Generative Technologies for Model Animation in the TOPCASED Platform

Xavier Crégut[1], Benoît Combemale[2], Marc Pantel[1],
Raphaël Faudoux[3], and Jonatas Pavei[1,4]

[1] Université de Toulouse, IRIT - France,
Firstname.Lastname@enseeiht.fr.
[2] Université de Rennes 1, IRISA, France,
Firstname.Lastname@irisa.fr.
[3] ATOS Origin, Toulouse - France
Firstname.Lastname@atosorigin.com.
[4] Universidade Federal de Santa Catarina - Brazil

Abstract. Domain Specific Modeling Languages (DSML) are more and more used to handle high level concepts, and thus bring complex software development under control. The increasingly recurring definition of new languages raises the problem of the definition of support tools such as editor, simulator, compiler, etc. In this paper we propose generative technologies that have been designed to ease the development of model animation tools inside the TOPCASED platform. These tools rely on the automatically generated graphical editors of TOPCASED and provide additional generators for building model animator graphical interface. We also rely on an architecture for executable metamodel (i.e., the TOPCASED model execution metamodeling pattern) to bind the behavioral semantics of the modeling language. These tools were designed in a pragmatic manner by abstracting the various model animators that had been hand-coded in the TOPCASED project, and then validated by refactoring these animators.
Key words: Generative technologies, Model animation, Model execution, Metamodeling pattern

1 Introduction

Model Driven Engineering plays a key role in the development of safety critical systems by providing early Validation & Verification (V&V) activities for generic and domain specific models. It is thus mandatory to be able to build easily V&V tools dedicated to each Domain Specific Modeling Language (DSML). We will present in this paper some experiments conducted in the TOPCASED project on the use of generative technologies for graphical model animation tools. In order to design these technologies, we first defined a generic framework for implementing the model execution engines [1], this framework relies, on the one hand, on a metamodeling pattern [2] that extends the classical language definition metamodel with execution related metatypes and attributes; and on the other hand on a generic "discrete event" execution engine. The handling of each discrete event is represented as an endogenous model transformation (i.e., with the same source and target metamodel) that modifies only the attributes of

the execution specific metatypes. This framework has been used to implement several model animators in TOPCASED using JAVA and SMARTQVT as transformation languages. Then, these animators were refactored in order to make commonalities explicit, and generation patterns were proposed in order to ease their development. In a last step, the specific parts of the animators were extracted from the first versions and integrated in the generated ones in order to validate our proposal.

Our presentation will rely on the SIMPLEPDL DSML, a toy process description language derived from SPEM. SIMPLEPDL has been designed in TOPCASED as a simple yet representative use case for teaching and experimenting MDE generative technologies for V&V [3,4]. These technologies have been validated through complete use cases in TOPCASED such as SYSML/UML state machine and SAM[5] model animators.

We will first present the TOPCASED project, the SIMPLEPDL use case and the requirements expressed for model animators by industrial end users for the design of safety critical systems. Then the contributions of this paper are the key facts about the framework for model animation in TOPCASED that relies on: a) a metamodeling pattern for expressing model execution related metatypes; b) a discrete event system execution kernel and c) the use of endogenous model transformation for defining the handling of a specific event in the execution of a model. Then we show how generative technologies were introduced in order to factor out the common parts of the various model animators that had been hand-coded in the first versions of TOPCASED. Finally, we conclude on the current state of our experiments, and we give insights on our future work.

2 The TOPCASED toolkit

2.1 The TOPCASED project

TOPCASED[6] (*Toolkit In OPen source for Critical Applications & SystEms Development*) [5] is a project started in 2005 from the French "Aerospace Valley" cluster, dedicated to aeronautics, automotive and space embedded systems. TOPCASED aims at defining and developing an open-source, Eclipse-based, modular and generic CASE environment. It provides methods and tools for the developments of safety critical embedded systems. Such developments will range from system and architecture specifications to software and hardware implementation through equipment definition.

