Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

Development of an automated MBT toolchain from UML/SysML models

Jonathan Lasalle · Fabien Peureux · Frédéric Fondement

Received: date / Accepted: date

Abstract This paper reports about the VETESS project results and experience with building a Model-Based Testing toolchain to validate automotive embedded systems. This approach, based on existing test generation and test execution tools, makes it possible to automatically derive and execute functional test cases from UML or SysML models. This process is composed of the following steps: modeling (UML or SysML functional view), abstract test case generation (symbolic execution of the model), concretization (generation of executable test scripts from abstract test cases) and analysis (assignation of the test verdict). This process is automated by a toolchain based on Topcased modeler, Smartesting test generator and Clemessy TestInView. This developed prototype made it possible to demonstrate that Model-Based Testing from UML/SysML models is an efficient way to automate testing process for systems mixing software and hardware parts.

Keywords Model-Based Testing · Automated testing process · UML/SysML notations · Embedded systems

J. Lasalle · F. Peureux
LIFC - Université de Franche-Comté,
16 route de Gray, 25030 Besançon, France
Tel.: +33 (0)3 81 66 66 63
E-mail: {jlasalle,peureux}@lifc.univ-fcomte.fr

F. Peureux
Smartesting R&D center,
18 rue Alain Savary, 25000 Besançon, France
Tel.: +33 (0)3 81 80 47 62
E-mail: fabien.peureux@smartesting.com

F. Fondement
MIPS - Université de Haute-Alsace,
12 rue des Frères Lumière, 68093 Mulhouse, France
Tel.: +33 (0)3 89 33 69 78
E-mail: frederic.fondement@uha.fr

1 Introduction

Model-Based Testing approach (MBT) aims to derive test cases from the specification of the System Under Test (SUT) [16]. The generic process of MBT is depicted on the figure 1. The first step of this approach consists to specify a model that captures the functional behavior of the SUT. From this specification, an (semi-)automatic tool generates test cases, which can be seen as an abstract execution trace of the system. These test cases are abstract because they are defined at the same abstraction level than the model of the SUT. Afterwards, a concretization step makes it possible to produce, from the abstract test cases, test scripts that can be directly executed either on a simulation platform of the system, or directly on the concrete system to be tested. The automation of such test generation process is a strategic issue, since it can replace the (so current) manual development of test cases, which is known as costly and error-prone [20,4].

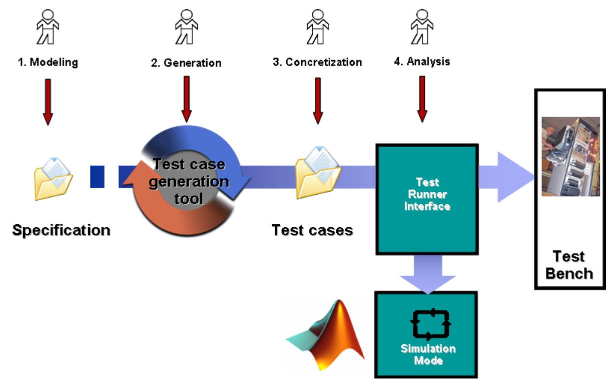


Fig. 1 Model-Based Testing process.

This paper reports about the results and our experience with building an MBT toolchain prototype using UML or SysML notation to describe the SUT. This prototype, based on the existing MBT solution provided by the company `Smartesting`, is intended to demonstrate the *proof-of-concept* that MBT process can increase testing process automation within automotive embedded system domain. More precisely, it aims to show that MBT approach from UML (and SysML) notation is a relevant way to test hybrid architecture (systems mixing software and hardware aspects). This work was supported by the project VETESS (from September 2008 to August 2010) and labelled by the French competitiveness cluster "automotive of future"[1].
This project has rallied the three companies Smartesting[2] (MBT software editor), Clemessy[3] (testing bench provider) and PSA[4] (car manufacturer), and the two academic laboratories MIPS[5] (Model Driven Engineering expertise) and LIFC[6] (MBT expertise).

The remainder of this paper is organized as follows. Section 2 gives an overview of the toolchain developed within the project VETESS to support MBT approach. Section 3 describes the proposed process and the toolchain component developed to achieve it. Section 4 reports our experience about software engineering approach and experimentation feedback. Finally, Section 5 gives conclusions and outlines future work.

