Automatic Test Concretization to Supply End-to-End MBT for Automotive Mechatronic Systems

Jonathan Lasalle, Fabien Peureux *
Laboratoire LIFC (EA-4269)
Université de Franche-Comté
16, route de Gray
25030 Besançon, France
{jlasalle,fpeureux}@lifc.univ-fcomte.fr

Jérôme Guillet
Laboratoire MIPS (EA-2332)
Université de Haute-Alsace
12, rue des Frères Lumière
68093 Mulhouse, France
jerome.guillet@uha.fr

ABSTRACT

This paper presents an effective end-to-end Model-Based Testing approach to validate automotive mechatronic systems. This solution takes as input a UML/OCL model describing the stimuli of the environment that can excite the mechatronic System Under Test. It applies model coverage criteria to automatically generate test cases, and finally takes an offline approach to translate the generated test cases into executable test scripts that can be executed both on simulation model and physical test bench. The mechatronic System Under Test is then tested against a Matlab/Simulink simulation model, which defines the test oracle. This tooled and automated approach has been successfully experimented on a concrete case study about the validation of a vehicle front axle unit. This experimentation enabled us to validate our approach, and showed its effectiveness in the validation process of mechatronic systems.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.8 [**Operating Systems**]: Performance; I.6.4 [**Simulation modeling**]: Model Validation and Analysis

General Terms

Experimentation, Reliability

Keywords

Model-Based Testing, automated testing process, UML/OCL notations, Matlab/Simulink simulation, mechatronic systems

*Fabien Peureux is also external scientific consultant for:
Smartesting R&D center
18, rue Alain Savary
25000 Besançon, France

ETSE '11, July 17, 2011, Toronto, ON, Canada

1. INTRODUCTION

The growing complexity of software-intensive, real-time embedded systems, combined with constant quality and time-to-market constraints creates new challenges for engineering practices in this domain. Testing is today the principal validation activity in industrial context to increase the confidence in the quality of systems, and appears therefore to be strategic. Nevertheless, the industry is still faced to the use of specific validation techniques which rely on manual, repeated and tedious efforts. In the last decade, Model-Based System Engineering (MBSE) methodologies have emerged on the sharing and standardisation of embedded software technologies [9]. These approaches put a strong emphasis on the use of models at the different steps of the system specification, and even for testing activity. In this particular purpose, code is no longer the single source for selecting test cases: testing against original expectations can be done using Model-Based Testing (MBT) approach [13].

The main idea behind MBT is that a behavioural model of the system, called a high-level test model, can be adopted as the basis for automatically deriving test cases following different coverage criteria. MBT aims thus to ensure that the final product conforms to the initial functional requirements. It promises higher quality and conformance to the respective functional safety and quality standards, and increased automation of the testing process [6]. Even if some obstacles remain to a full-scale deployment, MBT approaches are today getting closer and closer to an industrial reality. In the one hand, theoretical concepts (and associated tools) are now mature enough to be applied in many application areas. In the other hand, MBT approaches have still to provide a better degree of automation, especially to translate the generated test cases into executable test scripts, to empirically show that it can give a good Return On Investment [7].

In this paper, we propose an end-to-end tooled MBT solution for the validation of mechatronic systems including real-time and continuous execution issues. This solution takes as input a UML behavioural model of the SUT environment, uses model coverage criteria to automatically generate specific configuration (test cases), and finally takes an offline approach to translate these configurations into test scripts, which can be executed both on a Matlab/Simulink execution model and physical test bench. In the same time, the execution model is indeed executed to compute the expected values, that are compared in real-time with the results obtained on the physical test bench.

This proposed solution, providing a suitable, automated and repeatable process, has been experimented using a concrete case study about the validation of a vehicle front axle unit.

This paper is organized as follows. Section 2 introduces a simplified version of the Steering case study, which will be used in the next sections to illustrate our approach. Section 3 presents an overview of the MBT process and its associated toolchain about model specification and test generation steps. Section 4 characterizes the concretization of the generated test cases into executable test scripts. Section 5 reports our experience about test execution process. Finally, section 6 gives conclusions and outlines future work.

2. STEERING CASE STUDY

The testing approach has been applied on a real case study about vehicle front axle unit dynamics. This study more precisely concerns the behavior of the steering column and the dampers of a motorized vehicle. The mechanical part of the steering dynamic, depicted in Figure 1, contains:

- A steering wheel that allows the driver to activate the steering column.
- A steering column that transfers loads of the steering wheel to the rack rail.
- A rack rail that transforms the rotation movement of the steering column to a translation motion.
- Hubs that allows to turn the wheels using the rack rail movement.

