A Design Pattern for Executable DSML

Benoit Combemale, Xavier Crégut, Marc Pantel

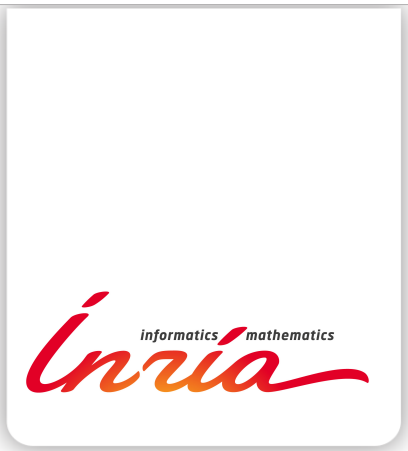
A Design Pattern to Build Executable DSMLs

Benoît Combemale
Université of Rennes 1, IRISA
benoit.combemale@irisa.fr

Xavier Crégut, Marc Pantel
Université de Toulouse, IRIT
{xavier.cregut, marc.pantel}@enseeiht.fr

RESEARCH REPORT
N° 8063
September 9, 2012
Project-Teams Triskell

ISSN 0249-6399 ISRN INRIA/RR--8063--FR+ENG



A Design Pattern to Build Executable DSMLs

Benoît Combemale
Université of Rennes 1, IRISA
benoit.combemale@irisa.fr

Xavier Crégut, Marc Pantel
Université de Toulouse, IRIT
{xavier.cregut, marc.pantel}@enseeiht.fr

Project-Teams Triskell

Research Report n° 8063 — September 9, 2012 — 19 pages

Abstract: Model executability is now a key concern in model-driven engineering, mainly to support early validation and verification (V&V). Some approaches allow to weave executability into metamodels, defining executable domain-specific modeling languages (DSMLs). Model validation can then be achieved by simulation and graphical animation through direct interpretation of the conforming models. Other approaches address model executability by model compilation, allowing to reuse the virtual machines or V&V tools existing in the target domain. Nevertheless, systematic methods are currently not available to help the language designer in the definition of such an execution semantics and related tools. For instance, simulators are mostly hand-crafted in a tool specific manner for each DSML.
In this paper, we propose to reify the elements commonly used to support state-based execution in a DSML. We infer a design pattern (called *Executable DSML* pattern) providing a general reusable solution for the expression of the executability concerns in DSMLs. It favors flexibility and improves reusability in the definition of semantics-based tools for DSMLs. We illustrate how this pattern can be applied to ease the development of V&V tools for DSMLs, either by direct interpretation of the model (*e.g.,* graphical model animators), or by translating it to another formalism to reuse a specific tool (*e.g.,* an existing model-checker).

Key-words: Software Engineering, Model Driven Engineering, Metamodeling, Model Validation & Verification

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

A Design Pattern to Build Executable DSMLs

Résumé : Model executability is now a key concern in model-driven engineering, mainly to support early validation and verification (V&V). Some approaches allow to weave executability into metamodels, defining executable domain-specific modeling languages (DSMLs). Model validation can then be achieved by simulation and graphical animation through direct interpretation of the conforming models. Other approaches address model executability by model compilation, allowing to reuse the virtual machines or V&V tools existing in the target domain. Nevertheless, systematic methods are currently not available to help the language designer in the definition of such an execution semantics and related tools. For instance, simulators are mostly hand-crafted in a tool specific manner for each DSML.

In this paper, we propose to reify the elements commonly used to support state-based execution in a DSML. We infer a design pattern (called *Executable DSML* pattern) providing a general reusable solution for the expression of the executability concerns in DSMLs. It favors flexibility and improves reusability in the definition of semantics-based tools for DSMLs. We illustrate how this pattern can be applied to ease the development of V&V tools for DSMLs, either by direct interpretation of the model (*e.g.,* graphical model animators), or by translating it to another formalism to reuse a specific tool (*e.g.,* an existing model-checker).

Mots-clés : Génie Logiciel, Ingénierie Dirigée par les Modèles, Métamodélisation, Validation & Vérification

1 Introduction

The use of Domain Specific Modeling Languages (DSMLs) is a key feature of Model Driven Engineering (MDE) because it makes easier the separation of concerns during the development process. Metamodeling in MDE is the modeling (i.e., definition) of DSML. Inspired by object-oriented modeling (*e.g.,* UML), metamodeling languages such as MOF (combined with OCL) were proposed to define abstract syntaxes of DSMLs as a class diagram (called *metamodel*). Metamodels capture domain specific concepts and their relationships. They are an essential part of DSML specific tool development. For instance, based on such a metamodel, syntactic tools may be partially or totally generated (*e.g.,* using GMF[1] and TMF[2] for graphical and textual model editors respectively).

