Model-in-the-loop Testing of a Railway Interlocking System

Fabio Scippacercola[1], Roberto Pietrantuono[1], Stefano Russo[1], and András Zentai[2]

[1] DIETI, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy, {fabio.scippacercola, roberto.pietrantuono, sterusso}@unina.it

[2] Prolan Process Control Co., Szentendreiút 1-3, H-2011 Budakalász, Hungary, zentai.andras@prolan.hu

Abstract. Model-driven techniques offer new solutions to support development and verification and validation (V&V) activities of software-intensive systems. As they can reduce costs, and ease the certification process as well, they are attractive also in safety-critical domains. We present an approach for Model-in-the-loop testing within an OMG-based model-driven process, aimed at supporting system V&V activities. The approach is based on the definition of a model of the system environment, named Computation Independent Test (CIT) model. The CIT enables various forms of system test, allowing early detection of design faults. We show the benefits of the approach with reference to a pilot project that is part of a railway interlocking system. The system, required to be CENELEC SIL-4 compliant, has been provided by the Hungarian company Prolan Co. in the context of an industrial-academic partnership.

Keywords: Model-Driven Engineering, Safety-critical systems, Model-Driven Testing.

1 Introduction

The development of systems in safety-critical domains is complex and expensive. Normative certification standards in such domains require many verification, validation and certification activities to produce evidence that a high level of confidence has been achieved, that take most of resources of the entire development process. A key role in system V&V activities is played by the environment, whose correct specification is essential for the generation of proper test sequences.

In critical domains such as railway, automotive, and avionics, Model-Driven Engineering (MDE) is increasingly adopted with the aim of reducing time and costs. MDE refers to engineering processes in which models are key artifacts of the work [1]. Models are defined in (semi-)formal languages, and other artifacts are derived through defined transformations: model-to-model transformations (M2M), or model-to-text transformation (M2T) from models to textual documents, source code or testing artifacts (test cases and test scripts).

The current MDE practices emphasize a strong support to system design modeling and automatic generation of artifacts at different level of abstraction: while this is undoubtedly of primary importance, the MDE support to environment modeling, and more generally to system V&V activities, is still underestimated. If we want to fully exploit MDE potentials for critical contexts, the role of correct environment modeling for V&V tasks cannot be disregarded.

In the context of the European project CECRIS[3] – investigating new methods for certification of critical systems – we have defined a model-driven process incorporating concepts of the OMG MDA[4] [2] and MDT[5] [3] techniques, meant to support V&V of safety-critical systems [4]. The process has been experimented in conjunction with Prolan Co., a Hungarian company active in the domain of railway systems, on a pilot project for a new interlocking system.

In this paper, we focus on the topmost part of the process, and we detail how our methodology exploits model-driven techniques to define an environmental model enabling: *i)* an environment-aware specification supporting the definition of a test plan; *ii)* the detection of design flaws at early stages of the development lifecycle; *iii)* the set-up of a framework to run model-, software- and hardware-in-the-loop tests. We experiment the approach on a Prolan's system, named *Prolan Monitor* (PM), which is part of a railway interlocking system required to be certified as SIL-4 (the highest safety level). Results highlight the potential advantages that can be gained when the environment is taken into account from the beginning through modeling support.

In the remainder, we briefly survey the state-of-the-art about model-driven V&V and certification for critical systems (Section 2); then, our MD V-model process is described (Section 3). Section 4 presents the experience regarding environment modeling for the railway case study. Section 5 discusses the results.

2 Related work

There are several experiences on the application of MDE in industrial projects; a systematic review of the available literature is in [5]. Although MDE is generally perceived as positive – in terms of productivity, quality improvement, or automation support – not all claimed benefits are fully supported by consolidated empirical evidence. MDE is used mainly for code generation, while few meaningful studies document its use for simulation, test generation, validation and early detection of design flaws. In particular, not many success stories are documented on the adoption of MDA/MDT techniques based on standard (non-proprietary) modeling notations to support evidence of their maturity.

[3] CErtification of CRItical Systems, www.cecris-project.eu.

[4] MDA (Model-Driven Architecture) is the specific Model-Driven Development (MDD) approach proposed by the OMG standardization organization.

[5] MDT (Model-Driven Testing) refers to MDE V&V activities. It is not an OMG standard, but it is based on a UML standard profile, the UML Testing Profile (UTP), which adapts UML as a test specification language. In MDT, test infrastructure, test cases, and test scripts are derived from UTP models through transformations.

MDE has found application for safety-critical systems particularly in the railway domain. Authors in [6] report a successful application of Simulink/Stateflow models to the design of an on-board equipment of a Train Protection System. With particular restrictions on models, and using a model-based testing approach called Translation Validation, the authors were able to certify the system according to the CENELEC standards. Another interesting application of MDD for the generation of proper configurations of computer based interlocking systems is presented in [7]: secondary artifacts are automatically generated by model transformations to support CENELEC certification.