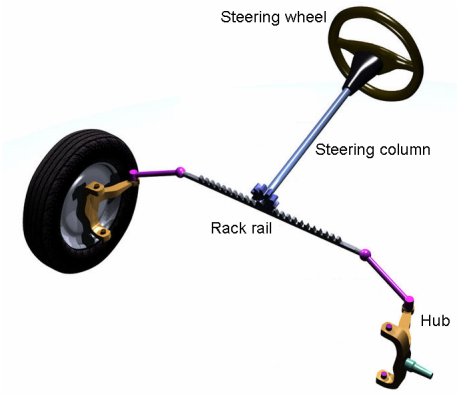

Figure 1: Mechanical overview of the steering.

From testing point of view, the steering vertical dynamic of the vehicle defines the SUT. Figure 2 depicts in broad outline the steering vertical dynamic of the vehicle, which is composed of the following mechanical parts:

- Two tires fixed on two suspension arms.
- A stabilizer bar between both arms.
- A suspension composed of a damper and a spring connected to the vehicle body.
- A vehicle body with free rotations and translations.

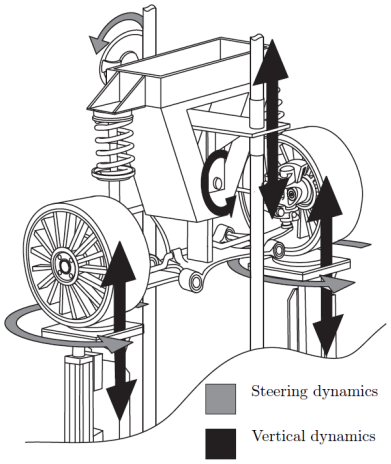

Figure 2: Broad outline overview of the case study.

Testing such a vehicle full system requires to take into account the complete driving framework that is defined, as shown in Figure 3, by the environment, the driver and the vehicle. This framework has indeed been proposed in several papers that consider the driving task as an evolution of the Driver-Vehicle-Environment triplet [8].

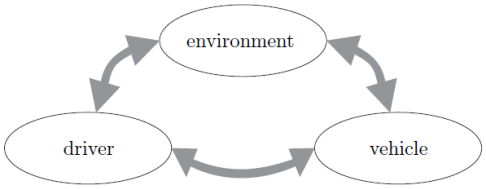

Figure 3: Driving framework.

To simplify the presentation, the impact of the driver, which is modeled by several controllers, is not discussed in this paper. Therefore, we consider that steering vertical dynamic of the vehicle is only stimulated by its environment through the layout of the road (which can contain bends, slopes, banks, holes...) in relation with its own dynamic characteristics (weight, speed...).

This simplified version of the Steering case study is used in the rest of this paper to illustrate our MBT approach and the executable script generation technique, which are introduced in the next sections.

3. MBT APPROACH

MBT refers to the processes and techniques for the automatic derivation of abstract test cases from abstract formal models (abstract because relying on a model) and the generation of concrete tests from abstract tests.

In this section, we briefly describe the MBT approach that is used in the context of mechatronic systems to derive test cases from abstract formal models (called test model) written with the Unified Modeling Language (UML [11]) and the Object Constraint Language (OCL [15]). The first subsection gives an overview of the MBT process, subsections 2 and 3 respectively introduce the test model specification and the automated test generation steps. The approach is illustrated with the Steering case study.

3.1 Overall MBT Toolchain

Our MBT approach is based on the Test Designer^{TM} tool provided by the company Smartesting[1]. Test Designer^{TM} implements an integrated and automated MBT solution from UML/OCL models [2] that defines the keystone of the proposed MBT toolchain, as shown in Figure 4.

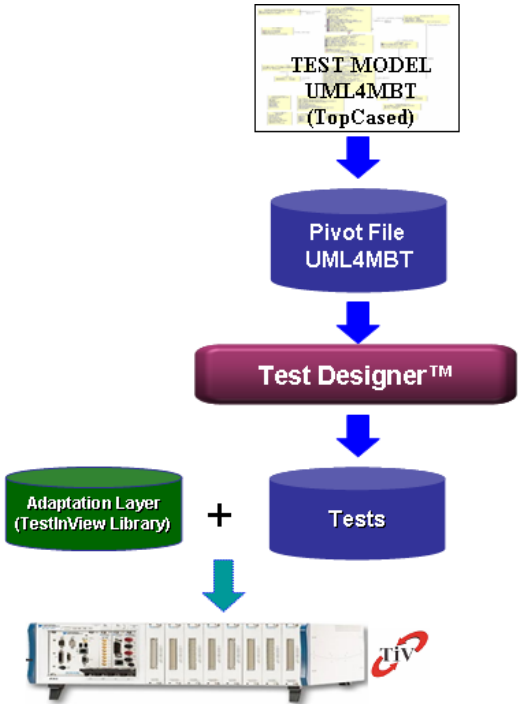

Figure 4: MBT toolchain.