In this purpose, TOPCASED provides both domain specific (as SAM) and general purpose modeling languages (such as SYSML/UML, AADL, EAST-ADL, SDL, etc.) and associated tools like graphical and text editors, documentation and code generators, validation through model animation, verification through model checking, version management, traceability, etc.

TOPCASED relies on Model Driven Engineering (MDE) generative technologies to build all these tools for all these languages. It is thus an MDE platform both for building system models and for building the platform itself. MDE technologies used in TOPCASED for defining and tooling languages are centered around Ecore[7] and configuration models taken as inputs by generative tools (e.g. graphical editor generator).

[5] SAM is a DSML used at Airbus in the A350WB program for specifying inter-system interfaces, mode automata and early design level protocols.

[6] `http://www.topcased.org`

[7] Ecore is the metalanguage of Eclipse Modeling Framework, `www.eclipse.org/modeling/emf`

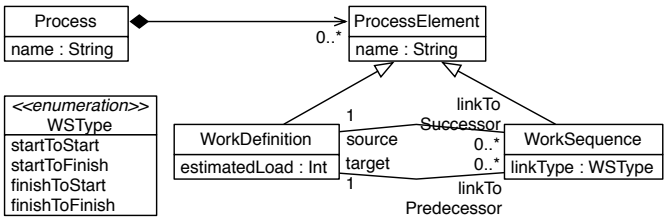

Fig. 1. Domain Definition Metamodel (DDMM) of SimplePDL

2.2 Use case

To illustrate the MDE approach used in TOPCASED, we rely on a simplified process description language called SimplePDL. SimplePDL is deliberately simplified to avoid overloading this presentation with useless details.

A metamodel is used to define the concepts (metaclasses) of the domain addressed by the DSML and the relationships between them (references). We call it the *Domain Definition MetaModel*, *DDMM*. The DDMM of SimplePDL is shown on figure 1. It defines the concepts of process (*Process*) composed of process elements (*ProcessElement*) that can be either a work definition (*WorkDefinition*) or a work sequence (*WorkSequence*). Work definitions are the activities that must be performed during the process. A work sequence defines a dependency relationship between two work definitions. The second work definition can be started or finished only when the first one is already started or finished according to the value of the attribute *linkType*.

A metamodel defines only an abstract syntax which is not adequate for human beings. Graphical concrete syntaxes are often a better way to create and manipulate DSML models. TOPCASED provides a graphical editor generator based on the description of the desired editor. Fig. 2 shows the generated SimplePDL graphical editor. For SimplePDL, one has to define how the SimplePDL concepts are graphically presented (as vertices or arcs) and to explain how to update the SimplePDL model when the graphical elements are created or changed.

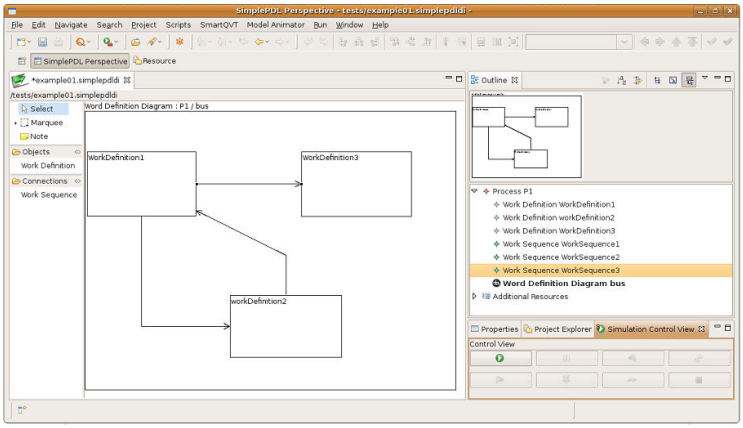

Fig. 2. SimplePDL Graphical Editor Generated with TOPCASED

2.3 Motivations for model animators

The use of DSML allows to introduce early V&V activities at the modeling stage, long before the end product is developped. Here are some key aspects extracted from TOPCASED end users requirements for model animation :