Indeed, important MD advantages for certification lie in the support for requirements traceability and for formal V&V. An MDE technique for the assessment of railway control systems is proposed in [8]; it is based on specialized UML profiles enabling translations to specific formalisms, for automated test cases generation and model checking. In the airborne domain, a similar solution for integrating model checking with various synchronous dataflow languages adopted by commercial MDD tools (e.g., MATLAB Simulink or SCADE) is discussed in [9]. SCADE is a DO-178B qualified model-based environment for mission- and safety-critical embedded applications [10]. A study on the use of the SCADE suite for the verification of railway control systems can be found in [11], while a success story of its application is reported in [12]. Techniques to enhance traceability and documentation capabilities of MDE to ease safety inspections and certification processes are proposed in [13, 14].

MDE sees a growing adoption for embedded systems development [15], where it enables additional forms of V&V to achieve the required behavior and quality. *Model-in-the-loop* is an iterative testing activity involving the models of the system and of its environment. When the system model is replaced with a concrete software implementation, the testing is named *Software-in-the-loop*, aiming at identifying faults due the translation of the model into code. Finally, *Processor-in-the-loop* and *Hardware-in-the-loop* testing aim at assessing the system in realistic environments, i.e., on the target processor or on the production hardware. Such forms of testing are spread in the automotive industry, where MATLAB Simulink is generally adopted [16, 17].

3 The overall approach

MDE is often introduced in industrial contexts adapting traditional domain-specific processes. This Section introduces the proposed model-driven process originated from a conventional V-model process. The activities are grouped in those concerning development (left side) and V&V (right side), where we exploit, respectively, MDA and MDT.

On the left side of the V-model we follow the MDA approach: at each step we focus on one of the three viewpoints of the system (*Computation Independent Viewpoint*, *Platform Independent Viewpoint* and *Platform Specific Viewpoint*), used to define the *Computation Independent Model* (CIM), the *Platform Independent Model* (PIM), and the *Platform Specific Model* (PSM). The same

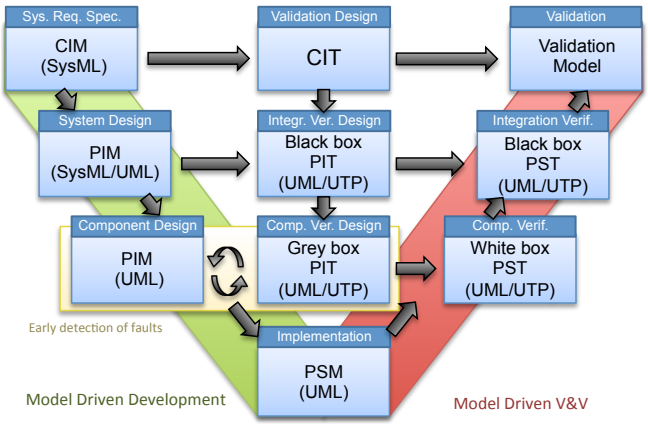

Fig. 1. The proposed model-driven V-Model. Boxes show the activities (e.g., *System Requirements Specification*), the models produced (e.g., CIM), and the formalisms used (e.g., SysML). The arrows indicate that the model at the arrow's origin is exploited by the activity at the arrow's destination (e.g., the *Validation Design* exploits the CIM to derive the CIT model). Note that, in a model-in-the-loop testing, the *Component Design* also exploits the test models to allow early fault detection.

abstractions are adopted in the activities of the center and right side of the 'V', but focusing on defining additional models that support the V&V exploiting the different views of the system.

In *System Requirements Specification*, we define a CIM for modeling the environment and the system requirements. The CIM is defined in SysML; this language is particularly suited for this phase, as it offers requirements diagrams and allows modeling both hardware and software components.

System Design refines the CIM into a PIM, by defining the high-level system architecture and its components. Requirements are assigned to the components. In this phase the PIM describes for each component the allocated requirements, the interfaces, and the expected interactions at components' interfaces. UML *Protocol State Machines* are suited at this stage, because they describe I/O relations without reference to the internal design of components.

Component Design completes the PIM with the internal design of the elements. Considering the software, this model is expressed in UML and should be specific enough to be subject to simulation. Since the *Component Design* focuses on describing the dynamic behavior of the elements, it can exploit UML *Behavioral State Machines*.

In the *Implementation* phase, the PIM is refined into one or more PSMs bound to the target platforms. For instance, a PSM adapts the generic types of the variables with the actual ones provided by a programming language, and binds data and function calls to the interfaces of the middleware and OS that have been chosen for the instantiation. The PSM can be translated into code to provide a partial or a total implementation of the system.

In the *Validation Design* we move to environment modeling. At this stage we exploit the CIM to define a novel model named *Computation Independent Test* model (CIT). The CIT is unaware of computation details, and focuses on the interactions with the actors. It allows expressing such interactions as properties and conditions related to the environment, e.g., "the station (interacting with the interlocking system under development) cannot invert the railway direction if there is a train occupying the block". The CIT allows creating a simulated environment in which engineers are supported in the validation activity of the systems' design models. The CIT is a basic block of the proposed Model-in-the-loop testing approach, as further discussed in the next section.

