SysML to UML model transformation for test generation purpose

Jonathan Lasalle[1] Fabrice Bouquet[1,2] Bruno Legeard[1,2] Fabien Peureux[1,2]

1: Laboratoire d'Informatique de l'Université de Franche-Comté
16, route de Gray - 25030 Besançon, France
e-mail: {jlasalle,fbouquet,blegeard,fpeureux}@lifc.univ-fcomte.fr

2: Smartesting
18, rue Alain Savary - 25000 Besançon, France
e-mail: {bouquet,legeard,peureux}@smartesting.com

Abstract

The work introduced in this paper is in line with an original Model-Based Testing approach by taking as input a SysML specification of a system under test and automatically translating it into an equivalent behavioural UML model. This generated UML model is finally used to derive test cases and executable test scripts. This test generation process is supported by an existing UML/OCL Model-Based Test generation tool. This paper focuses on the definition of the subset of SysML notation supported by this Model-Based Testing approach, and proposes rewriting rules to derive UML test model from SysML model.

1 Introduction

With testing, a system is executed with a set of selected stimuli, and observed to determine whether its behaviour conforms to the specification. Software testing is today the principal validation activity in industrial context to increase the confidence in the quality of software, even if *Program testing can be used to show the presence of bugs, but never to show their absence !* [Dij70]. Indeed, due to combinatorial explosion of reachable states, exhaustive testing is infeasible in practice.

A strategy to manage this explosion and to keep a relevant quality assessment concerns specification testing assumptions: the input domain of the system can be partitioned into different classes that behave equivalent in terms of observed outputs [OB88, DF93]. Coverage criteria can be applied to specify which test cases shall be selected [OXL99]. To achieve such test selection, Model-Based techniques are relevant because they allow to automatically apply coverage criteria on the abstract behavioral model of the system under test. Moreover, reasoning on models makes it possible to automatically derive test cases (performing stimuli and expected outputs) and offers the additional benefit of providing a structured and repeatable process [ZB09].

Model-Based Testing [UL06] thus takes as input a specification of the system under test and generates automatically test cases and executable test scripts on the basis of model coverage criteria. The specification takes the shape of a behavioral model, allowing the Model-Based test generator both to determine relevant contexts of execution, and to predict the expected system behavior.

In this paper, we propose to apply this approach using a subset of the SysML modeling language [FMS09b] as specification language. This original approach consists in adapting the existing Model-Based Testing process [BGLP08] embedded into the Smartesting Test Designer^{TM} tool [Sma09]. This technology, enables automated test generation from a model written with a subset of the UML modeling language [RJB04] and OCL constraints [WK96]. This UML/OCL fragment is called UML4MBT [BGL$^+$07]. This tool is currently deployed in domains such as Enterprise IT and electronic transaction applications.

Supporting SysML model as input of this process makes it possible to re-use (and adapt) existing test generation techniques to target the specific needs of testing critical embedded systems. Even if SysML is a recent modeling language, it is indeed on the rise in embedded system domain and some studies already use it to develop new industrial validation approaches (e.g. Model Checking and test of on-board space applications [FMS09a]).

To achieve Model-Based Testing using SysML, a dedicated metamodel has been developed specifying which concepts of SysML are truly relevant for the test generation process. This subset of the SysML language is called SysML4MBT. The aim of the paper is also to define the SysML4MBT notation and to propose rewriting rules to translate SysML4MBT into UML4MBT test models.

The paper is organized as follows: Section 2 gives an overview of the Model-Based Testing process associated with Smartesting Test Designer^{TM}, and introduces the features to support SysML4MBT model as input of the method. Section 3 characterizes the subset of SysML notation, called SysML4MBT, used to specify such input model. Section 4 describes the rewriting rules used to automatically derive UML4MBT from SysML4MBT. Section 5 presents and discusses some case-study results.