2 MBT toolchain overview

The goal of the project VETESS was to develop a toolchain based on the Smartesting MBT process, and to adapt it to address specific testing needs and requirements within the automotive domain. To achieve this goal, this project has focused both on MBT theoretical aspects and technical issues.
The theoretical aspects deal with the capability of the toolchain to take, as input models, representation of automotive embedded system, and the capacity to generate relevant test cases within mecatronic embedded systems. These objectives have been reached by choosing SysML [5] language, as well as UML [12], to specify the test model, and by defining specific model coverage criteria to generate dedicated test cases for embedded system validation. Concerning technical issues, the main goal was to develop an enhanced integration with a modelling environment, a test generation engine and a test execution environment dedicated to embedded systems. A strategic issue is indeed to provide a full automated approach all the way from the test model to the execution of the generated test cases. These goals have been achieved by:

– using an open-source and Eclipse-based modeling tool, namely Topcased,
– using Smartesting Test Designer^{TM} to automate the generation of test cases,
– exporting the generated test cases into a test manager and test execution environment (dedicated to embedded system validation process), namely the Clemessy TestInView platform,
– building up interactions between the modelling environment (Topcased), the test generation tool (Smartesting Test Designer^{TM}) and the test execution platform (Clemessy TestInView) to ensure a fully automated approach.

All these technical choices have been lead by the current practices and interests of PSA end-user engineers, (UML/SysML modeling using open-source software), and by the technologies provided by the companies involved in the project (Smartesting Test Designer^{TM} and Clemessy TestInView platform). Before introducing the overall developed toolchain, each of its component is firstly described in the next subsections.

2.1 MBT process using Smartesting Test Designer^{TM}

Smartesting company has released a Eclipse-based tooled MBT solution to generate and manage functional tests from behavioural models specified with a subset of UML notation called UML4MBT [3]. Symbolic execution techniques are applied to derive abstract test cases according to model coverage criteria or use-case scenarios. These generated abstract test cases can finally be concretized into executable test scripts that aim to evaluate functional conformance of the SUT with respect to the associated UML test model [1,2]. Smartesting Test Designer^{TM} clearly defines the keystone of the toolchain proposed within the project VETESS. In this context, this toolchain can be seen as an extension of the existing Smartesting MBT tooled approach, which is depicted in the figure 2. This approach consists in the following MBT process:

1. A UML4MBT test model is written using IBM Rational Software Architect (RSA).
2. This model is exported into a proprietary pivot file that is managed by Test Designer^{TM} in order to derive test cases from. A test case is composed of a sequence of operation calls with expected output results.

[1] `http://www.vehiculedufutur.com`
[2] `http://www.smartesting.com`
[3] `http://www.clemessy.com`
[4] `http://www.psa-peugeot-citroen.com`
[5] `http://www.mips.crespim.uha.fr/`
[6] `http://lifc.univ-fcomte.fr/`

3. The generated abstract test cases are basically exported into XML proprietary files from which some ad-hoc API can be provided to translate the generated test cases into specific languages or specific environments.

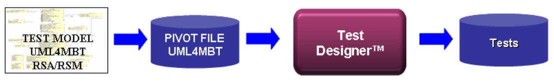


Fig. 2 Smartesting MBT tooled approach.

2.2 UML/SysML modeling using Topcased

UML is widely used as a modelling support in industrial context and is today the main specification language for object modelling. Recently, to provide sufficient features to make this language useful for systems engineers, SysML profile has been created. Even if SysML is a recent modeling language, it is on the rise in industrial domain to specifically address system engineering issues, and several modeling tool already support SysML models. Within the project VETESS, in order to facilitate the integration of the existing tools (especially Smartesting Test Designer^{TM}), and to ease delivery, extensibility and updatability of the toolchain, we decided to use an open-source and Eclipse-based modeling tool. We thus choose Topcased [15], which appeared, at the beginning of the project, the more mature technology satisfying these criteria.

2.3 Test execution using Clemessy TestInView platform

TestInView (TIV) is a test execution platform based on a National Instruments hardware architecture (NI TestStand). It is designed to generate and acquire simple or complex electric signals and to import mathematical models (as Matlab/Simulink) that simulate the behaviour of an item of equipment that is absent from its future working environment. This platform can be used to describe the test sequences, to execute them to within the nearest millisecond and to automatically assess the expected results. One of the challenges of the VETESS project was to create an efficient translation from abstract test cases (embodied by the message sets as generated by Test Designer^{TM}) into continuous signals intended to excite the System Under Test. Within the project VETESS, the test generated with Smartesting Test Designer^{TM} are loaded by TestInView to be executed either on physical or simulated equipment.