Integration Verification Design defines a model of the expected behavior of the system's components, independent from their inner design. We refer to it as *Black Box Platform Independent Test* model (BB-PIT). This model provides static and dynamic views of the system's components, and it is mainly used to support functional testing in the unit/integration/system verification. The static description supports the generation of the test infrastructure, including stubs and drivers for unit and integration testing. The dynamic description is composed by: *(i)* behavioral models, such as UML Behavioral State Machines, defined starting from requirements allocation to components in the PIM model; *(ii)* test cases, which are specified by Sequence, Activity and State Machines diagrams using the UTP profile. Behavioral models are useful indeed for automatic generation of test cases. A BB-PIT models the behavior of one component with more state machines, each focusing on a different subset of functionalities, with the possibility of composing test suites by grouping tests derived by several state machines. In addition, a BB-PIT supports the detection of design faults by comparing the behavior it describes with the one defined in the PIM. In fact, the behavior of a component is modeled differently in the PIM and in the BB-PIT, due to their different purposes: a PIM specifies how to build the system, and represents the specification of an actual implementation must comply with; a BB-PIT describes the expected behavior in a way to verify its correspondence between requirements and implementation (e.g., by using the BB-PIT for test case generation, the description represents the specification test cases must comply with). It is worth noting that, since BB-PIT derives from requirements and is barely influenced by design details, it supports validation too.

Component Verification Design refines the BB-PIT defining a *Grey Box PIT* model (GB-PIT), exploiting the PIM at *Component Design*-level that provides a partial internal view of the system. The GB-PIT enables additional verification techniques that can exploit structural features to assess correctness. Following this flow, engineers first focus on a functional V&V modeling in the *Integration Verification Design*, and then move to functional and structural V&V modeling in this phase. Moreover, since an executable PIM is available in this phase, the PIT allows performing a preliminary verification and validation of the design model, in order to detect defects before implementation. In addition, by exploiting the CIT properties on the PIM, model checking techniques can assess the absence of any undesired condition in operation.

The V&V activities of the right side of V-Model refine the CIT and the PITs considering details of the target platform, of implementation, and of the PSM. With these refinements, the BB-PIT and the GB-PIT become the *Black Box* and the *White Box Platform Specific Test* (respectively, BB-PST and WB-PST). For instance, the BB-PST allows to define test cases based on the knowledge of the target platform arithmetic, while the WB-PST can adopt a coverage criterion based on source code. Finally, testing plans, test cases, and artifacts supporting the V&V are derived by the PSTs through (automatic) transformations.

4 Environment Modeling

4.1 The Computation Independent Test model

We have introduced a new form of environment modeling in the process during the *Validation Design*, through the definition of a Computation Independent Test model. The CIT is an *executable* model of the environment that captures the behavior of the system's context and provides interfaces *complementary* to those of the SUT. Therefore, it can be seen as a *symmetric PIM* and can be linked or put in a feedback loop with the system to enable early V&V.

The CIT is useful to create a simulated environment to reason about the operational aspects of the system in its context: we can validate the system against its expected interactions with the external actors and perform special kinds of system assessment (like performance testing or stress testing) by generating representative operational profiles. Also, whenever model checking techniques are adopted, undesired condition in operation can be detected by analyzing the state space of the SUT combined with the environment.

4.2 Case study

In our previous work, we experimented the model-driven process integrated in the V-model in a pilot project of the *Prolan Block* (PB) railway interlocking system, under development by Prolan. Here we focus on the environment modeling, considering the *Prolan Monitor* (PM), another part of the interlocking system, which must be CENELEC EN50126, EN50128 and EN50129 SIL-4 certified. The PB shares with the PM the same hardware and middleware platform (*Prosigma*), which is the basis of the next generation of the company's products: according to the CENELEC terminology, Prosigma is a *generic product*, the PB and the PM are *generic applications*, and their instances are *specific applications*.

The PM is deployed on railway segments, named *blocks*. Each block is equipped with a legacy interlocking system and the PM, whose goal is to receive binary signals from the interlocking system and to transmit the information to newer devices that adopt different protocols, e.g., as datagrams over an IP network. In particular, the PM monitors one or more *railway objects*: these objects send a bit via couples of electric signals coding the information by valent and antivalent physical signal values. The PM must transmit the logical values to other devices and detect invalid states when couples of electric signals are not consistent.

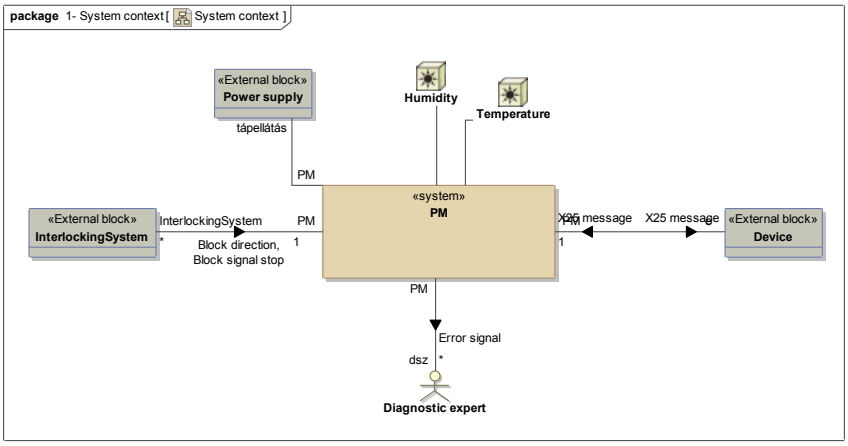

Fig. 2. A SysML Block Definition Diagram of the CIM, showing the PM in its context.