2 Overview of Model-Based Testing process

The Smartesting solution [BBC$^+$06] is a tooled testing approach to generate and manage functional tests from behavioral models specified with a subset of UML/OCL notation (UML4MBT). The main principles of this solution are:

- Modeling of UML4MBT behavioral model. The model is an abstraction of the system under test.
- Automated generation of abstract test cases by covering each behaviour of the UML4MBT model (using theorem prover technology).
- Generation of executable scripts to automate the test execution on the system under test.

This test generation toolchain is based on a UML modeling tool for the specification of the system under test, and the test engine Smartesting Test Designer^{TM} for the generation of the test cases. The automation of this toolchain is performed through the definition and the use of a dedicated metamodel as interface file, that allows the user to carry out each task in a continuous way. Figure 1 describes this processing (black arrows).

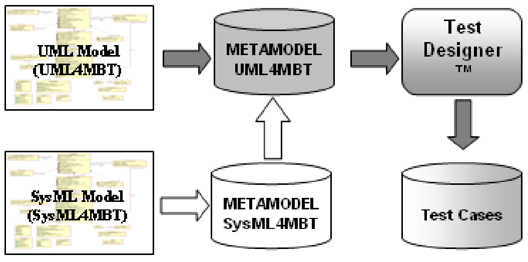

Figure 1: Toolchain.

To extend the input notation with a subset of the SysML notation, a new metamodel dedicated to SysML4MBT has been implemented. The proposed approach also consists in automatically transforming the elements of the SysML4MBT metamodel into elements of the UML4MBT metamodel. This approach makes it possible to re-use the test generation techniques initially developed for UML4MBT models. This adding processing to address SysML model is illustrated by the white arrows in Figure 1.

The next section introduces the UML4MBT and SysML4MBT notations, respectively subsets of the UML and SysML languages.

3 UML4MBT and SysML4MBT

UML4MBT is the subset of UML notation used by Test Designer^{TM}. The goal of this subset is to offer precise, necessary and sufficient modeling features to design behavioural models for test generation purpose. The proposed subset for Model-Based Testing is based on three UML diagrams: class diagrams (to model the points of control and observation of the SUT), object diagrams (to define test data) and statemachines. OCL is used to model dynamic behaviour of the system under test constraints. To be able to execute transition actions and operation postconditions, UML4MBT uses an operational interpretation of OCL expressions.

Like UML4MBT, SysML4MBT defines restrictions on SysML to design comprehensive, precise and interpretable models to specify embedded system behaviors for test generation purpose. This section introduces UML4MBT notation and definition of SysML4MBT restrictions.

3.1 UML4MBT

A UML4MBT model contains:

- One UML class diagram to represent the static view of the model. It describes the abstract entities of the system and their dependencies. The available UML elements are classes, associations, enumerations, class attributes and operations.
- One UML object diagram to list the concrete objects used to compute test cases and to define the initial state of the model. The object diagram must be an instantiation of the associated class diagram. All objects used to describe the life cycle of the system have to be defined in the object diagram. The dynamic creation (resp. deletion) of entities in the concrete system is simulated by creation (resp. deletion) of links between objects of the UML model.

The dynamic view can be modeled in two manners:

- by insertion of OCL expressions on pre or postcondition of operations (which are inserted in classes).
- by creation of one statemachine diagram (annotated with OCL constraints). This one has also some restrictions. For example, it cannot contain parallel states, historic states, fork and join states or trans-hierarchical transitions.

OCL is thus used in class diagrams to formalize the expected behavior of class operations. It is also used within statemachines to formalize transitions: guards and effects of transitions are expressed as OCL predicates. To be able to execute OCL annotations, UML4MBT allows an operational interpretation of OCL expressions. For example, the OCL expression $self.attribute = true$ can be used in two different contexts: a passive and an active context. A passive context expresses constraints, while an active context expresses state changes in the model.