- Modeling is an error prone activity. A model animator allows the designer to check that the produced model really expresses the expected behavior. This is a validation that the model is a correct rendering of what the designer had in mind.
- The system design team needs to check that the proposed system fits the end user needs. A model animator allows to organise demonstrations of the system behavior connected to realistic system Human Machine Interface (HMI) and thus add the user in the loop. This is a validation that the specification is a correct rendering of the end user needs.
- Early models are usually an approximation of the final system that does not fit all the requirements, it is thus difficult to use exhaustive model verification tools that will detect a large number of errors that are due to the incomplete nature of these early models. Model animation allows to design interactively the verification scenario that fits the current state of the models. This is a partial verification of the appropriate parts of the model with respect to some specific requirements.
- Many behavioral verification tools rely on the semantics of the DSML. This semantics is usually specified by the DSML designer based on informal end users needs. A model animator allows the end user to play with this semantics and check that it really fulfills its needs. It is thus a good tool to validate the DSML definition.

3 Model Execution in the TopCased Toolkit

The TOPCASED toolkit targets executable DSML. Thus, it must provide means to define their executable semantics. A metamodeling pattern has been defined to capture all the data required to execute a model and a framework is provided based on this pattern. Model execution is then the core building block to add animation facilities for a DSML.

3.1 A metamodeling pattern for model execution

When we want to simulate a process, that is to execute one of its models, we first have to understand and define what are the interactions between the model and its environment. For SimplePDL, it is possible to start or finish a work definition or record the load already spent on a work definition. The user may also want to specify the meaning of a precedence constraint and thus add a threshold on a work sequence that could be changed during process enactment. All these interactions are external events generated by the environment that will induce changes on the model itself. These external events are captured in the *Event Definition MetaModel*, *EDMM* (Fig. 3). An event has several attributes. For example, the event "*start a work definition*" takes the targeted work definition as attribute. The event "*increase work load*" has two attributes, the targeted work definition and the load increment.

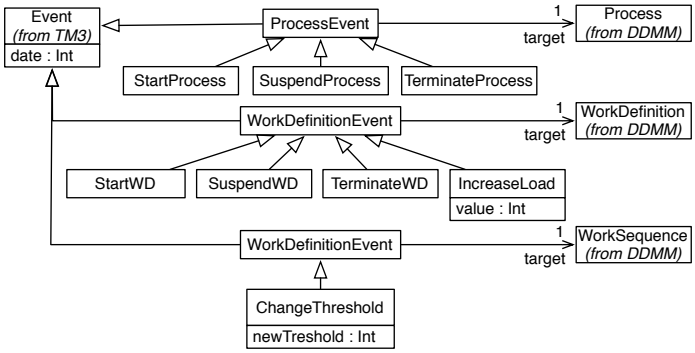

Fig. 3. One possible Event Definition MetaModel (EDMM) for SimplePDL

The next step is to define scenarios as a set of ordered external events. They may be defined interactively during the animation itself or before the execution starts (batch simulation). A scenario is used to drive the execution. Starting from a model and a scenario, the execution produces an output trace of all the events that occurred during the execution (including external events and also possible internal events triggered by the handling of the other events by the semantics). Traces and scenarios are defined in the *Trace Management MetaModel* (*TM3*, Fig. 4).

The concepts captured in the *DDMM* are not sufficient to animate a model. For instance, some events may require additional informations. For example, the event "*increase work load*" means that the load of a work definition has to be recorded. We also have to be able to decide whether an activity can be started or not. So we have to know the state of its preceding activities. Thus, we propose to define a *State Definition MetaModel SDMM* that captures all the data required during an execution. On Fig. 5, new data have been directly added on the *DDMM*: *loadConsumed* and *state* on *WorkDefinition*, *state* on *Process* and *threshold* on *WorkSequence*. In fact they are defined in the SDMM that "completes" the DDMM. This may be achieved using the *merge* operator defined in MOF specification [6]. So, several SDMM may be defined for the same DDMM, each corresponding to a different execution semantics.