In the System Requirements Specification, we define a CIM using SysML, that focuses on the requirements and on the context of the PM (Fig. 2). The CIM includes: 27 Activity Diagrams, 4 Block Definition Diagrams, 5 Internal Block Diagrams, 1 Package Diagram, 9 Sequence Diagrams, 7 State Machine Diagrams, and 1 Timing Diagram.

The timing diagram in Fig. 3 describes the requirements on the PM's functionality of signal filtering. The PM must sample its input with a period T_{sample}; since the input can suffer from transient states, a filtering solution must be implemented. By the valent and antivalent signals, two *debounced* signals are derived, which filter out the variations of signal that are shorter than $messageFilterTime$; then, the invalid configurations of the debounced couple of signals (i.e., $(0, 0)$ and $(1, 1)$) are masked if they last less than $maxTransientTime$. The railway objects assume invalid state if the signals bounce more than $maxBouncingTime$ or if the transient state lasts more than $maxTransientTime$.

In the System Design phase, the PIM is created, by defining the high level architecture of the system, including components' interfaces and their specification. In this pilot project, we design the PM system as composed by multiple instance of *PMRailwayObject*s, each of those in charge to manage a couple of input binary signals assigned to a physical railway object. In order to be platform-independent, the logic for accessing the hardware resources is masked by the *PMInterface*, required by the *PMSystem*.

In the Component Design phase, the PIM is refined completing the system internal design. The *PMRailwayObject* consists of one *Sampler*, two *PMDebouncer*s and one *PMInputFilter* (Fig. 4). The sampler reads from the input channels and notifies the values to the two *PMDebouncer*s, filtering the valent and antivalent signal. Finally, the *PMDebouncer*s propagate the signals to the *PMInputFilter* that filters out transient states, as specified in Fig. 3.

At this stage, we also define the behavior of the components, by using UML State Machines or Activity Diagrams. The state machine in Fig. 5 models the

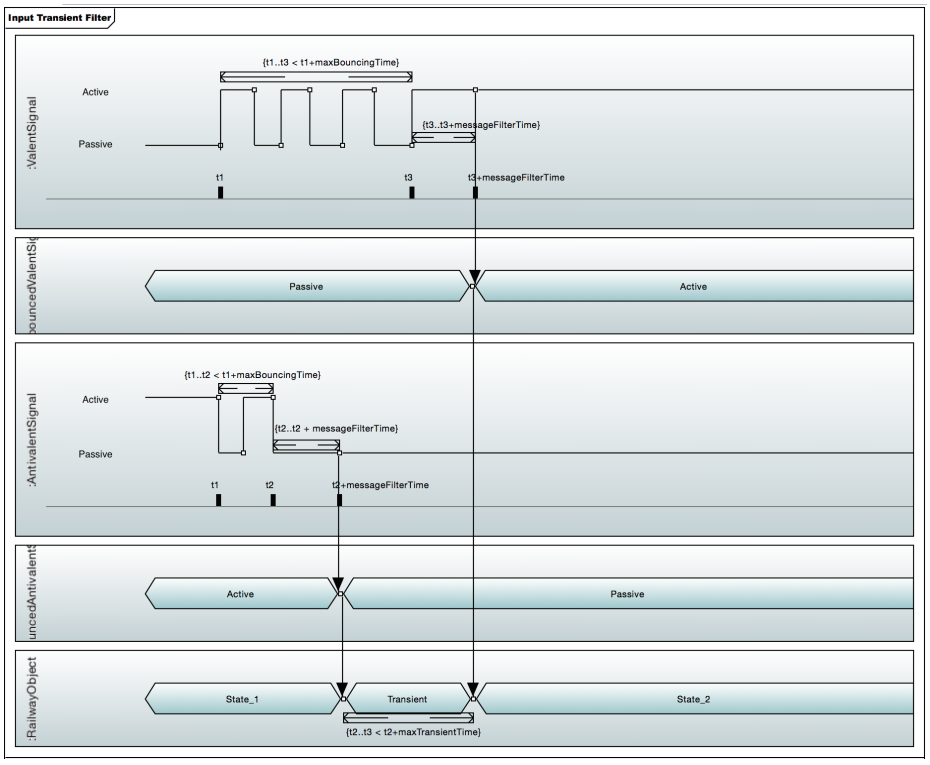

Fig. 3. A UML Timing Diagram included in the CIM, representing the requirements for the functionality of signal debouncing.

behavior of the *PMDebouncer*s as a two-orthogonal composite state: the state machine in the left monitors if the input is stable and sends *inputStabilizedEvent*s to the region in the right. The latter determines if the input is bouncing for a time longer than the maximum allowed.