3.2 SysML4MBT

The notation SysML4MBT defines the subgroup of elements that are taken into account in SysML. A SysML4MBT model contains:

- One Block Definition Diagram to represent the static structure of the system and its environment. This diagram can contain signals and ports, which represent communications between blocks.
- One Internal Block Diagram to specify the networking between all blocks. Connectors, which link ports, can represent electrical or mechanical communications.

- One or more statemachine diagrams, annotated with OCL constraints, specifying, in a formal way, the dynamic aspect of the system.
- One requirement diagram, which permits to express functional requirements of the system and link them with the related models elements.

SysML4MBT statemachine can contain much more elements than UML4MBT statemachine can. Then, fork/join, historic and parallel states are allowed. About OCL, the circumflex(^) is added in OCL constraints for SysML4MBT model. This element enables to send signal to an other block.

4 Transformation from SysML to UML models

In the toolchain, we target to specify the system with a SysML4MBT model and to use it as input of Test Designer^{TM} Model-Based Testing process. Test Designer^{TM} can only evaluate UML4MBT model, therefore we suggest to translate SysML4MBT model into UML4MBT model.

SysML being defined as a UML profile, the simple way to translate SysML4MBT model into UML4MBT consists to leave out the SysML stereotype, which denotes a simple transformation. Model transformation is indeed a widespread approach in Model-Driven Engineering domain [SK03] to achieve for example model verification [ALL09] or code generation [Old04]. This solution is thus adopted to translate the following concepts:

- Block Definition Diagram elements,
- sequential elements of statemachine diagram.

However, the following elements are allowed in SysML4MBT but have no corresponding element in UML4MBT:

- signal on Block Definition Diagram,
- Internal Block Diagram,
- signal send (^) and signal receive (transition with signal receiving trigger) in SysML4MBT statemachines,
- all parallel elements of SysML4MBT statemachines (fork, join, parallel states and multiple statemachines),
- historic states in SysML4MBT statemachines.

For those elements, rewriting rules have to be applied instead of a simple transformation. In the same way, some elements are compulsory in UML4MBT models and do not exist on SysML4MBT. For example, a UML4MBT object diagram must be defined from the SysML4MBT model.

4.1 From Block to Class

This section presents the transformation of the SysML Block Definition Diagram (BDD) into a UML Class Diagram. Each element allowed in SysML4MBT Block Diagram has an equivalent in UML4MBT. Indeed, a block becomes a class and associations, compositions, operations and attributes exist in both metamodel.

∼ ***Example:***
The SysML4MBT diagram of the figure 2 is rewritten by UML4MBT diagram depicted in figure 3.

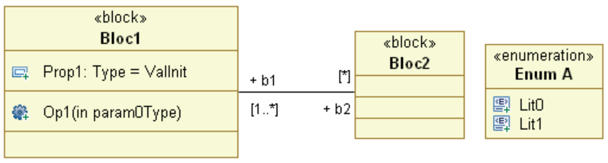

Figure 2: Example of SysML4MBT Block Definition Diagram.

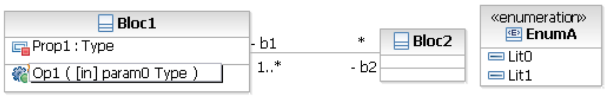

Figure 3: Representation of the BDD in figure 2 using UML4MBT class diagram.

4.2 Internal Block Diagram

Ports and signals, which are used in Internal Block Diagram (IBD), have to be rewritten in UML4MBT.

4.2.1 Signals

In SysML, signals are defined with the BDD and used in IBD. Each signal defined on BDD becomes a class of the UML class diagram. For a given signal, a class with the same name is created and attributes of the signal become attributes of the class. It is also necessary to add an attribute in each class. This new boolean attribute named $isUsed$, initialized to $false$, indicates whether the corresponding signal has been sent or not.

∼ ***Example:***
Figure 5 represents the transformation of the signals introduced in the figure 4.