The four previous metamodels DDMM, SDMM, EDMM and TM3 constitute the architectural part of the metamodelling pattern that we have implemented in TOPCASED to support model execution. The execution semantics is still lacking. Its purpose is to define how the SDMM model evolves when a concrete event from the EDMM model occurs. Before describing that aspect, we present the TOPCASED framework for model execution based on this pattern.

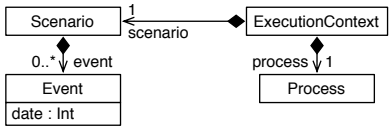

Fig. 4. Trace Management MetaModel (TM3)

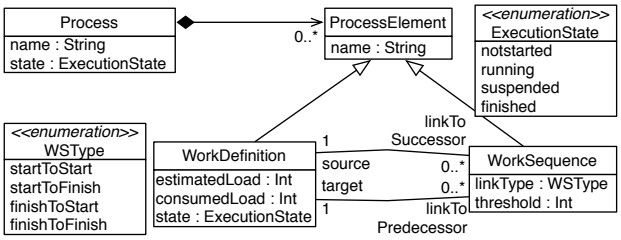

Fig. 5. State Definition MetaModel (SDMM) merged with the DDMM of SIMPLEPDL

3.2 The TOPCASED Framework for Model Execution

TOPCASED provides a framework for model execution based on the previous metamodelling pattern. It is composed of a generic core — independent of any DSML — (top of Fig. 6) that has to be specialized for a given DSML (bottom of Fig. 6).

The framework only depends on the runtime events defined for the DSML in the EDMM. The execution engine is built from three main components: Agenda, Driver and Interpreter. The first two components implement a discrete event computation model, based on the elements in the TM3 metamodel. They are thus generic and independent of any particular executable modelling language. The Agenda stores the runtime events corresponding to one particular execution. Runtime events are ordered according to their occurring date. At the beginning of the execution, the Agenda instance is initialized with all the events contained in the scenario to be run. The Agenda provides the API required by the Driver to handle the events (e.g., to establish the next runtime event, to add a new runtime event).

The Driver controls the execution. It constitutes the interface with external components (mainly the Control Panel) thanks to a dedicated API, which allows both batch and interactive execution. Its step method consists in getting the next runtime event from the agenda and asking the Interpreter to handle it. The generated endogenous runtime events are then stored in the agenda.

Finally, the Interpreter abstracts the different possible semantics of the pluggable executable modelling language. Its run method interprets runtime events, updates the dynamic information of the model, and returns the list of generated endogenous runtime events. Obviously, the Interpreter is specific to the modelling language and supports its own semantics. It thus has to be specialized for the considered DSML.

The previous classes are the core of the framework. A generic user interface is developed on top of it. It is composed of an interactive *Control Panel* that emulates the environment and allows to add new runtime events into the Agenda. The SDMM model is displayed on the graphical visualization developed for the graphical editors. It is based on the notification facilities provided by the underlying EMF framework and the Adapter pattern to change the color of the graphical elements. These changes of graphical properties were sufficient to animate SYSML/UML state machines or SAM models but had to be enhanced for SIMPLEPDL (see section 4).

3.3 Operational semantics

The last part consists in implementing the *Intepreter* class to define the execution semantics. According to the concrete event received as parameter of the *run* method, one