This toolchain takes as input a test model defined by the UML4MBT language [4], which is a subset of UML and OCL notations. The UML4MBT models have a precise and unambiguous meaning, so that the behaviour of those models can be understood and manipulated by the Test Designer^{TM} technology. OCL expressions indeed provide the expected level of formalization necessary for model-based testing modelling. This precise meaning makes it possible to simulate the execution of the models and to automatically generate test cases [3].

Each generated test case is typically an abstract sequence of high-level actions from the UML4MBT models. These generated test sequences contain the stimuli to be executed, but also the expected results, obtained by resolving the associated OCL constraints.

To make them executable using a test automation tool, a further concretization step is then needed to automatically translate each abstract test case into a concrete (executable) script. This translation is performed in an adaptation step: this step involves a SUT-specific adaptation layer, designed once for the project by the validation engineer, to define script patterns and mappings. They are basically used to translate abstract names from the model into concrete names in the target language, and to translate the test case structure into scripts directly executable on a simulated system or on a physical test bench.

Generated test cases can also be exported to a large variety of format including customizable HTML or proprietary XML files (for documentation for example). In the scope of the Steering case study, a dedicated real-time platform has been used to execute the generated test scripts. This framework, called TestInView, will be introduced and precisely described in section 4.

In this paper, we propose to adapt this process to validate automotive real-time mechatronic system. The two major issues of this work are thus about the automation of the process (especially the concretization step performed by the adaptation layer) and the management of continuous systems. Indeed, in order to represent behaviors of the steering vertical dynamic, it is necessary to consider physical and mathematical rules that cannot be modelled using a UML4MBT model. UML4MBT, which defines the specification language taken as input of the toolchain, only describes discrete actions. Consequently, it cannot be used to express the continuous behaviours of the SUT.

Regarding that, our approach consists in modelling, using UML4MBT, the environment of the SUT in a discrete manner, and in deferring the management of continuous time issues at the concretization level. In this way, the test model describes the dynamic of the SUT environment, meaning how the SUT can be stimulated by its environment (and not how it evolves against these stimuli). The expected behaviours of the SUT is thus computed latter during the adaptation step of the process, which then appears more complex than a simple mapping between abstract and concrete data.

The next sections introduce in a more detail way each step of this process and illustrates the corresponding toolchain results using the Steering case study.

3.2 Modeling

UML is widely used as a modelling support for Model-Based Testing [5]. There are several reasons for this interest. Firstly, UML provides a large set of diagrammatic notations for modelling purposes, with several complementary representations. A static representation (i.e. class diagrams) can be used to model the points of control and observation, and the data that represents the abstract state of the modelled system. Secondly, OCL associated with UML makes it possible to have precise models: this means that the expected behaviour can be formalized using OCL, and test cases can also be derived in an unambiguous way [1, 12].

Our test generation approach is based on a UML test model that synthesizes the behavior of the environment of the SUT (i.e the layout of the road), and not the behavior of the SUT as usually performed in traditional MBT approach. This test model is based on 2 types of UML diagrams:

- A class diagram defines entities (name, attributes), the relationships between these entities (by association) and actions (by operations) carried-out to change the value of these entities. This defines the static view of the modelled environment.
- An object diagram represents instances pertaining to the different classes of the test model. Thus, this diagram defines the initial value of the entities representing the environment.

[1] `http://www.smartesting.com`

Finally, the dynamical aspect of the test model is captured by annotating the operations of the class diagram with OCL formula, which precisely formalize their expected behaviour. This level of precision makes the model formal and allows the test generation to be completely automated.

To model the environment of the steering vertical dynamic using UML4MBT, only one class diagram and one object diagram are needed. The class diagram, shown in Figure 5, contains only one class (called `Road`) and some enumerations classes defining set of possible values.

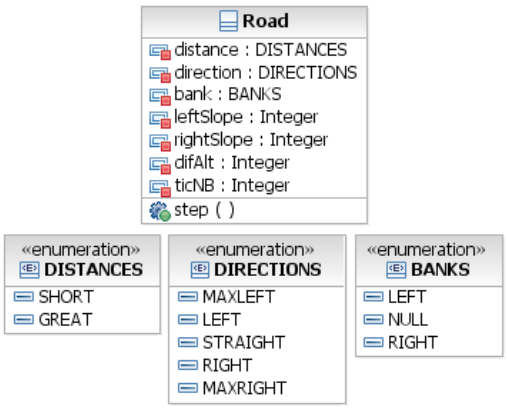

Figure 5: Class diagram of the Steering case study.