The PIM is defined using IBM Rhapsody Developer (hereinafter: Rhapsody) [18], following guidelines to let the model be platform-independent.

As the derived model is executable, it can be animated to observe the running program and the system's interactions. This is a feature that we exploited to get an immediate feedback on program behaviour. Moreover, we can easily create prototype of user interface for interacting with the model: by means of the Rhapsody Panel Diagrams, we can set the logical signals and observe the output of the PM (Fig. 9, panel diagram on the left).

In the Implementation phase, we define the PSM. In this project we set several tagged values and tool-dependent parameters to enrich the PIM; then, the tool uses the additional information for translating the model into code. The automatic translation generated around 4.3 thousands of lines of Java code; the source code is readable, understandable and ready to run.

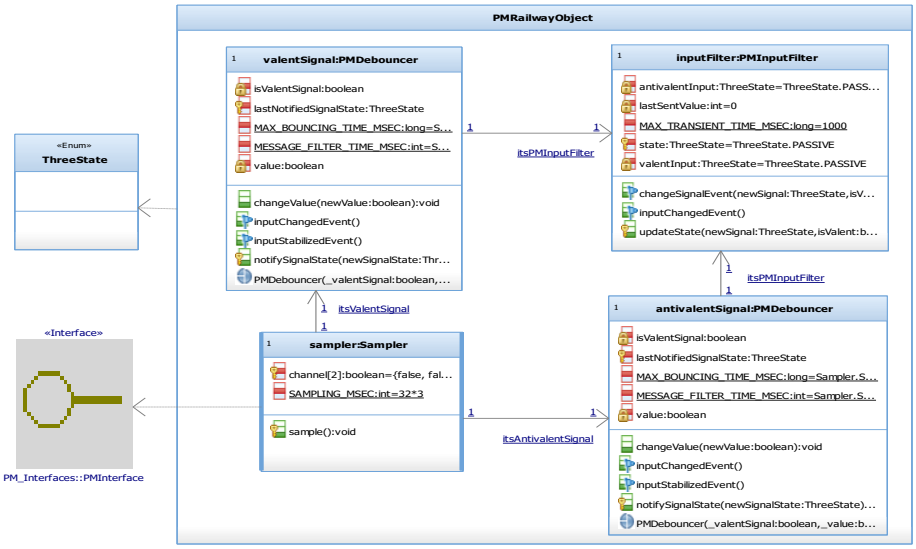

Fig. 4. The UML Internal Structure Diagram of the *PMRailwayObject*, defined in the PIM during Component Design.

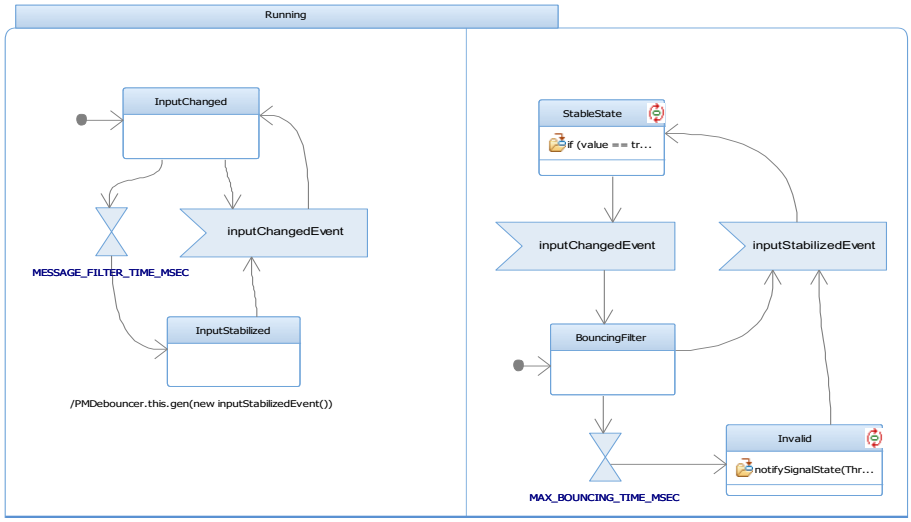

Fig. 5. The UML Behavioral State Machine of the class *PMDebouncer*, defined in the PIM during Component Design.

The Validation Design phase aims at creating an executable model of the environment. For the Prolan Monitor, the architecture of the CIT model is formed by two *CITRailwayObject*s: each *CITRailwayObject* controls the couple of logical signals associated with the binary information that it encapsulates; form the CIT *point of view*, the PM is an actor (Fig. 6).

The *CITRailwayObject*s are modeled as composed by one *SignalGenerator* and one *EventGenerator*: the *EventGenerator* determines the next output to be triggered, as specified by a user-defined operational profile. The *EventGenerator* generates the following events:

NoAction: the output is not altered in the next event generation loop;
ChangeStableState: the railway object switches between valid stable states;
CreateSpike: the output moves to an invalid state and then back to the previous stable state in order to simulate a transient in the electric signals;
Fail: the railway object moves to an invalid state and then fails.