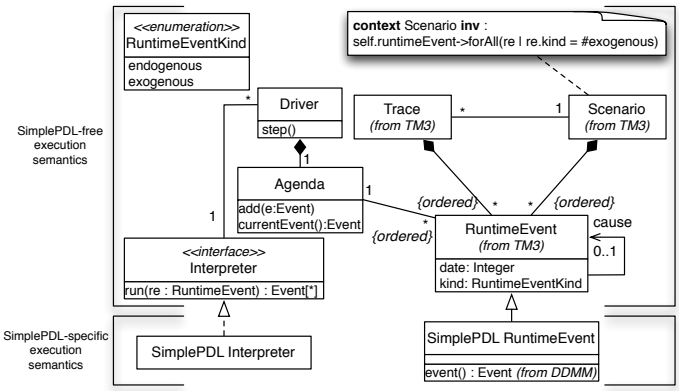

Fig. 6. Framework for Model Execution

has to describe how the SDMM model evolves. In the first version of the TOPCASED animators, the *run* method was hand-coded using JAVA and the EMF API. Then, it has been written using SMARTQVT, an open source transformation language that implements the OMG QVT specification and generates Java code that facilitates its integration in the TOPCASED framework. The main benefit of using SMARTQVT is to ease the navigation on model elements as discussed in the next section.

Based on these technologies, model animators were implemented for SYSML/UML state machines and SAM models. The interactions with the end user and the graphical editors were hand-coded in JAVA. As these languages were quite similar one to the other, we developed a model animator for SIMPLEPDL in order to reveal additional requirements. These implementations were abstracted in order to detect common patterns that could be generated thus leading to the following proposals.

4 Generative tools and extensions to the animator's core

Model animators are composed of three main parts: *semantics*, *controllers* and *animator* which have to be developed for each new DSML. Semantics implements the execution semantics. Controllers allows the user to inject new runtime events. Animator is responsible for allowing the user to inspect the execution. It relies on visualisation tools to display dynamic information and a control panel to drive the animation. In the first versions of TOPCASED model animators, these components were hand-coded. Our purpose for this work was to develop generative tools that accelerate the development of new animators by generating as much as possible parts of these tools. The first results of these experiments are presented here after.

4.1 Multiple semantics definition

In the actual architecture, one has to implement the Interpreter interface and its *run* method to define the DSML execution semantics. The code then generally starts with a big switch that selects and executes the reaction corresponding to the concrete event received as parameter of the *run* method. Furthermore, when writing the code implementing the reaction, one has to access the model. It would be helpful to add new

helper methods on the model (EDMM and SDMM) to facilitate this. As these methods are specific to the semantics being implemented and because several semantics could be implemented for the same DSML, it is not a good idea to pollute the DDMM or SDMM with all those helper methods. A better solution is to implement the Visitor pattern.

We have defined a new plugin called org.topcased.semantics that provides two interfaces (considered as Eclipse extension points). The first one is called Semantics and uses overloading to specify one run method for each possible concrete event defined in the EDMM. The second interface is called Dispatcher and contains one single method dispatch(RuntimeEvent event, Semantics semantics). These two interfaces correspond to the Visitor and Visitable elements of the Visitor pattern. They are independent of any DSML.

The *Dispatcher* interface has only to be implemented once for each DSML. In fact it is generated from the EDMM model using Acceleo[8]. It implements the *accept* method that is normally present on the *Visitable* elements. In doing so, no change is required on the EDMM. The implementation of Dispatcher's dispatch method is a big switch on the concrete type of the runtime event that selects and executes the right run method of the semantics received as parameter. As the class is generated (like the *Semantics* and *Dispatcher* interfaces), there is no risk of missing some cases.

Then, defining a new execution semantics for a given DSML simply consists in implementing the Semantics interface. It is then possible to register several semantics for the same DSML. The user may then choose which one will be executed. For example, it would be possible to define a semantics that only handles start and finish events and a more precise one that also handles increments of work loads and thresholds.

The Visitor pattern could also be generated for the DDMM (and SDMM) models. The semantics defined on SimplePDL is rather easy to implement because most of the events only imply changes on the target element and do not require heavy navigation on the model. Nevertheless, *WorkDefinition*'s start and finish events require to check whether the states of the previous work definitions are consistent with the constraints defined on the corresponding work sequences. If the semantics had been implemented using Java/EMF, it would be useful to write such helper methods as new instances of the *Visitor* interface. But, as we rely on high level transformation languages such as SmartQVT or ATL, this pattern is useless. Indeed, such languages already provide the possibility to define new operations on the metaclasses of the models they manipulate.