The class `Road` is defined by the following 7 attributes (the two last attributes are used by OCL constraints to make sure that the generated roads are realistic and executable in safety on the test bench):

- the `distance` that can be `short` or `great`.
- the `direction` that can be `straight`, `maxleft`, `left`, `maxright` or `right`.
- the `bank` that can be `NULL`, `left` or `right`.
- the slopes (`leftSlope` and `rightSlope`) that can be different between the left and the right side of the vehicle (in order to represent pothole for instance). Each slope (left and right side) is represented by an integer value comprised between minus two and two.
- the constant `difAlt` that memorizes the higher gap of the slopes between the two previous road parts.
- the counter `ticNB` that is incremented each time the road attributes are updated.

Figure 6 shows the corresponding object diagram describing the initial state of the model.

The class `Road` also contains a single operation called `step`. A step represent a part of a road. Each time this operation is executed, it means that the vehicle run a road part having the characteristics defined by the 5 parameters of the operation: the distance (`inDistance`), the direction (`inDirection`), the bank (`inBank`), the left and the right slopes (`inLeftSlope` and `inRightSlope`). This operation updates the first five attributes of the class `Road` with the abstract value given by the corresponding input parameter.

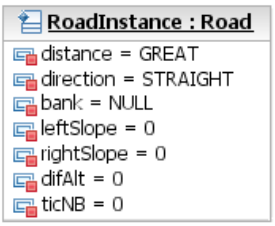

Figure 6: Object diagram of the Steering case study.

As mentioned above, there are some restrictions for the succession of two steps to ensure that the generated roads plots are practicable in a real context. For example, a road part with a left bank just after a road part with a right bank (or vice/versa) is not allowed (a null bank between them is required). That is why the precondition of the operation `step` contains a lot of OCL constraints to ensure the correctness of the generated road. The previously given restriction is expressed by the following OCL constraint:

```
(self.bank<>BANKS::LEFT or inBank<>BANKS::RIGHT)
                        and
(self.bank<>BANKS::RIGHT or inBank<>BANKS::LEFT)
```

Once the test model is achieved, it can be used to automatically produce test cases using the test generation tool. The next subsection presents this step.

3.3 Test Generation

Basically, automatic test generation algorithm carries out a systematic coverage, either of all behaviours (effects defined by the OCL constraints) of the test model, or of predefined use-case scenarios given by the validation engineer. Each generated test corresponds to a sequence of operations taking the form of a 3-part structure:

1. the preamble (potentially empty) is a sequence of operation calls that place the system in a state that allows to activate the targeted behavior.
2. the test body contains the operation call (or sequence of operation calls in the case of use-case scenarios) that execute the given behavior.
3. the postamble (optional) that puts the system in the same state than before the execution of the preamble in order to link the execution of the tests.

Within the Steering case study, we have thus defined some use-case scenarios to be able to cover a lot of configurations. Moreover, in order to easily investigate more configurations, we have also implemented a random-based algorithm. By selecting a number of expected sequences and a number of operation calls per sequences, the test generation engine automatically computes, in a random way, a set of test cases corresponding to the selected parameters.

All these algorithms allow to obtain road layouts that are defined by a sequence of *step* operation calls. Such a sequence, containing a large panel of configurations, is given in Figure 7.

```
step(GREAT, STRAIGHT, NULL, 0, 0)
step(GREAT, STRAIGHT, RIGHT, 0, 0)
step(SHORT, STRAIGHT, NULL, 0, 0)
step(GREAT, RIGHT, LEFT, 0, 0)
step(GREAT, MAXLEFT, NULL, -1, -1)
step(SHORT, STRAIGHT, NULL, 0, 0)
step(SHORT, STRAIGHT, NULL, -2, 2)
step(SHORT, STRAIGHT, NULL, 2, -2)
step(SHORT, STRAIGHT, NULL, 0, 0)
step(SHORT, RIGHT, NULL, -1, 1)
step(SHORT, STRAIGHT, NULL, 2, 0)
step(GREAT, MAXLEFT, RIGHT, 0, 0)
step(SHORT, STRAIGHT, NULL, 2, 0)
step(SHORT, STRAIGHT, NULL, 0, 0)
step(SHORT, STRAIGHT, NULL, -2, 0)
step(GREAT, STRAIGHT, LEFT, 1, 1)
step(GREAT, STRAIGHT, NULL, -1, -1)
step(GREAT, STRAIGHT, NULL, 0, 0)
```

Figure 7: Steering case study scenario.