According to the events sent by the *EventGenerator*, the *SignalGenerator* properly sets the couple of output signals and manages the duration of the transients. The behavior of the *EventGenerator* is modeled by an activity diagram, while the behavior of the *SignalGenerator* by a state machine (Fig. 7): the *SignalGenerator* can generate sets of valid or invalid signals; when a *changeOutputEvent* is triggered by the *EventGenerator*, it evolves to the next stable state passing through invalid signals (i.e., $(1,1)$ or $(0,0)$) for a time equals to *transientDuration*. Then, if the *CITRailwayObject* is not failed, the *SignalGenerator* notifies the *EventGenerator* that the next stable state has been reached, and it starts to wait for the next event.

A panel diagram allows to interact with the CIT (Fig. 9, on the right): it offers a couple of knobs to set the period of event generation as well as the duration of transient states, and shows the output generated by the railway object.

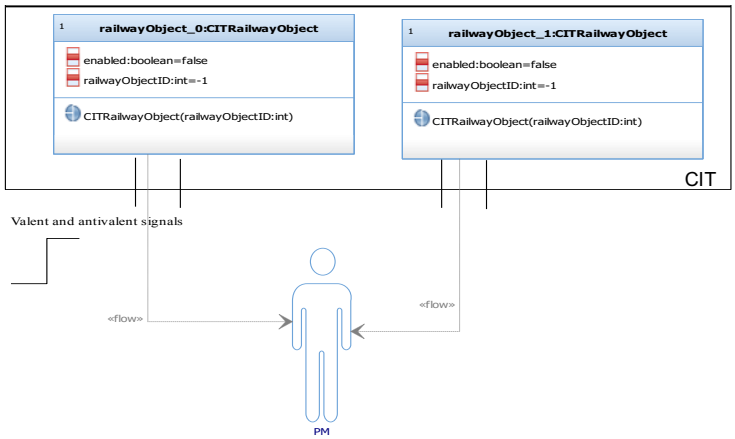

Fig. 6. The architecture of the CIT.

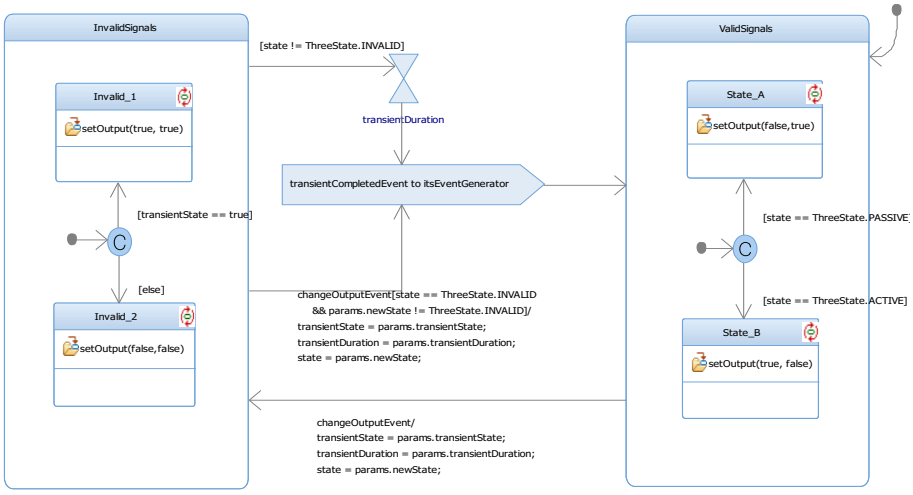

Fig. 7. The UML Behavioral State Machine model of the *SignalGenerator*, defined in the CIT during Validation Design.

4.3 Model-in-the-loop testing

We integrate the CIT and the PIM in order to perform the Model-in-the-loop testing. Since their interfaces are complementary, we link the two models by an adapter that simulates a physical relay, the *VirtualRelay* (Fig. 8): the CIT sends commands to switch the virtual relays, while the PIM reads their status.

Once the CIT and the PIM are linked, we can execute the whole model and examine its evolution by means of the output console and of the panel diagrams of Fig. 9. To use the CIT for Software- and Hardware in-the-loop testing, we just need to change the adapter to forward the events to the actual SUT. Specifically, to run Hardware-in-the-loop tests in our case study, we replace the *VirtualRelay* with a physical relay card connected to Prosigma.

To assess the fulfillment of the requirements for the functionality of signal filtering, we have designed a test plan to assess if the events received by the *External Device* are the expected ones, according to the behavior of the *Interlocking System* and of the input signals (Fig.s 2, 3). We apply category partition testing (CPT) on the CIT's interface (Table 1), deriving six test case obligations: in this type of testing, categories are configurations of the environment (i.e., of

Fig. 8. The software adapter linking the interfaces required by the CIT and the PIM.

the CIT) that lead to the generation of different sequences of effective stimuli to the SUT. The test case specification is summarized in Tables 2 and 3. As test oracle, we implemented a script to analyze the execution traces of the actors and of the PIM in order to detect any undesired behavior.