2.4 Overview of the VETESS toolchain

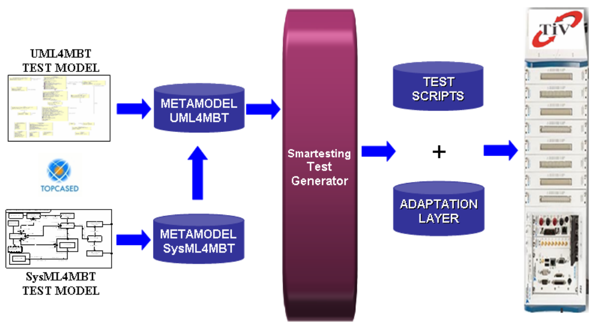


Fig. 3 VETESS final toolchain.

The project VETESS gives rise to the toolchain depicted in the figure 3. The associated test process is defined as follows:

1. A UML or SysML model is realized using Topcased.
2. The standard UML or SysML model is translated into a UML4MBT or a SysML4MBT model. The UML4MBT metamodel was directly derived from [3] while the SysML4MBT metamodel was defined in the context of the project to define the SysML subset that can be handled by the toolchain.
3. In the case of SysML test model, the SysML4MBT file is transformed into a UML4MBT file to be managed by Smartesting Test Designer^{TM}.
4. Smartesting Test Designer^{TM} generates test cases from the model stored in the UML4MBT file.
5. The generated test cases are then exported into Clemessy TestInView platform. During this step, an adaptation layer concretizes the abstract data of the generated test cases into concrete operation calls and values. This layer thus allows to automate the execution of the exported test cases on a simulated system or on a physical test bench.

The next section gives more details about each of these steps and details the dedicated tooling development making it possible to automate this process.

3 MBT process implementation

To automate the MBT process from UML and SysML models, the user functionalities are directly inspired by the current features of Smartesting Test Designer^{TM}, which is the keystone of the toolchain. The implementation phase aims to adapt Smartesting Test Designer^{TM} UML-MBT features to Topcased modeler, to extend it to take SysML models as input of the process, and to connect test generation results to Clemessy TestInView platform.

3.1 Test model design

The test model, specified with UML or SysML diagrams, defines the expected behavior of the SUT. It formalizes the control points and observation points of the system, and the expected dynamic behavior of the system by using OCL notations [19]. However, UML and SysML contain a large set of diagrams and notations that are defined with a flexible way and some freedom to support different semantical interpretations. So, for practical MBT, it is necessary to select a subset of UML and SysML, and clarify its semantics so that MBT tools can interpret the UML models.

Smartesting process natively defines such subset from UML, called UML4MBT, which is precise and complete enough to allow automated derivation of tests from these models. It should be underlined that such restrictions are often necessary in order for the test generation process to be reliable and/or scalable [13, 6]. Thus, a UML4MBT model has to contain one UML class diagram to represent the static view of the system (with classes, associations, enumerations, class attributes and operations) and one UML Object diagram to list the concrete objects used to compute test cases and to define the initial state of the system (it must be an instantiation of the associated Class diagram). The dynamic view can be modelled by OCL expressions on pre or postcondition of class operations, and/or by a Statemachine diagram (annotated with OCL constraints). OCL expressions must actually conform to a subset of the OCL language. As an example, the iterate operation is not supported (more details available in [3]). A UML4MBT Statemachine has some restrictions: it cannot contain parallel states, historic states, fork and join states, and trans-hierarchical transitions.

Within project VETESS, the UML4MBT expressiveness and its restrictions have been integrally preserved. However, we have defined the subset of SysML language, called SysML4MBT, to be supported for test generation. A SysML model has to contain one Block Definition Diagram to represent the static view of the system (with blocks, associations, compositions, enumerations, properties, operations, signals, flow ports...), one Internal Block Diagram to formalize interconnections between blocks, and one or more Statemachine diagrams to specify the dynamic view of the system. SysML4MBT Statemachine can contain much more elements than UML4MBT Statemachine: fork/join, historic and parallel states are indeed allowed. About OCL annotations, the circumflex (^) is allowed to enable signal sending.