4.2 Hierarchical runtime events

Using a model animator is useful to see the evolution of the model being executed. Nevertheless, the presentation of all the states and events and the associated navigation can be quite complex. It is thus mandatory to provide a functionality close to the *step into*/*step over* behavior of program debuggers. When debugging a program, step into shows the code of the called method and the execution of each of its instructions, one at a time. On the contrary, step over executes the method call in one step and only the final state is seen.

[8] `http://www.obeo.fr/pages/acceleo`

The same kind of mechanism is useful for model animation. For example, in the case of UML state charts, one transition is triggered by an UML event. An action may be associated to this transition. The action may even be a compound action composed of several actions. It means that when the transition is fired, it will generate a new internal runtime event to run the associated compound action and then other internal events, one for each action of the compound action and so on. The user may want to see the effect for each individual action or only the result of firing the transition.

The solution we provide is based on hierarchical events. When an internal event is created it is considered as being generated by a parent event (either an external event or an internal one). When executing an event, the user can then decide to execute only that event (step into) or also its sub-events (step over).

This new functionality is not useful for the semantics implemented for SimplePDL, but would be required if we had hierarchical work definitions. We could also slightly change the semantics so that an activity whose load consumed is at 100% and all precedence constraints are fulfilled is automatically terminated. In this case, "*Change consumed load*", "*Start WorkDefinition*" or "*Finish WorkDefinition*" events may trigger new "*Terminate a WorkDefinition*" on the current activity or the activities depending on this one. Using step into and step over, the user could see individual changes to the model or only the final state of each work definition.

Hierarchical events are a more general solution than the previous one used for SysML/UML and SAM based only two levels of events (steps and micro-steps).

4.3 Improvement of the model graphical visualization

The previous model animators in the TopCased project only allowed to change graphical properties (like color, font, etc.) of graphical elements representing DDMM elements. For example, to animate UML state charts, current states were shown in red and fireable transition were displayed in green. Unfortunately, changing only graphical properties is not enough to display all the information that is managed in the SDMM. We have thus enhanced the visualisation for the model animator. The basic idea is to rely on the basic graphical editor and to add decorations to represent information added in the EDMM. Fig. 7 shows the new SimplePDL animator. The work load of a *WorkDefinition* is represented as a progress bar. An icon in the upper right corner of a *WorkDefinition* shows its state (not started, runing, finished or interrupted). The threshold of a *WorkDefinition* and the *linkType* are displayed on the arcs that link the *WorkDefinition*.

The decoration mechanism is already provided as an extension point by the GMF[9] library. The work has thus mainly consisted in adding decoration such as progress bars, labels, figures, etc. on the elements of the graphical representation and relying on EMF notifications to update the graphical representations on changes on the SDMM model.

For the moment, this work is done manually. We are now working on a generator that would allow the end-user to define the decoration he/she wants to have and then to produce the enhanced animator.

[9] The Eclipse Graphical Modeling Framework, `http://www.eclipse.org/modeling/gmf`