This scenario is described using a graphical representation in Figure 8, where perpendicular lines separate the different steps, gray lines represent a flat road stretch, light gray lines define a downhill part, black lines define an ascending part, and finally arrows depict the various banking. This graphical representation is automatically generated using a dedicated HTML publisher that directly takes as input the generated test cases. This representation makes it possible to check the correctness of the generated road very soon in the process.

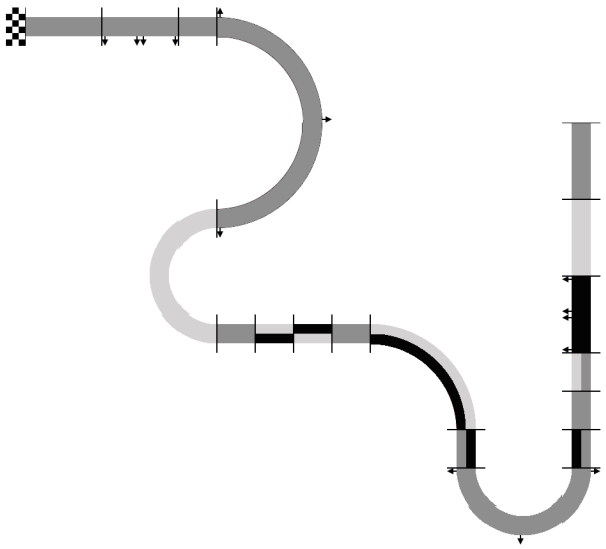

Figure 8: Steering case study graphical scenario.

A generated abstract test case describes a sequence of stimuli that defines the behaviour of the SUT environment, i.e. a road that the vehicle has to run. These generated test cases are abstract, i.e. they are not directly executable on the real system. It is necessary to concretize them to link abstract operations and data with the corresponding real actions and values. In case of continuous time systems, as UML4MBT models do not contains time information, the addition of temporal structures is necessary during the concretization step. The next section introduces the features of the automated concretization and execution of the resulting concrete test cases.

4. TEST CONCRETIZATION

The concretization step has two main purposes: first, translate abstract operations into real actions and inject observation operations into test cases to be able to assign a verdict. In this section, we call *abstract operations* the operations of the generated abstract test cases, and *observation operations* the operations that has to be injected in the concrete scripts to perform observation.

In case of continuous time systems, as UML4MBT models do not contains time information, the addition of temporal structures is necessary during the concretization step. For abstract operation, this is done by specific test publishers which take into account the sequential aspect of the operation calls (using a classical mapping between abstract data and concrete values). For observation operations, a SUT execution model must be provided. Generally, such execution models are created during system design step to simulate the expected real system. This execution models are used to compute the expected SUT values. Therefore, they are used in two ways: one the one hand, to simulate the system and validate the specifications of the retained solution, and on the other hand to compute the expected values of the real system (oracle).

The concretization and execution steps are managed by the TestInView platform presented in subsection 4.1. To address the case of continuous systems, the execution models are created to compute the expected values with the Matlab/Simulink platform presented in subsection 4.2. Finally, the whole process of concretization and tests execution on the SUT is given in the subsection 4.3.

4.1 TestInView Platform

TestInView is a software platform, based on TestStand National Instrument software[2], which allows to automatically execute tests during the complete conception system cycle. It is commercialized by the Clemessy company[3]. This platform is particularly designed for testing embedded electronic systems, but it is also able to manage mechanical test bench. It can be used on the whole testing process, from the modeling to the execution on the test bench. The solution architecture is based on a supervisor computer which manages tests execution. If the SUT is physically available, the supervisor is connected to the real time controller which manages the SUT.

In our case, TestInView is used in two different ways:

- By executing tests locally on the Supervisor. In this case, an execution model is loaded and executed into TestInView to simulate the SUT.
- By using the real time controller connected to a test bench.

Firstly, local execution allows to validate the execution model of the SUT. When this model is definitely validated

[2]`http://www.ni.com/teststand/`
[3]`http://www.clemessy.fr/`

and if a physical prototype of the SUT is available, the tests must be executed on it. To do this, the real time constraint is added and TestInView manages the execution of the tests on the SUT connected to the real time controller.

In case of discrete event systems (e.g. embedded electronic systems), the oracle can be directly obtained from the UML4MBT model and none other model of the system is needed. In case of continuous system, the SUT execution model must be used to create the expected values of the test scenarios. The execution model is designed on the Matlab/Simulink software platform.

4.2 Matlab/Simulink

Matlab is a numerical computation platform which includes an environment for multi-domain simulation and model design for dynamic and embedded systems (Simulink[4]). Based on components libraries, Simulink is a graphical environment that allows to design, simulate, implement and test time-varying systems. The SUT designed on Simulink defines a model with inputs and outputs. Inputs must be excited by the abstract operations described by the tests. Outputs are used to be compared with the oracle. When the model is validated, it becomes the oracle and it is used to compute expected values.

