, T., Marchand, H., Le Guernic, P.: Polychronous controller synthesis from MARTE CCSL timing specifications. In: 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 21–30 (2011)
63. Selic, B.: A systematic approach to domain-specific language design using UML. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time, Distributed Computing, 2007. ISORC'07, pp. 2–9 (2007)
64. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall PTR, Upper Saddle River (2001)
65. OMG: Concrete Syntax for UML Action Language (Action Language for Foundational UML–ALF), Version 1.0–Beta 1 (2010). http://www.omg.org/spec/ALF/
66. Simulink Coder. http://www.mathworks.se/products/simulink-coder/index.html
67. Musa, J.D.: The operational profile in software reliability engineering: an overview. Presented at the Third International Symposium on Software, Reliability Engineering (1992)
68. Oldevik, J.: MOFScript user guide. Version 0.6 (MOFScript v 1.1.11) (2006)
69. Bruegge, B., Dutoit, A.: Object-Oriented Software Engineering Using UML, Patterns, and Java. Prentice Hall, Upper Saddle River (2009)
70. Sun Java Real-Time System. http://java.sun.com/javase/technologies/realtime/index.jsp. Accessed on 09/02/2012
71. SUSE Linux Enterprise Real Time Extension. http://www.novell.com/products/realtime/. Accessed on 09/02/2012
72. McMinn, P.: Search based software test data generation: a survey. Softw. Test. Verif. Reliab. **14**, 105–156 (2004)
73. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.: A Search-based OCL constraint solver for model-based test data generation. Presented at the 11th International Conference on Quality Software (2011)
74. Runeson, P., Rainer, A., Höst, M., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. Wiley, New York (2012)
75. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131–164 (2009)
76. Zheng, M., Alagar, V., Ormandjieva, O.: Automated generation of test suites from formal specifications of real-time reactive systems. J. Syst. Softw. **81**, 286–304 (2008)
77. Iqbal, M., Ali, S., Yue, T., Briand, L.: Experiences of applying UML/MARTE on three industrial projects. In: Model driven engineering languages and systems, pp. 642–658 (2012)
78. Fiorentini, C., Momigliano, A., Ornaghi, M., Poernomo, I.: A constructive approach to testing model transformations. In: Theory and Practice of Model Transformations, pp. 77–92 (2010)
79. Andrews, J., Briand, L., Labiche, Y., Namin, A.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Softw. Eng. **32**, 608–624 (2006)

Author Biographies

Muhammad Zohaib Iqbal is currently an Assistant Professor at the Department of Computer Science, National University of Computer and Emerging Sciences (Fast-NU), Islamabad, Pakistan. He received his PhD degree in software engineering from University of Oslo, Norway in 2012. Before joining Fast-NU, he was a research fellow at Simula Research Laboratory, Norway. He has also worked as a lecturer at the Department of Computer Science, International Islamic University, Islamabad, Pakistan and Department of Computer Science, Mohammad Ali Jinnah University, Pakistan. His research interests include model-driven engineering, software testing, and empirical software engineering. He has been involved in research projects in these areas since 2004.

Andrea Arcuri received a B.Sc. and a M.Sc. degree in computer science from the University of Pisa, Italy, in 2004 and 2006, respectively. He received a PhD in computer science from the University of Birmingham, England, in 2009. He works as a Senior Software Engineer in the gas and oil industry, while having a small part time position as Adjunct Research Scientist at Simula Research Laboratory, Norway. His research interests include search-based software testing and randomized algorithms.

Lionel Briand is professor and FNR PEARL chair in software verification and validation at the SnT centre for Security, Reliability, and Trust, University of Luxembourg. Lionel started his career as a software engineer in France (CS Communications & Systems) and has conducted applied research in collaboration with industry for more than 20 years. Until moving to Luxembourg in January 2012, he founded and was heading the Certus center for software verification and validation at Simula Research Laboratory, where he was leading applied research projects in collaboration with industrial partners.Before that, he was on the faculty of the department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he was full professor and held the Canada Research Chair (Tier I) in Software Quality Engineering. He has also been the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany, and worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland, USA. Lionel has been on the program, steering, or organization committees of many international, IEEE and ACM conferences. He is the coeditor-in-chief of Empirical Software Engineering (Springer) and is a member of the editorial boards of Systems and Software Modeling (Springer) and Software Testing, Verification, and Reliability (Wiley). He was on the board of IEEE Transactions on Software Engineering from 2000 to 2004. Lionel was elevated to the grade of IEEE Fellow for his work on the testing of object-oriented systems. He was recently granted the IEEE Computer Society Harlan Mills award for his work on model-based verification and testing. His research interests include: model-driven development, testing and verification, search-based software engineering, and empirical software engineering.