Table 1. CPT test categories.

Parameter	Categories	Constraints	Test case ID
Input domain	(1.1) Valid values and invalid transients		TC1-4
	(1.2) Valid and invalid values	[ERROR]	TC5
Input frequency	(2.1) Low frequency		TC1-2, TC4-6
	(2.2) High frequency	[SINGLE]	TC3
Duration of transients	(3.1) Undetectable by the SUT	[SINGLE]	TC3, TC6
	(3.2) Detectable by the SUT		TC1-2, TC5
	(3.3) Erroneous for the SUT	[ERROR]	TC4
Signal fluctuations	(4.1) Low probability		TC1, TC3-6
	(4.2) High probability	[SINGLE]	TC2

To execute the tests, the code of the model (in configuration *in-the-loop*) is generated without the instrumentation needed for animating the model in Rhapsody, so as to avoid slowing down the execution. Since the tests TC3 and TC6 require a time granularity of 10 ms, we tuned the *time tick* to 5 ms: this parameter specifies the time resolution to be used to poll the time events of the

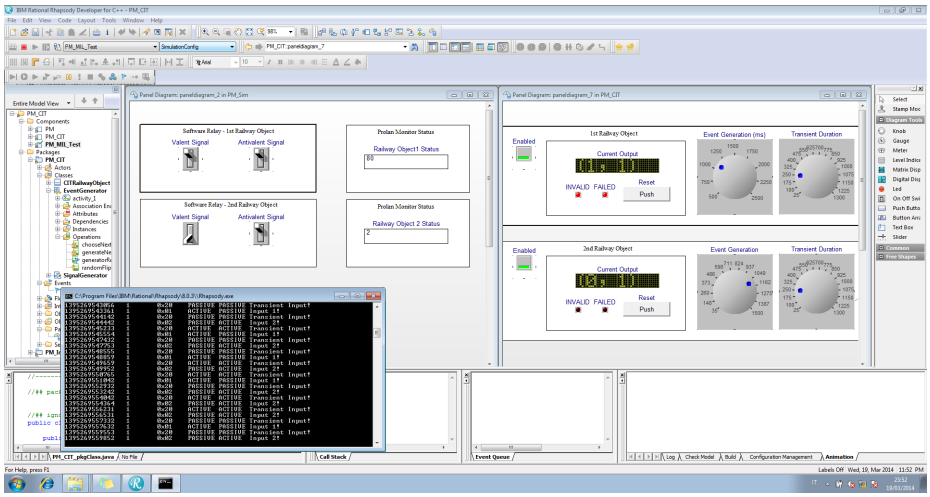

Fig. 9. A screenshot of Rhapsody showing two panel diagrams and an output console: the panel diagram on the left is linked with the PIM, whereas the panel diagram on the right is connected to the CIT. In Model in-the-loop configuration, both are linked through the *VirtualRelay*, and are part of the animation that produces output in the console.

Table 2. Specification of the test cases. All test cases except TC5 send to the SUT valid input values ($(1,0)$ and $(0,1)$) and invalid transient values ($(0,0)$ and $(1,1)$).

TC ID	Categories covered	Event Generation Period	Transient Duration	Probability of Fluctuations
TC1	1.1, 2.1, 3.2, 4.1	2.5 s	100 ms	1%
TC2	1.1, 2.1, 3.2, 4.2	2.5 s	100 ms	40%
TC3	1.1, 2.2, 3.2, 4.1	65 ms	10 ms	1%
TC4	1.1, 2.1, 3.3, 4.1	2.5 s	5,000 ms	1%
TC5	1.2, 2.1, 3.2, 4.1	2.5 s	100 ms	1%
TC6	1.1, 2.1, 3.1, 4.1	2.5 s	10 ms	1%

Table 3. Configuration of the SUT for the experiments.

Parameter	Value
T_{sample}	35 ms
messageFilterTime	$3 * T_{sample} = 105$ ms
maxBouncingTime	$5 * T_{sample} = 175$ ms
maxTransientTime	1,000 ms
Time tick	5 ms
Execution time	300 s

state machines and of the activities. By analyzing the execution during testing, we assure that the hardware was adequate to meet the timing constraints.

Note that, as the events are sent to a running software system, we are actually performing a form of Software in-the-loop testing. However, tests are not executed on the final software code, but on the instantiation of the PIM generated by Rhapsody that we are adopting for animation and testing purposes.

In our experiments, the SUT passed all the tests, behaving correctly. Using this approach, we enabled an early detection of design fault, because we could exercise the design model in its context before a complete implementation was available. Focusing on modeling the environment, we are suggesting a form of separation of concerns to design the test plan, that guides to design test cases considering the conditions of the environment.

5 Discussion and conclusions

In this study, we presented an approach for Model-in-the-loop testing, embedded in a model-driven process, aimed at supporting the activities of V&V. The approach accounts for the relevant role of the environment, especially in safety-critical system V&V, by introducing modeling activities suited for designing and executing additional forms of verification and validation. The approach is experimented on a pilot project that is part of an interlocking system required to be CENELEC EN50126, EN50128 and EN50129 SIL-4 certified.