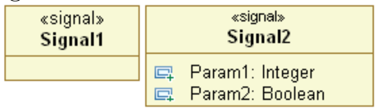

Figure 4: Example of SysML4MBT signal on BDD.

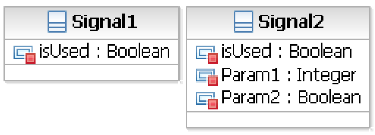

Figure 5: Signal representation using UML4MBT.

4.2.2 Ports

The only information about ports that has to be preserved in UML is which signals are expected on which ports. Then, an association between the class representing the block, which host the port, and the class representing the signal, which can be received by the port, is added. Each association is named with the concatenation of the name of the port, the name of the signal and finally *_trans*. The signal role is named like the transition (without *_trans*) and has for multiplicity $*$. The other side takes the multiplicity 0..1 and an arbitrary name is given as role.

∼ ***Example:***
A port p of a block B, which can receive signal $Sig1$ or $Sig2$, is translated into UML as shown in figure 6.

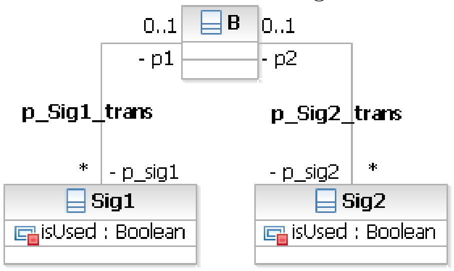

Figure 6: UML4MBT representation of the port p of the block B, which can receive signals $Sig1$ and $Sig2$.

4.3 StateMachine diagrams

The major part of the model transformation concerns the SysML4MBT statemachine diagrams to a UML4MBT statemachine diagram. A lot of elements are present on both metamodels, and also do not need to be rewritten:

- initial state,
- final state,
- composite state,
- all transitions triggered by a receive of operation call,
- all transitions without trigger,
- onEntry and onExit on states.

For all other following elements, rewriting rules have to be applied:

- all transitions triggered by a signal receive,
- historic states (standard or deep),
- parallel states,
- fork and join states.

Moreover, in SysML4MBT, it is possible to have more than one statemachine diagram. Thus, it is necessary to merge all statemachines to obtain a unique UML4MBT statemachine diagram. Finally, about OCL, circumflex (^) is the unique OCL element added to SysML4MBT in comparison with UML4MBT.

4.3.1 OCL expression ^

The rewriting of the OCL expression including the signal send (^) is realized by instantiation of class diagram associations.

- those associations represent the fact that signals have been sending and are still pending for reception.
- the sending of a signal in OCL is written: *"block"."port"^"signal"("parameters")* where:
  - *block* is the path to the block which hosts the port
  - *port* is the name of the port which is the receiver of the signal
  - *signal* is the name of the signal which is sent
  - *parameters* are the values of all the attributes of the signal

Then, the translation of this expression is an association that:

- starts from the instantiation of the class, which represents the receiver block of the signal,
- ends on an unused instantiation of the class that represents the sent signal (whose parameter $isUsed$ is equal to false),
- with role on side *signal*:
  "Name of port"_"Name of signal".

∼ ***Example:***
The OCL action in SysML4MBT:

$$block.port^{\wedge}signal(val1, val2)$$

becomes in UML4MBT

$let\ s = signal.allInstances() \rightarrow any(isUsed = false)\ in($
$s.Param1 = Val1\ and\ s.Param2 = Val2\ and$
$s.isUsed = true\ and$
$block.port\_signal \rightarrow includes(s))$

4.3.2 Signal receive

The received signal is modeled in SysML by a trigger with a port and a signal attached to a transition. In UML4MBT, it is transcribed into a dedicated transition without trigger but with a specific guard. This one verifies if the expected signal is pending on the port.