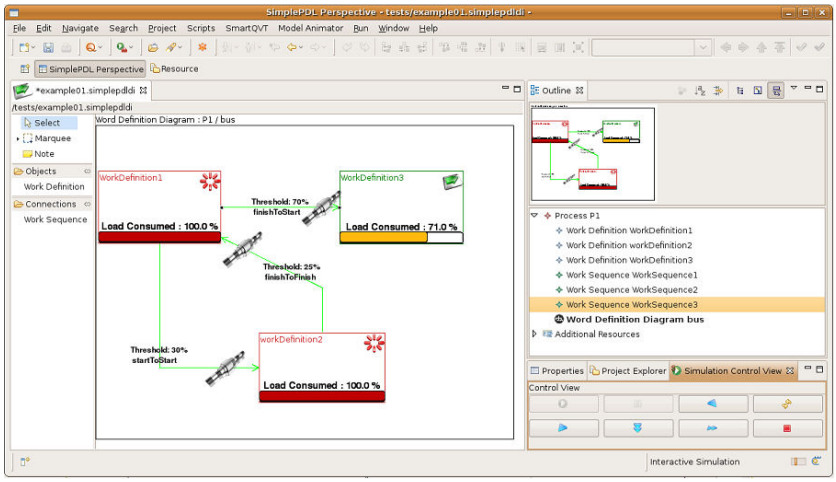

Fig. 7. Visualization of animated models by decorating the graphical editor

4.4 Controllers for Event Creation

The last enhancement described in this paper concerns the controllers that allow the user to inject new runtime events in the simulation. For UML state machines animators, a common controller had been defined. It mainly consists in collecting all the UML events[10] that could trigger transitions on the animated model. Those UML events were displayed in a list from which the user chooses the UML event to inject.

In the case of SimplePDL, runtime events have a target (a Process, a WorkDefinition or a WorkSequence) and may require parameters. For instance, IncreaseWDLoad targets a *WorkDefinition* and has an additional attribute corresponding to the value of the increment. In the same way, the newThreshold attribute of *ChangeWSThreshold* stores the new value of the threshold of a targeted *WorkSequence*.

Using Acceleo to perform model to text transformations, we have generated dialogs to display the possible runtime events and the associated parameters that have to be typed in. It is generated from the EDMM model. This dialog may be displayed from the contextual menu after having selected an element on the graphical visualisation. Obviously, only the events that have this type of element as target are selected.

4.5 Refactoring of existing animators

The above proposals have been applied to the existing model animators that had been hand coded in TOPCASED. The generators were applied on the metamodels following the execution pattern without any changes. Then the elementary semantics actions were cut and pasted from the previous implementation to the generated skeletons. This refactoring was done in less than one day for each model animator and provided identical tools with better visualisation capabilities thus validating our proposal.

[10] UML events are different from runtime events from the EDMM. Indeed the UML EDMM defines one runtime event called *InjectUMLEvent* which consists in injecting a UML event that will then be used to evolve the UML state machine (firing transitions).

5 Related works

Several tools support editions and simulations of models, described for example in an automata-like notation. Let us mention, among the more popular ones: *Sildex* [7], *StateMate* [8], *Uppaal* [9], the *Stateflow* module [10] in *Matlab/Simulink*, *Scilab/Scicos* [11], the Finite State Machine (FSM) model of computation of *Ptolemy II* [12], and the UML State Machine [13]. These tools provide graphical visualization of simulations highlighting active states and fireable transitions, coupled with means to visualize and record execution traces. Nevertheless, these tools embed their own hard-coded semantics for a given DSML, and there are important development work without possible reuse. In another way, we address in this paper a generative approach, specifying a DSML based tool for model animator definition of any DSMLs.

Sadilek, Wachsmuth et al. have followed a similar purpose in the EPROVIDE project: bestow a DSML with execution power [14,15,16]. Their framework allows to express the semantics of DSML using various technologies (including JAVA, PROLOG, ASM, QVT). They have experimented its use for PetriNet and SDL DSMLs. The dynamic informations are added to the metamodels in an ad-hoc manner depending on the use case, thus it does not allow to rely on generative technologies. Developers of graphical model animators are required to explicitly rely on APIs requiring a bit more work.

Soden, Eichler et al. have proposed the MXF (Model eXecution Framework) eclipse project [17] in order to define the M3Action graphical semantics description language. The EPROVIDE and TOPCASED projects are parts of the official potential technology users in the project and we plan to commit our metamodeling pattern for executable DSML and the associated generative tools in this context. We plan to provide implementation language specific adapter generators linked to EPROVIDE in that context.