A dedicated Eclipse-based plugin was developed to customize and extend Topcased UML and SysML modelling capabilities, and to connect it to Smartesting Test Designer^{TM}. The UML4MBT version of this Eclipse plugin is depicted in the figure 4 (in the remainder of this section, to simplify the presentation of the plugin functions, we only describe UML4MBT plugin since SysML4MBT plugin offers the same services).

The UML4MBT Topcased plugin functionalities can be divided into verification and test generation features. Firstly, the verification panels allow designing and checking the correctness of the models with:

1. Syntactical verification (item 1 in figure 4), which checks that the UML model satisfies UML4MBT restrictions and verify the absence of errors.
2. Functional verification of the model can be performed using a simulator that animates the model, and allows to check manually the correctness of modelled behaviours (frame 2 in figure 4).
3. To complete the native Topcased design facilities, a specific OCL editor (frame 6 in figure 4) has been developed to help engineer specifying OCL constraints (syntactical colouring, completion. . .).

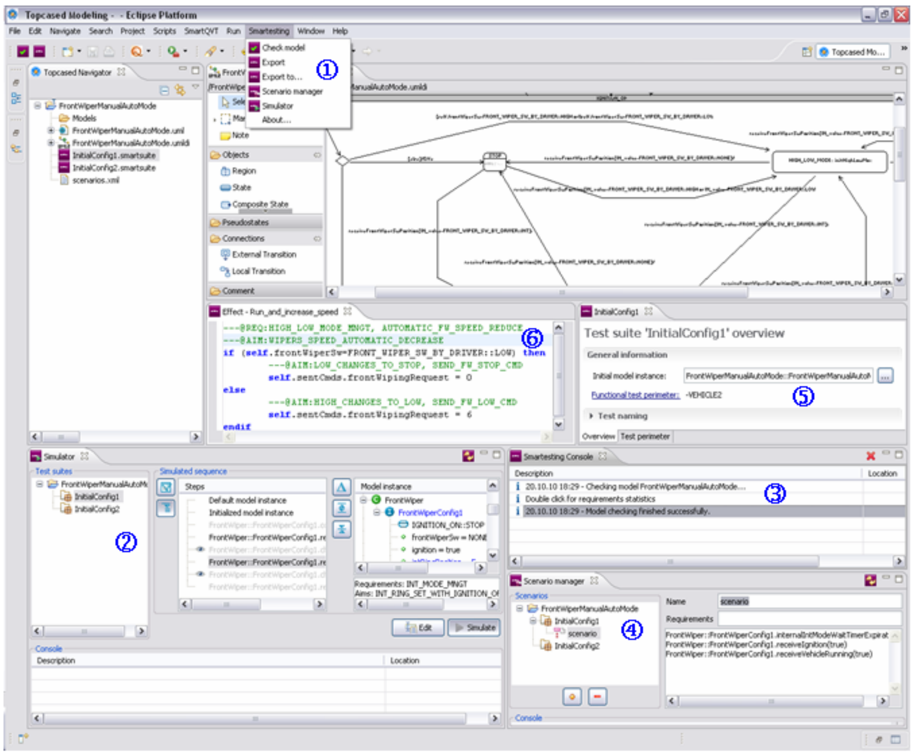
When the syntactical verification process does not find any errors (status is displayed in the frame 3 of figure 4), it is possible to use the following functionalities addressing test generation features:

1. The scenario manager (frame 4 in figure 4) allows to create scenarios (sequences of operation calls) that will be exported as user test cases.
2. The test suite organizer (frame 5 in figure 4) makes it possible to manage several test suites (for instance by filtering model element to be covered by the test generation algorithm).

Finally, in order to generate test cases, user can export the test model in a dedicated XML file, which defines the connection with Smartesting Test Designer^{TM}. The next section gives details about the model transformations performed during this export step.

3.2 Test model export

The Test Designer^{TM} input models are serialized with a XML dialect called TDMODEL. As such, it was necessary to develop software components so that UML and SysML models designed in Topcased can be automatically processed by Test Designer^{TM} to generate abstract test cases from. The developed components, which appear at the left-hand side of figure 3, are the following:



```
---@REQ:HIGH_LOW_MODE_MNGT, AUTOMATIC_FW_SPEED_REDUCE
---@AIM:WIPERS_SPEED_AUTOMATIC_DECREASE
if (self.frontWiperSw=FRONT_WIPER_SW_BY_DRIVER::LOW) then
        ---@AIM:LOW_CHANGES_TO_STOP, SEND_FW_STOP_CMD
        self.sentCmds.frontWipingRequest = 0
else
        ---@AIM:HIGH_CHANGES_TO_LOW, SEND_FW_LOW_CMD
        self.sentCmds.frontWipingRequest = 6
endif
```

Fig. 4 All windows of the VETESS UML plugin.