A signal receive trigger with those features:

- receive port = *port*
- block associated to the port = *block*
- received signal = *signal*

becomes in UML4MBT:

$$[block.port\_signal \rightarrow notEmpty()]$$

which is the guard of the dedicated transition. This expression verifies the existence of a link which represents a pending signal in the port.

In the action of such a transition, the pending signal is released using the following OCL expression:

$let\ s =$
$block.port\_signal.allInstances() \rightarrow any(true)\ in($
$s.isUsed = false\ and$
$block.port\_signal \rightarrow excludes(s))$

4.3.3 Composite state

To delete a composite state, it must not contain fork, join, parallel or historic state. If it is the case, they have to be deleted before.

To delete a composite state, the following process is performed:

- All transitions, which arrive on composite state, are relocated to arrive on the sub-state that is pointed by the initial state of the composite state.
- The action of the transition leaving the initial state is duplicated to each of those transitions and the initial state is delete.
- All transitions, which leave the composite state, are duplicated on each state of the composite state.
- Then the composite state can be deleted.

∼ ***Example:***
The figure 8 depicts the UML4MBT diagram obtained from the SysML4MBT composite state of the figure 7.

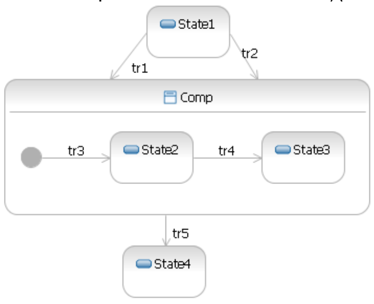

Figure 7: Example of SysML4MBT composite state.

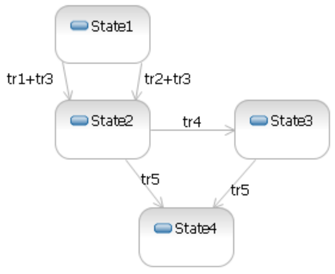

Figure 8: UML4MBT diagram from the composite state of the figure 7.

4.3.4 Historic state

An historic state represents the most recent active sub-state of its containing state. To rewrite an historic state, multiple stages are needed.

First, for each historic state, a variable, which represents the memory, is created in a dedicated class. It permits to identify the previous active state in the composite state. This new attribute is named like the historic state and is typed integer. Its default value is 0. An other variable named $CurrentHist$ and initialized to 0 is needed to identify the possible current active historic state.

Next, we number each historic state (we associate each historic state with a different integer). Independently, we number each state of all the composite states containing an historic state.

∼ ***Example:***
The figure 9 represent the numbering of a statemachine.

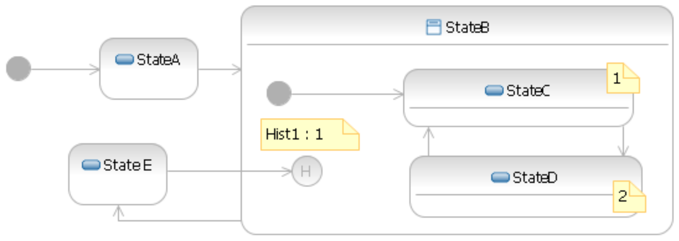

Figure 9: Example of a numbering statemachine containing historic state.

Each transition, which points to an historic state, is transformed to point to the composite state and this effect is added:
$self.CurrentHist = "number\ of\ the\ historic\ state"$.
On the composite state, the transition, which leaves the initial state, is replaced by an empty transition, which points to a new choice state. The old initial transition starts to this new choice state. It is the *else* transition of the choice state. For each state of the composite state, are added:

- the onEntry OCL expression:
  $"HistName" = "NumState"$
- a transition, which starts from the choice state and arrived in the state. This transition has the following configuration:
  - no trigger
  - the guard: $CurrentHist = "HistNum"$ and $"HistName" = "NumState"$
  - the effect: $CurrentHist = 0$

where:

- $CurrentHist$ is the variable created previously,
- $"HistNum"$ is the number of the Historic State during the current transformation,
- $"HistName"$ represents the variable that replace the historic state we are rewriting,
- $"NumState"$ represents the number (defined during the classification) of the destination state of the transition.