6 Conclusion

The tools presented in this paper were the first results of the experiments in TOPCASED on generative technologies for model animator. We are currently extending that work in several directions:

- Common programming language debugger provides sophisticated conditional breakpoints facilities, we propose to rely on OCL in order to define conditional breakpoints that would stop the execution as soon as a property becomes false.
- Behavioral models properties usually encompass both static properties that must be satisfied at each step of the execution, and temporal properties that relates the various steps of an execution. The use of TOCL would allow to define conditional breakpoints triggered by sequences of events and not only state contents.
- The current semantics does not provide a step-back facility. The user must start again from the begining if he wants to jump back in time. In order to avoid to store all the intermediate states of the model, we propose to rely on a bi-directional semantics implementation.
- graphical decorations for model animation are currently hand-coded. We propose to define an animation configuration model derive from the graphical editor configuration model to specify the decorators that must be added for a given semantics.

Acknowledgement

This work was partially supported by the french government DGCIS through the FUI TOPCASED project. The authors wish to thank P. Farail and J.-P. Giacometi from Airbus for their helpful comments, and the team from Atos Origin for their intensive development work in TOPCASED.

References

1. Combemale, B., Crégut, X., Giacometti, J.P., Michel, P., Pantel, M.: Introducing Simulation and Model Animation in the MDE TOPCASED Toolkit. In: ERTS. (2008)
2. Combemale, B., Rougemaille, S., Crégut, X., Migeon, F., Pantel, M., Maurel, C., Coulette, B.: Towards rigorous metamodeling. In: MDEIS, INSTICC Press (2006) 5–14
3. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X., Vernadat, F.: A Property-Driven Approach to Formal Verification of Process Models. In: Enterprise Information System IX. Springer-Verlag (2008)
4. Bendraou, R., Combemale, B., Crégut, X., Gervais, M.P.: Definition of an executable spem 2.0. In: APSEC. (2007) 390–397
5. Farail, P., Gaufillet, P., Canals, A., Camus, C.L., Sciamma, D., Michel, P., Crégut, X., Pantel, M.: The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In: ERTS. (2006)
6. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Core Specification. (January 2006) Final Adopted Specification.
7. Winkelmann, K.: Formal Methods in Designing Embedded Systems - The SACRES Experience. In: Formal Methods in System Design. Volume 19. Springer (2001) 81–110
8. Harel, D., Naamad, A.: The STATEMATE semantics of Statecharts. ACM Transactions on Software Engineering and Methodology (TOSEM) **5**(Issue 4) (1996) 293–333
9. Behrmann, G., David, A., Larsen, K.G., Möller, O., Pettersson, P., Yi, W.: Uppaal - Present and Future. In: Proceedings of the 40th IEEE Conference on Decision and Control (CDC'2001), Orlando, Florida, USA (2001)
10. Colgren, R.: Basic Matlab Simulink and Stateflow. AIAA Education Series. American Institute of Aeronautics and Astronautics (2007)
11. Campbell, S.L., Chancelier, J.P., Nikoukhah, R.: Modeling and Simulation in Scilab/Scicos. Springer (2005)
12. Lee, E.A.: Overview of the Ptolemy project. Technical Memorandum UCB/ERL no M03/25, University of California at Berkeley (2003)
13. Dotan, D., Kirshin, A.: Debugging and testing behavioral uml models. In: OOPSLA Companion, ACM (2007) 838–839
14. Wachsmuth, G.: Modelling the operational semantics of domain-specific modelling languages. In: GTTSE. Volume 5235 of LNCS., Springer (2007) 506–520
15. Sadilek, D.A., Wachsmuth, G.: Prototyping visual interpreters and debuggers for domain-specific modelling languages. In: ECMDA-FA. Volume 5095 of LNCS., Springer (2008)
16. Sadilek, D.A., Wachsmuth, G.: Using grammarware languages to define operational semantics of modelled languages. In: TOOLS 2009. Volume 33 of LNBIP., Springer 348–356
17. Soden, M., Eichler, H.: Towards a model execution framework for eclipse. In: 1st Workshop on Behaviour Modelling in Model-Driven Architecture, ACM (2009) 1–7