By exploiting the newly defined *Computation Independent Test* model (CIT), we guided the generation of tests based on the analysis of the environment, suc-

cessfully supporting a Model-in-the-loop verification. Such a form of verification can favor the early detection of faults, while taking advantage of the automation offered by MDE tools. By performing testing on the SUT, we were able to verify the design model and show the correctness of the PIM integrated with a model of its environment.

As further remark, it is worth to emphasize that the CIT can be used to assess multiple systems, where the environment is the same: in other words, we are introducing a form of *environment model reuse*. For instance, in our case study, the CIT can be reused with the *Prolan Block*, because the PB is based on the same hardware platform and shares the requirements for input filtering. Additionally, applying a MDA approach for the development of the CIT, we can easily deploy the CIT under different configurations, so as to support verification tasks of a broader range of SUT.

Finally, when the implementation is available, the CIT can be adapted to perform Software-, Processor- and Hardware-in-the-loop testing, offering a ready-to-use environment to execute performance and stress tests.

As future work, we will mainly focus on evaluating the benefits achievable by the proposed process, as well as further improvement margins, under Software- and Hardware-in-the-loop tests.

Acknowledgment

This research has been supported by the EU FP7 Programme 2007-2013 under REA grant agreement n. 324334 CECRIS (CErtification of CRItical Systems, www.cecris-project.eu) within the IAPP (Industry Academia Partnerships and Pathways) Marie Curie Action of the People Programme.

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. 1st edn. Morgan & Claypool Publishers (2012)
2. OMG: MDA Guide. http://www.omg.org/cgi-bin/doc?omg/03-06-01 (2003) Version 1.0.1.
3. Baker, P., Dai, Z.R., Grabowski, J., Haugen, Ø., Schieferdecker, I., Williams, C.: Model-Driven Testing: Using the UML Testing Profile. 1st edn. Springer-Verlag Berlin Heidelberg (2008)
4. Scippacercola, F., Pietrantuono, R., Russo, S., Zentai, A.: Model-Driven Engineering of a Railway Interlocking System. In: Proc. of MODELSWARD 2015, 3rd International Conference on Model-Driven Engineering and Software Development, SCITEPRESS (2015) 509–519
5. Mohagheghi, P., Dehlen, V.: Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Proc. of the 4th European Conference on Model Driven Architecture: Foundations and Applications. ECMDA-FA '08, Springer-Verlag (2008) 432–443
6. Ferrari, A., Fantechi, A., Magnani, G., Grasso, D., Tempestini, M.: The Metrô Rio Case Study. Science of Computer Programming **78**(7) (2013) 828–842

7. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.J., Haugen, O.: The Future of Train Signaling. In: Proc. of MoDELS '08, 11th International Conference on Model Driven Engineering Languages and Systems, Springer-Verlag (2008) 128–142
8. Marrone, S., Flammini, F., Mazzocca, N., Nardone, R., Vittorini, V.: Towards Model-Driven V&V assessment of railway control systems. International Journal on Software Tools for Technology Transfer **16**(6) (2014) 669–683
9. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software Model Checking Takes off. Communications of the ACM **53**(2) (2010) 58–64
10. Esterel Technologies: SCADE Suite Product Description. http://www.estereltechnolgies.com (2014)
11. Lawrence, A., Seisenberger, M.: Verification of railway interlockings in SCADE. MRes Thesis, Swansea University (2011)
12. Invensys Rail: Invensys Rail Discovers Agile Development Process with SCADE Suite. http://www.esterel-technologies.com/success-stories/invensys-rail/ (2014)
13. Nejati, S., Sabetzadeh, M., Falessi, D., Briand, L., Coq, T.: A SysML-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. Information and Software Technology **54**(6) (2012) 569–590
14. Panesar-Walawege, R., Sabetzadeh, M., Briand, L.: A Model-Driven Engineering Approach to Support the Verification of Compliance to Safety Standards. In: Proc. of ISSRE 2011, IEEE 22nd International Symposium on Software Reliability Engineering. (2011) 30–39
15. Shokry, H., Hinchey, M.: Model-based verification of embedded software. Computer **42**(4) (2009) 53–59
16. Amalfitano, D., Fasolino, A.R., Scala, S., Tramontana, P.: Towards automatic model-in-the-loop testing of electronic vehicle information centers. In: Proc. of WISE '14, International Workshop on long-term Industrial Collaboration on Software Engineering, ACM (2014) 9–12
17. Matinnejad, R., Nejati, S., Briand, L., Bruckmann, T., Poull, C.: Automated Model-in-the-Loop Testing of Continuous Controllers Using Search. In: Proc. of SSBSE 2013, 5th International Symposium on Search Based Software Engineering. Volume 8084 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 141–157
18. IBM Corp.: Rational Rhapsody Developer. http://www-03.ibm.com/software/products/it/ratirhap (2014)