∼ ***Example:***
The diagram of the figure 9 is translated into the diagram of the figure 10.

4.3.5 Deep historic state

For the rewriting of the deep history, sub-composite states are firstly deleted and deep historic state is next rewriting like a standard historic state.

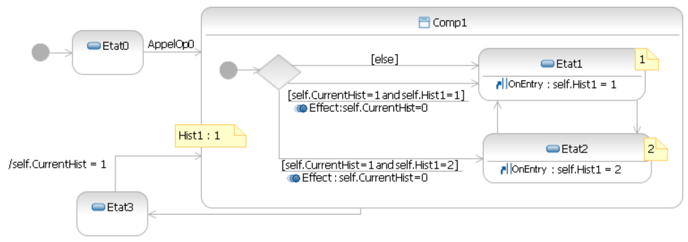
Comp1
Etat0
AppelOp0
[else]
Etat1
1
OnEntry : self.Hist1 = 1
[self.CurrentHist=1 and self.Hist1=1]
Effect:self.CurrentHist=0
Hist1 : 1
/self.CurrentHist = 1
Etat2
2
OnEntry : self.Hist1 = 2
[self.CurrentHist=1 and self.Hist1=2]
Effect : self.CurrentHist=0
Etat3

Figure 10: UML4MBT state machine obtained from the historic state of the figure 9.

4.3.6 Fork and join

Fork and join states are transformed on parallel states. For each couple fork/join state:

- For each transition, which leaves the fork, an initial state, which becomes the start of the transition, is created.
- For each transition, which arrives on the join, a final state, which becomes the destination of the transition, is created.
- A new parallel state is created.
- All transitions arriving on the fork, point to the parallel state and all transitions leaving the join, are moved to the new parallel state.
- Each path from new initial state to new final state is moved to the relating region of the parallel state.

∼ ***Example:***
For example, the fork/join state of the figure 11 becomes the parallel state of the figure 12.

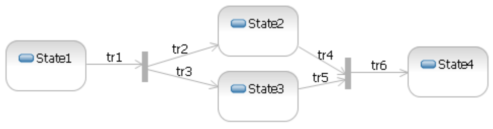

Figure 11: Example of fork/join state.

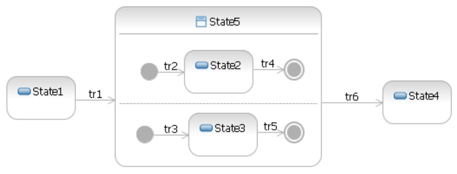

Figure 12: UML4MBT representation for fork/join state of the figure 11.

Following these transformation steps, the parallel state is rewritten as shown in the next section.

4.3.7 Parallel state

The same stage is performed to merge parallel states and parallel statemachines. Each region of the parallel state is considered as a statemachine. The applied strategy is now explained.

4.3.8 Parallelism

In SysML4MBT, more than one statemachine diagram can be specified in opposition to UML4MBT. Those diagrams are evolve in parallel manner. Then, it is necessary to merge them to obtain only one UML4MBT statemachine diagram. It is important to remind that those diagrams communicate by sending signals.

The sequence of operations, which makes it possible to merge multiple statemachine diagrams, is the following:

1. Cleaning of all complex states (fork, join, composite, parallel and historic) of all statemachines.
2. Providing a Cartesian product of all the statemachines.
3. A transition is drawn only if it exists one path to reach the start state.
4. All transitions triggered with an operation call receive and all transitions without trigger are drawn.
5. if a transition sends signals, the information that signals are pending is stocked in the state.
6. then transitions triggered with a signal receive are drawn only if this signal is pending on the root of the transition.
7. if a transition without trigger leaves a state with pending signal, the pending signals and the signals sent by the transition are duplicated into the target state.
8. also, if a state is pending two signals in the same time, and if a transition receives one of the both, the second is duplicated in the target state with signals sent by this transition.