1. The UML4MBT metamodel.
2. A translation tool inputting UML4MBT models and outputting equivalent TDMODEL models.
3. A model transformation inputting UML2 models as designed in Topcased and outputting equivalent UML4MBT models.
4. The SysML4MBT metamodel that states which concepts of SysML are to be supported by the VETESS toolchain.
5. A model transformation inputting SysML models as designed in Topcased and outputting SysML4MBT models.
6. A model transformation inputting SysML4MBT models and outputting UML4MBT models so that test cases can be generated from SysML models by the Test Designer^{TM} tool.

Topcased, following the example of IBM Rational Software Architect (RSA), relies on the Eclipse Modeling Framework (EMF)[7] for managing models. Such models must conform to a metamodel described in the Ecore metamodeling language. In order for the tool chain to seamlessly integrate Topcased (or any other EMF-based tool), the UML4MBT and SysML4MBT metamodels have been defined in Ecore. Since an EMF model can be easily manipulated in Java, we developed our transformations in Java. We chose not to develop our transformations using a dedicated model transformation language such as QVT [11] for avoiding to embed a model transformation interpretor, and thus simplifying the integration process.

UML4MBT was merely designed as a one-to-one translation of the TDMODEL language, which is actually defined by a metamodel written in Java. The graph of Java objects that represents a model can be serialized and deserialized in XML using the XStream tool[8]. As such, the translation from UML4MBT to TDMODEL is just one-to-one translation between models of the same nature but with different (Java-based) technologies. The translation tools makes thus possible for a UML4MBT model to be processed to generate test cases. The translation was made reversible in order for the legacy TDMODEL models to be processed in an EMF-based environment. For UML2 models designed in Topcased to be processed the same way, a model transformation was developed so that they are transformed into UML4MBT models. As TDMODEL was inspired by (a subset of) UML1.4, UML4MBT also looks like (a subset of) UML1.4 [3]. As such, the UML2 to UML4MBT transformation looks like a UML2 to UML1.4 translation for a subset of UML concepts.

[7] `http://www.eclipse.org/modeling/emf/`

[8] `http://xstream.codehaus.org/`

SysML4MBT was also formalized as an Ecore metamodel. One important point in that numerous concepts are shared between UML4MBT and SysML4MBT. Example such concepts are project, state-machine, and package. As there was a need to translate SysML4MBT models into UML4MBT models in order to generate test cases from SysML4MBT models, there was much to be gained if those two metamodels could keep as similar as possible. This is the reason why we decided to describe it as a UML4MBT dialect, i.e. UML4MBT with some additional concepts (e.g. block) and some other concepts dropped (e.g. instance). In order to create SysML4MBT, we thus added necessary concepts to UML4MBT using the package merge mechanism [10], following the example of the UML metamodel. Moreover, we developed the package unmerge mechanism as the package merge counterpart in order to remove those concepts from UML4MBT that are meaningless in SysML4MBT. Finally, we obtained the SysML4MBT metamodel by applying those changes to UML4MBT ; moreover, we were able to create the one-to-one transformation from SysML4MBT to UML4MBT for those concepts which were shared between the two metamodels. This latter transformation needed to be complemented in order for the concepts added to UML4MBT to be transformed in elements of UML4MBT. An exhaustive introduction of this transformation and dedicated translation rules can be found in [9].

Finally, a model transformation was developed to create SysML4MBT models from standard SysML models as designed in Topcased. To ensure compatibility between toolchain components and obtain more homogeneous code, it should be noted that this model transformation was not implemented using dedicated language, such as ATL [8], but with Java code. Since supported concepts of SysML were all made available in SysML4MBT, the transformation was easy to develop.

The export of the model to a XML file, based on the SysML4MBT metamodel has been developed. The input file of Smartesting Test Designer^{TM} is natively defined by a UML4MBT metamodel-based file. So, a transformation from SysML4MBT notation to this UML4MBT file format has been created and implemented in the dedicated Eclipse-based plugin. This approach makes it possible to re-use the Smartesting test generation process and technologies initially developed for UML4MBT models.