With real time constraint, model outputs are computed thanks to a solver. Solvers use a fixed-step time which determines the output computation times. As the platform TestInView is used during the whole testing process, the Simulink model must be first exported and integrated into the platform, and then executed in real time. Exportation is done using SIT (Simulation Interface Toolkit) from National Instrument.

4.3 Concretization Process

After the test generation, tests are exported to the TestInView platform: each one is translated into the TestInView file format which describes the executable sequence (sequence file). The latter is generated in a TestStand file format. A dedicated Test Designer^{TM} publisher was developed in order to generated automatically those files. This step concerns a traditional (manually designed) mapping between abstract data and concrete values and structures.

The TestStand sequence, *test vector* and execution model, can be loaded into TestInView and connected with the corresponding target, that is to say, either a simulated system, either a physical test bench. The complete process is depicted on the figure 9.

When all files are provided to TestInView, the tests execution could be performed. Thanks to the oracle, expected values are compared with real values during the tests. Comparison is done by *Sanctions* which define the comparison modes between expected values and reals values (such as equality, superior, inferior, validity area ...). An automatically generated report summarizes in details the executed tests and especially the results of the evaluated sanctions. By a SUT states backup performed during the tests, local execution can re-evaluate sanctions without using the SUT.

4.4 Steering Case Study Illustration

Once tests are generated, they can be exported to the TestInView platform. At this stage, tests are ever abstract.

[4] `http://www.mathworks.fr/`

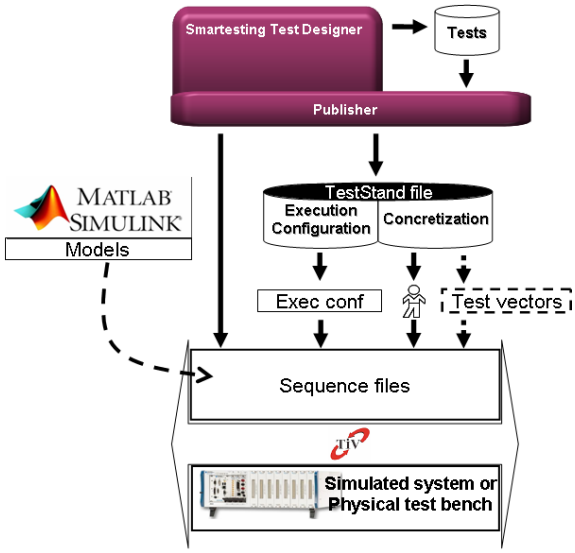

Figure 9: Concretization process.

Then, it is necessary to concretize them. This step consist to associate each abstract part of the model to a real concrete value. The table 1 represent the concretization values for this case study.

DISTANCE	GREAT	20 m
	SHORT	0.5 m
DIRECTION	MAXLEFT	0.06
	LEFT	0.05
	STRAIGHT	0
	RIGHT	-0.05
	MAXRIGHT	-0.06
BANK	NULL	0 rad
	LEFT	+0.08 rad
	RIGHT	-0.08 rad
SLOPE	2	0.35 rad
	1	0.1745 rad
	0	0 rad
	-1	-0.1745 rad
	-2	-0.35 rad

Table 1: Concretization values of the Steering case study.

Those values allow to calculate the real road layout. To do so, we use a Matlab program that calculate concrete characteristics of the vehicle during time. For example, the following code calculate the position of the vehicle according to slope and bank, as shown in Figure 10, where:

- `Xtraj` defines the trajectory of the vehicle gravity center along X axis,
- `Ytraj` defines the trajectory of the vehicle gravity center along Y axis,
- `ZtrajL` defines the height of left wheels,
- `ZtrajR` defines the height of right wheels,
- `Course` defines the course of the road.

```
%Slope calculation
slope = (leftSlope+rightSlope)/2;
Xtraj(indexPoint,1)=Xtraj(indexPoint-1,1)+
   r(indexPoint,1)*sin(pi/2-slope)*
   cos(course(indexPoint,1));
Ytraj(indexPoint,1)=Ytraj(indexPoint-1,1)+
   r(indexPoint,1)*sin(pi/2-slope)*
   sin(course(indexPoint,1));

%Bank calculation
Ztraj(indexPoint,1)=Ztraj(indexPoint-1,1)+
   r(indexPoint,1)*cos(pi/2-slope);
ZtrajL(indexPoint,1)=Ztraj(indexPoint,1)+
   r(indexPoint,1)*cos(pi/2-leftSlope)+
   b/2*sin(bank);
ZtrajR(indexPoint,1)=Ztraj(indexPoint,1)+
   r(indexPoint,1)*cos(pi/2-rightSlope)-
   b/2*sin(bank);
```

Figure 10: Concretization rules of the Steering case study.