This process leads to create a unique UML4MBT statemachine diagram.

4.4 Object Diagram creation

To have a correct UML4MBT model, it is necessary to have an object diagram. Since it does not exist in SysML, it is fully built in the following way :

- Each class of the new class diagram is valuated by one instance in the object diagram.
- Associations are instantiated using the minimum number of links, with respect to lower multiplicities.
- The Cartesian product of statemachine diagrams permit to know how many time each signal can be pending at the same time. Each class, which represents signal, is instantiated according to this number.

4.5 Requirement diagram

The requirement diagram permits to associate requirements with model elements that satisfies them. In UML4MBT, OCL expression can be used to annotate requirements. The syntax is:

$$/**@REQ : "text\ of\ the\ requirement" */$$

This expression in a UML4MBT OCL expression defines that the requirement identified by the text is satisfied by nearly OCL expression. Then, all the behavioral requirements defined in SysML4MBT requirements diagram, satisfied by transition, operation and *onEntry/onExit* expression, are translated respectively into effect of transition, postcondition of operation and *onEntry/onExit* expression as OCL annotations.

4.6 Implementation

All those rewriting rules have to be executed in a specific order described by the following algorithm:

1. The Block Definition Diagram and the Internal Block Diagram are transformed in class diagram.
2. Signal sends receive are translated.
3. Requirement diagram is rewritten.
4. All fork/join states of the statemachine diagrams are transformed on parallel state.
5. Each composite, historic and parallel states is rewritten by hierarchical stage. That means that we rewrite those elements only if they do not contain others. For example, if a composite state contains an historic state, the historic state is rewritten first and the composite state next.
6. Parallel statemachines are merged.
7. The object diagram is build.

Models are stored in JAVA structures that are based on UML and SysML metamodels (the JAVA structure has been developed according to UML4MBT and SysML4MBT metamodels). This algorithm is implemented with JAVA language. Test Designer^{TM} plug-in is fully developed with JAVA, so this choice is a key point to maintain the code homogeneity of the overall project, and facilitates its maintenance. However, it would be possible to use other technologies like ATL [JK06] to implement this algorithm.

5 Case Study

We have applied those rewriting rules on 3 case studies:

- Front lightings is the representation of the lightings system of the front of a car. This system permits to light on and light off independently headlights and highlights of the car. To activate it, a control lever is used.
- Steering, which is the representation of the steering column of a car. The goal of this example is to observe reaction of the steering column of a car with some road plot. Then, we model the road and it communicates with the column by the tyres and the wheel.
- FrontWiper is a specification of a the wiper system of the front of a car. The modeled functionalities are low speed drying up, high speed drying up, intermittently speed drying up and cleaning with drying up.

Table 1 presents results obtained on those case studies.

		Lightings	Steering	Wiper
SysML	Blocks	6	9	15
	Connectors	4	10	18
	SM	5	6	12
	States	(2,2,2,2,4)	(2,2,2,2,2,2)	(1,1,1,1,1,2, 17,10,2,2,2,2)
	Transitions	(3,3,3,3,9)	(3,3,3,3,2,8)	(3,4,3,5,2,4, 53,17,3,3,3,3)
UML	Classes	10	16	29
	Objects	15	20	57
	States	64	18	2526
	Transitions	256	123	31873

Table 1: Results of case studies.

Lines *states* and *transitions* for SysML model represent the number of states and transitions for each statemachine. We can see that Frontlightings SysML model is smaller than Steering SysML model but, after transformation (UML box), FrontLightings has more states and transitions than Steering. This is due to restrictions on signals transmissions (more limitations on Steering SysML model).