SysML being defined as a UML profile, the translation of SysML4MBT into UML4MBT model consists to leave out the SysML stereotype, which denotes a simple model transformation, which is a widespread approach in Model-Driven Engineering domain [14]. This solution is thus adopted to translate the equivalent concepts (SysML4MBT blocks into UML4MBT classes. . .). For all others SysML4MBT elements that have no corresponding elements in UML4MBT, dedicated rewriting rules are required. For example, all parallel elements of SysML4MBT Statemachine (fork, join, parallel states and multiple Statemachines) are merged using a kind of Cartesian product, historic states in SysML4MBT Statemachine are rewritten by a choice state and a memory variable,... An exhaustive introduction of this transformation and dedicated translation rules can be found in [9].

To ensure compatibility between toolchain components and obtain more homogeneous code, it should be noted that this model transformation was not implemented using dedicated language, such as ATL [8], but with Java code.

3.3 Test case generation and execution

Once the model has been exported (including the translation into UML4MBT language in the case of SysML model), abstract test cases can be automatically generated using test generation strategy implemented in Smartesting Test Designer^{TM} (see figure 5). This strategy basically ensures the covering of each behaviour of the system: it means that, for each behaviour specified in the initial UML or SysML model, at least one test case reaches this behaviour.

Moreover, a specific test generation strategy has been developed for SysML4MBT models in order to produce dedicated embedded test cases focusing on send or receive signals coverage (during the export step, a check box allows to select whether the strategy should be applied or not).

Once test cases are generated, a specific publisher is necessary to translate the generated abstract test cases into executable test scripts. This step relates to several issues:

– Generated test cases refer to discrete dimension and the system to be tested is often continuous.
– Observation should be managed to establish a test verdict.
– Delaying and performance aspects.

To address these items, abstract test cases are translated into test scripts, which are directly executable by an execution environment dedicated to embedded system features, namely Clemessy TestInView platform (see figure 6). A dedicated Java publisher (called adaptation layer) has been developed to export generated test cases to a TestInView file. Moreover, a specific TestStand file is also generated: it defines the test execution patterns (TestInView steps that are necessary to execute the test cases: for example *initialization*, *simulation model loading*, *tests execution*...), and the concretization mapping table of operations used by abstract test cases. This generated table has to be manually filled to map each abstract operations to simulated or real concrete actions.

Finally, to manage continuous system, a last step consists to create real-time test vectors, which are used to define stimuli temporal sequences with an accuracy to one millisecond. Each test is also defined with one specific test vector, which represents the situation of the system in a continuous way. In order to define such real-time test vector, a specific Matlab component has been developed to automatically concretize the generated abstract test cases.

This environment, which is fully integrated to the toolchain, makes it possible to execute test scripts both on simulated system or real physical test bench.

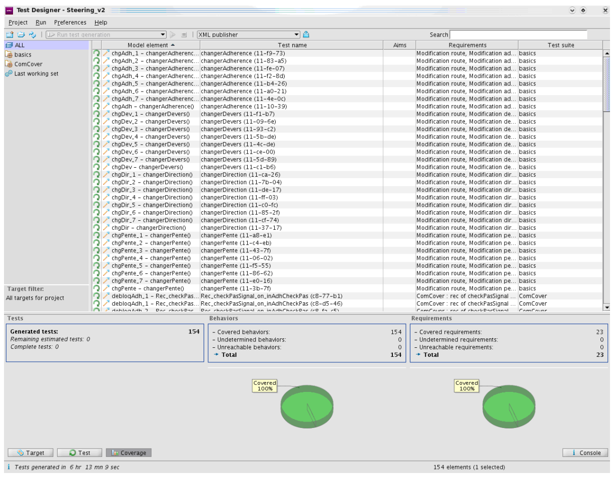


Fig. 5 Smartesting Test Designer^{TM} interface

The next section relates to our experience about software engineering issues and Agile development process, which made it possible to experiment this toolchain very soon during the VETESS project.