Then, using this program, we obtain the table depicted on the figure 11 that represents concrete values of the road configuration during time (first column).

Time	Xtraj	Ytraj	Course
0.000000	0.000000	0.000120	0.000120
0.010000	0.000094	0.000183	0.000183
0.020000	0.000282	0.000257	0.000257
0.030000	0.000564	0.000343	0.000343
0.040000	0.000939	0.000440	0.000440
0.050000	0.001409	0.000629	0.000629
0.060000	0.001973	0.000852	0.000852
0.070000	0.002630	0.001109	0.001109
0.080000	0.003382	0.001400	0.001400
0.090000	0.004227	0.001726	0.001726
0.100000	0.005167	0.002086	0.002086
0.110000	0.006200	0.002480	0.002480

Figure 11: Example of vector data of the Steering case study.

5. TEST EXECUTION

At this stage, a Simulink execution model calculates the vehicle reaction according to the road characteristics. Thus, we obtain theoretical response of the vehicle on this road. The last step consists to the comparison of those theoretical results with practical results by launching the layout on a test bench. This last step of our approach is described through the Steering case study experimentations.

5.1 Execution Model

A simulation version of the SUT was developed: the graphical part was realized using OpenGL whereas data calculations were done using a Simulink model. The simulation module is depicted by the figure 12. It calculates expected results.

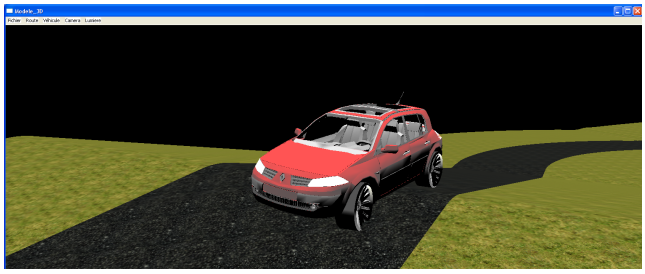

Figure 12: Simulator of the Steering case study.

The simulation mode allows to determinate the expected results, but also to validate the Simulink model real time execution. Indeed, several outputs of the model are connected to the test bench to activate it. If the real time execution of the tests is not available, i.e. some cycle time of execution are lost, the Simulink model could become unstable. In this case, outputs could have some unexpected values. If these outputs are connected to an actuator of the test bench, unexpected motions could damage it. Therefore, simulation mode also allows to validate all previous developments before the execution on the real system.

5.2 Physical Test Bench

The vehicle front axle unit dynamic test bench (shown in Figure 13) includes all mechanical parts involved in the steering dynamic and vertical dynamic of the front part of a vehicle.

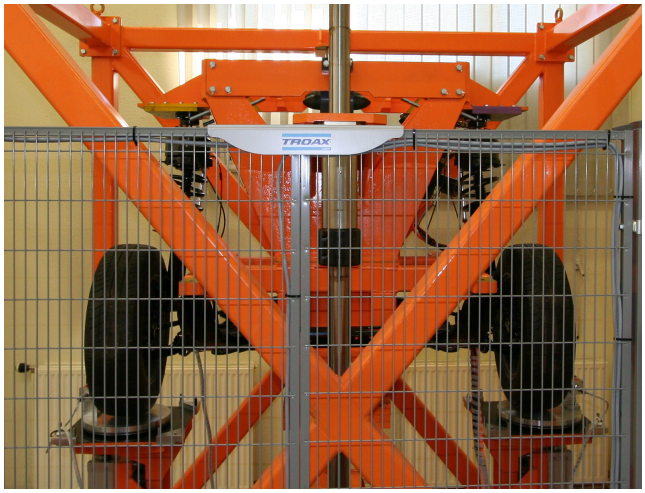

Figure 13: Test bench of the Steering case study.

The test bench takes only into account the front vehicle part. Therefore, the body is replaced by a mass which corresponds to the vehicle front mass. The latter has a free vertical translation and a free roll rotation. Three actuators allow to generate dynamic load: one steering motor which replaces the steering wheel, and two vertical electrical jacks under tires. The steering motor is used to control the steering dynamic and the vertical jacks simulates a vertical road profile.

5.3 Experimentation Results

In the same time, the full Simulink model of the vehicle is executed to compute reference values. These values are compared, in real-time, with the test bench sensor values and several sanctions allow to check the SUT behavior.