About generation time, we notice that this algorithm is efficient for small model: approximatively 3 seconds for FrontLightings or Steering. However, if the model is too heavy, the generation time explodes: more than 30 minutes for FrontWiper. To decrease this generation time, it would be necessary to change the strategy of the translation: the FrontWiper model becomes very big in UML format because our algorithm to reduce parallel statemachine into a single UML one is based on the well-known costly Cartesian product.

6 Concluding remarks and future works

This paper introduced a subset of SysML for Model-Based Testing, and proposed rewriting rules to translate a SysML4MBT model into an equivalent UML4MBT model. This transformation made it possible to generate test cases from embedded system SysML specification using Smartesting Test Designer^{TM}. Three case-studies show the feasibility and the relevance of this approach to generate test cases dedicated to critical embedded systems. However, some problems relating to scalability have been underlined. It is due to the transformation rules, based on a Cartesian product, of parallel SysML statemachine into a single UML statemachine.

Future works may consist in addressing the following issues:

- to increase scalability by improving the rewriting rules,
- to increase expressiveness of UML4MBT notation to natively support parallel statemachine diagram (in this way, the complexity would be managed on-the-fly during the test cases calculation),
- to increase model coverage by setting up new test generation strategies.

Acknowledgments Part of this work has been supported by the French project VETESS [VET10].

References

[ALL09] M. Asztalos, L. Lengyel, and T. Levendovszky. A formalism for describing modeling transformations for verification. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa'09)*, pages 1–10, Denver, Colorado, 2009. ACM Press.

[BBC+06] 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 International Workshop on Model-based Testing (MBT'2006)*, volume 94 of *LNCS*, pages 223–230, Dresden, Germany, October 2006. Springer Verlag.

[BGL+07] 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 3rd International Workshop on Advances in Model Based Testing (A-MOST'07)*, pages 95–104, London, UK, July 2007. ACM Press.

[BGLP08] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux. A test generation solution to automate software testing. In *Proceedings of the 3rd International Workshop on Automation of Software Test (AST'08)*, pages 45–48, Leipzig, Germany, May 2008. ACM Press.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the International Conference on Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.

[Dij70] E.W. Dijkstra. Notes on structured programming. Technical Report EWD249, Eindhoven University of Technology, 1970.

[FMS09a] J.M. Faria, S. Mahomad, and N. Silva. Tactical results from the application of model checking and test generation from uml/sysml model of on-board space applications. In *Proceedings of the International Conference on DAta Systems In Aerospace (DASIA'09)*, Istanbul, Turkey, May 2009. ESA Press. ESA SP-669.

[FMS09b] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann OMG Press, 2009. ISBN 978 0 12 374379 4.

[JK06] F. Jouault and I. Kurtev. *Transforming Models with ATL*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin / Heidelberg, 2006.

[OB88] T.J. Ostrand and M.J. Balcer. The Category-Partition Method for Specifying and Generation Functional Test. *Proceedings of the ACM Conference*, 31(6):676–686, June 1988.

[Old04] J. Oldevik. *UMT: UML Model Transformation Tool.* SINTEF Information and Communication Technology, Oslo, Norway, March 2004. http://umt-qvt.sourceforge.net/.

[OXL99] A.J. Offut, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5th International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las-Vegas, USA, October 1999. IEEE Computer Society Press.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 2th edition, 2004. ISBN 0 321 24562 8.

[SK03] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Journal of Software*, 20:42–45, October 2003.

[Sma09] The Smartesting web site. `http://www.smartesting.com`, 2009.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach.* Elsevier Science, 2006. ISBN 0 12 372501 1.

[VET10] The VETESS web site. `http://lifc.univ-fcomte.fr/VETESS/`, 2010.

[WK96] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1996. ISBN 0 201 37940 6.

[ZB09] H. Zhu and F. Belli. Advancing test automation technology to meet the challenges of model-based software testing. *Journal of Information and Software Technology*, 51(11):1485–1486, 2009.