4 Software engineering features and experimentations

The motivation of the toolchain consists in demonstrating that MBT approach can efficiently address embedded system validation needs within automotive domain. To reach this goal, we decided to be driven by case-studies and end-user requirements and feedbacks. This

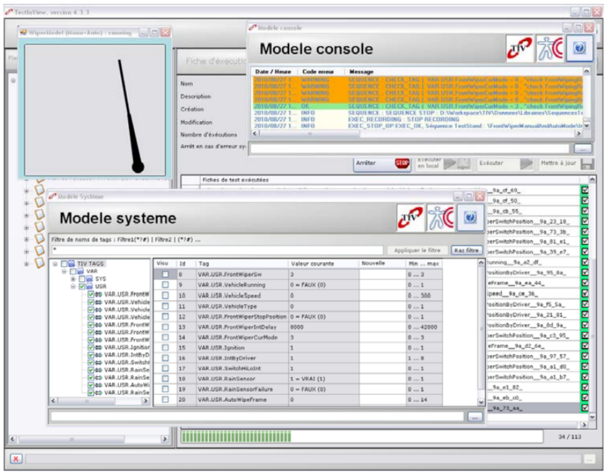


Fig. 6 Clemessy TestInView interface

approach leads to deliver toolchain releases early and frequently during the project. In the context of this project (about 10 development engineers were involved during one and a half years, and about 100km separates the different development teams of the project), one important issue is a big issue concerns the management of scattered development team regarding the common vision of the product, code conflicts and functionality stability. To be able to nevertheless deliver reliable releases of toolchain, our development methodology has followed most of the principles of Agile software development [7] such as:

- Incremental and iterative process including requirements. analysis, design, coding, unit testing and acceptance testing.
- Close interaction and cooperation between developers with a lot of pair programming sessions between academic and industrial partners.
- Face-to-face meeting between developers and end-users (almost one per month).
- Obtaining functional software is more priority than writing a lot of short-lived documentation.

As mentioned in the previous section, all developments have been performed using Java language to maintain the code homogeneous and to facilitate the integration of the components into the toolchain based on Smartesting Test Designer^{TM}, which is fully developed with Java. To coordinate all these development, the following technical approaches and resources have been achieved:

- The whole source code is stored in a Subversion[9] source control repository in order to save all code versions and to easily share it with all project contributors.

[9] `http://subversion.tigris.org`

– JUnit tests are developed to cover the executable code of the toolchain.
– The process of building the toolchain release is automated using MAVEN[10], which automatically compiles Java code and running tests each time Hudson requests it. a modification is committed.
– To provide a continuous integration and uninterrupted non-regression verification, HUDSON[11] software manages the process of building the software and running the tests each time code is committed in the source repository by calling MAVEN.

Early and frequent stable releases of the toolchain made it possible to experiment and evaluate our approach on several case studies during the two year project. The most advanced case study, called "Steering", deals with the specification of the steering column of a car: road variations activate the column through the steering wheel and tyres. The obtained results and feedbacks given by users about each of these experimentations in terms of test relevance, process reliability and approach scalability were always straightaway taken into account to incrementally improve the MBT tooled process and toolchain presented in this paper. Finally, it can be underlined that the case study "Steering"gives rise to the generation of 154 tests (from a SysML model) that have been executed both on simulation platform and physical test bench, as illustrated respectively in figures 7 and 8. A short videotape, describing and exemplifying the toolchain with this case study, is available at [18].

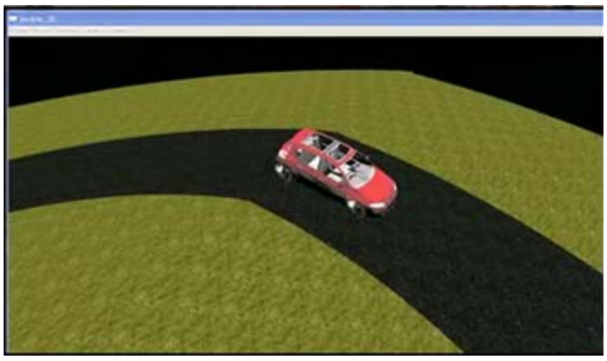


Fig. 7 Simulated environment screenshot

Fig. 8 Physical test bench picture

5 Conclusions and future work

This paper reports about the results and our experience with building an MBT toolchain prototype that aims to automate the generation of executable test scripts from either UML or SysML models, which specify the expected behaviours of the System Under Test. This prototype is based on existing tools that have been adapted and customized to achieve testing process automation. Such an integrated approach and continuous process indeed appear to be a strategic step in moving MBT techniques and methodologies to embedded system domain. Several case-studies has been successfully experimented, and made the *proof-of-concept* that such MBT approach from UML and SysML notations is suitable and can gain benefits within automotive embedded system validation in an industrial context.