An illustrative example can be found in Figure 14, which draws the expected and the real value of the vehicle roll angle. The black lines represent the sanction on this output defined by a validity area of 5% of the expected value. The execution of such scenario on this test bench is available at the end of the video located in [14].

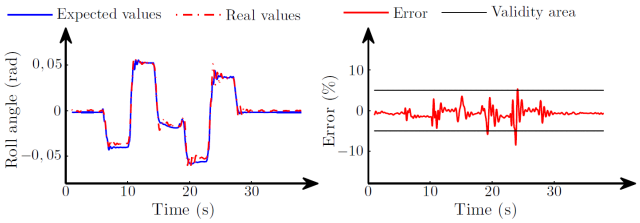

Figure 14: Roll angle of the Steering case study.

6. CONCLUSION

This paper presents a fully automated and suitable end-to-end test framework, based on a UML/OCL MBT approach, to address automotive mechatronic domain. It also reports about successfully experimentation results carried out to validate the steering vertical dynamic of a vehicle. Such an integrated approach and continuous process, including the automation of executable test script generation, constitute strategic issues for testing activity to save time and to increase safety and quality.

Within this test framework, the future works mainly concern the nature of the test model and so the purpose and the relevance of the generated test cases. In this way, we are extending the expressiveness of the test model, which is the input of the MBT solution. Therefore, we are currently implementing a prototype to take into account embedded systems characteristics by using SysML notation to specify the test model [10]. We also plan to manage real-time issues by using the UML MARTE profile: this feature makes it possible to manage real-time constraints in the test model, and thus allows to generate new types of test cases.

7. ACKNOWLEDGMENTS

This work has been supported within the French project VETESS [14] (from September 2008 to August 2010) and labelled by the French competitiveness cluster "automotive of future" (`http://www.vehiculedufutur.com`).

8. REFERENCES

[1] M. Benattou, J.-M. Bruel, and N. Hameurlain. Generating test data from ocl specification. In *Proceedings of the Int. Workshop on Integration and Transformation of UML models (WITUML'02)*, 2002.

[2] E. Bernard, F. Bouquet, A. Charbonnier, B. Legeard, F. Peureux, M. Utting, and E. Torreborre. Model-based testing from UML models. In *Proceedings of the Int. Workshop on Model-based Testing (MBT'2006)*, volume 94 of *LNCS*, pages 223–230, Dresden, Germany, October 2006. Springer Verlag.

[3] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux. A test generation solution to automate software testing. In *Proceedings of the* 3^{rd} *Int. Workshop on Automation of Software Test (AST'08)*, pages 45–48, Leipzig, Germany, 2008. ACM Press.

[4] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *Proceedings of the* 3^{rd} *Int. Workshop on Advances in Model Based Testing (A-MOST'07), colocated with ISSTA'07*, pages 95–104, London, UK, July 2007. ACM Press.

[5] D. Buchs, L. Pedro, and L. Lucio. Formal Test Generation From UML Models. *Dependable Systems: Software, Computing, Networks: Research Results of the DICS Program*, LNCS 4028:145–171, 2006. ISSN: 0302-9743.

[6] A. Dias-Neto and G. Travassos. A Picture from the Model-Based Testing Area: Concepts, Techniques, and Challenges. *Advances in Computers*, 80:45–120, July 2010. ISSN 0065-2458.

[7] E. Dustin, T. Garrett, and B. Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Addison Wesley Professional, 2009. ISBN 0 32 158051 6.

[8] D. Ehmanns and A. Hochstadter. Driver-model of lane change maneuvers. In *Proceedings of the* 7^{th} *World Congress on Intelligent Transportation Systems*, Turin, Italia, November 2000.

[9] J. Estefan. Model-Based Systems Engineering (MBSE) Methodologies. Survey INCOSE-TD-2007-003-01.B, MBSE Initiative and INCOSE Group, June 2008.

[10] J. Lasalle, F. Bouquet, B. Legeard, and F. Peureux. SysML to UML model transformation for test generation purpose. In *Proceedings of the* 3^{rd} *IEEE Int. Workshop on UML and Formal Methods (UML&FM'10)*, Shanghai, China, November 2010.

[11] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 2^{th} edition, 2004. ISBN 0 321 24562 8.

[12] A. Shaukat, M. Z. Iqbal, A. Arcuri, and L. Briand. A search-based ocl constraint solver for model-based test data generation. Technical Report 2010-16, Simula Research Laboratory, October 2010.

[13] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Morgan and Kaufmann, 2006. ISBN 0 12 372501 1.

[14] The VETESS web site. `http://lifc.univ-fcomte.fr/vetess/`, 2010.

[15] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1996. ISBN 0 201 37940 6.