The future challenges mainly deal with scalability issues especially about model expressiveness and test case generation time. These scopes of improvement can be addressed, among other, by increasing the UML4MBT and SysML4MBT subsets to address component-based architecture, and by implementing specific model evaluation rules to derive test cases from UML/SysML models. Smartesting company are notably entering these issues within the ITEA2 project VERDE [17] that aims to develop a solution for iterative/incremental development and validation of Real-Time Embedded System.

Acknowledgements The authors would like to thank all people who were involved in the project VETESS, especially M. Alter, M. Bernhard and C. Scherrer from Clemessy, S. Colin and B. Legeard from Smartesting, F. Bouquet and E. Oudot from LIFC, J. Guillet, P.-A. Muller and B. Wittmann from MIPS, and R. May from PSA.

[10] http://maven.apache.org
[11] http://hudson-ci.org

References

1. Bernard, E., Bouquet, F., Charbonnier, A., Legeard, B., Peureux, F., Utting, M., Torreborre, E.: Model-based testing from UML models. In: Proceedings of the International Workshop on Model-based Testing (MBT'2006), *LNCS*, vol. 94, pp. 223–230. Springer Verlag, Dresden, Germany (2006)

2. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F.: A test generation solution to automate software testing. In: Proceedings of the 3^{rd} International Workshop on Automation of Software Test (AST'08), pp. 45–48. ACM Press, Leipzig, Germany (2008)
3. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: A subset of precise UML for model-based testing. In: Proceedings of the 3^{rd} International Workshop on Advances in Model Based Testing (A-MOST'07), pp. 95–104. ACM Press, London, UK (2007)
4. Dias-Neto, A., Travassos, G.: A Picture from the Model-Based Testing Area: Concepts, Techniques, and Challenges. Advances in Computers **80**, 45–120 (2010). ISSN: 0065-2458
5. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann OMG Press (2009). ISBN 978 0 12 374379 4
6. Herrmannsdoerfer, M., Ratiu, D., Koegel, M.: Metamodel usage analysis for identifying metamodel improvements. In: Proceedings of the 3^{rd} International Conference on Software Language Engineering (SLE'10), *LNCS*, vol. 6563, pp. 62–81. Springer Verlag, Eindhoven, the Netherlands (2010)
7. Highsmith, J.: Agile software development ecosystems. Addison-Wesley (2002). ISBN 0-201-76043-6
8. Jouault, F., Kurtev, I.: Transforming Models with ATL, *LNCS*, vol. 3844, pp. 128–138. Springer Verlag (2006)
9. Lasalle, J., Bouquet, F., Legeard, B., Peureux, F.: SysML to UML model transformation for test generation purpose. In: UML&FM'10, 3rd IEEE Int. Workshop on UML and Formal Methods. Shanghai, China (2010)
10. Object Management Group: Unified Modeling Language (UML) infrastructure specification, version 2.3. OMG Document formal/2010-05-03 (2010)
11. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v. 1.1. OMG Document formal/2011-01-01 (2011). MOF QVT Final Adopted Specification
12. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2^{th} edn. Addison-Wesley (2004). ISBN 0 321 24562 8
13. Sen, S., Moha, N., Baudry, B., Jézéquel, J.M.: Meta-model pruning. In: Proceedings of the 12^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS'09), *LNCS*, vol. 5795, pp. 32–46. Springer Verlag, Denver, CO, USA (2009)
14. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Journal of Software **20**, 42–45 (2003)
15. Topcased: Toolkit in OPen-source for Critical Application and SystEms Development. `http://www.topcased.org` (2010)
16. Utting, M., Legeard, B.: Practical Model-Based Testing - A tools approach. Elsevier Science (2006). ISBN 0 12 372501 1
17. Web site of the project ITEA2 VERDE. `http://www.itea-verde.org` (2010)
18. Web site of the project VETESS. `http://lifc.univ-fcomte.fr/VETESS` (2010)
19. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley (1996). ISBN 0 201 37940 6
20. Zhu, H., Belli, F.: Advancing test automation technology to meet the challenges of model-based software testing. Journal of Information and Software Technology **51**(11), 1485–1486 (2009)