Model executability is now a key concern in MDE, especially to support early validation and verification (V&V) in the development process. In this purpose, we are targeting systems which are time and context dependent. Their models provide a representation of complex cyber-physical system behavior (*e.g.,* software, computer hardware or even physical systems like sensors or actuators). The execution of a system relates its evolution (finite ordered sequence of states) during time according to its environment changes (external stimuli) and its internal state. Recently, several ways have been explored to implement the execution semantics of DSML. Basically, they map the abstract syntax, defined by the metamodel, to a semantic domain [**?**].

Most proposals translate models into an existing semantic domain in order to reuse available tools (*e.g.,* simulators, graphical animators or model-checkers). Such a semantics, called *translational semantics*, is used for instance by the group pUML[3] (a.k.a. *Denotational Meta Modeling*) in order to formalize some UML diagrams [**?**]. Even if more expressive languages like Maude [**?**] or TOPCASED-FIACRE [**?**] may be used to ease the writing of the translation between the DSML high level concepts and the formal language low level ones, this approach may require complex transformations to implement the semantic mapping. Furthermore, execution results are only obtained in the target domain. Getting back the results in the source language is difficult and usually requires to extend its abstract syntax in order to model these results.

Other approaches propose to weave executability into metamodels using an action language (e.g., Kermeta [**?**], xOCL [**?**], MOF action languages [**?**] or even JAVA with the EMF API). Similarly, endogenous model transformations[4], including graph transformations [**?**], were widely investigated to give a declarative specification of the execution semantics. For example, in [**?**] the authors use QVT, the OMG specification dedicated to model transformation [**?**], to express in-place rewriting rules that gradually compute the values of an OCL expression. Kuske et al. [**?**] have used graph transformation to define the executable semantics for some UML diagrams. These approaches allow a more intuitive definition of executable DSMLs. The semantic domain is an extension of the abstract syntax, and the semantic mapping is defined using an action language. Thus, the language designer has only to deal with concepts of the DSML and not with another language and an explicit mapping. Nevertheless, such approaches require to implement for each DSML all the execution-based tools.

In all cases, the definition of DSMLs is facing today hard methodological problems for the specification of tool supported execution semantics. DSMLs are often empirically defined without any uniformity and underlying best practices [**?**]. For example, the information capturing the state of a model being executed, a key part of the semantic domain, is often scattered in a tool-specific way, without any explicit relation to the abstract syntax. Thus, different tools such as simulators, model checkers or code generators may easily be inconsistent, and not interoperable as they rely on slightly different semantic domains. In the same way, no methodology to define an executable DSML provides the flexibility to associate dif-

[1] Eclipse Graphical Modeling Framework, `ww.eclipse.org/modeling/gmf`
[2] Eclipse Textual Modeling Framework, `www.eclipse.org/modeling/tmf`
[3] *The precise UML group*, `www.cs.york.ac.uk/puml/`
[4] The source and target models conform to the same metamodel.

ferent semantics to the same DSML, to combine different models of computation (*e.g.,* multi-modeling), and to easily weave time and communication models; nor the evolvability to manage semantics changes. Consequently, semantics-based tools (*e.g.,* simulators and graphical animators) are most of the time re-defined without any capitalization (*e.g.,* dynamic execution related information, execution engine, etc.), and without any guidances to ease this error prone and time consuming development task.

In this paper we introduce a general, reusable and tool-supported approach that leverages existing works to assist a DSML designer in the definition of an execution semantics and the related tools. It relies on capturing the different concerns involved in the definition of an executable DSML. These concerns are reified, in a structural design pattern to support executability into DSML: the *Executable DSML* pattern. It addresses several common use cases relying on execution semantics, especially model V&V. Based on this pattern, generic and generative approaches are proposed to partially or totally automate the definition of DSML tools for V&V.

This work has been applied in the TOPCASED project [**?**], an open-source MDE toolkit for safety critical application design. V&V capabilities for MDE are one of its key features. It is therefore of uttermost importance to ease the development of V&V tools for the various DSMLs considered in TOPCASED. Especially, the application of this work led to the development of the current SYSML/UML model simulator and graphical animator that handle the Class, State Machine and Activity diagrams including OCL constraints. In this context, we first illustrate our pattern on graphical model animation for validation purposes. With the help of our design pattern, we show how a model execution framework has been defined to offer an independent Model of Computation (MoC) shared by different DSMLs. We also present generative tools to automate the definition of dedicated (i.e., DSML-dependent) graphical model animators. While our approach provides broader support for model execution and V&V facilities, we also give insights on the use of our pattern to provide model-checking facility. We show how the pattern ease the definition and the verification of the translation from the DSML to a dedicated formalism in order to reuse an existing model-checker.

The remainder of this paper is structured as follows. Section 2 introduces the *Executable DSML* pattern illustrated with model animation for validation. Section 3 details this use of the pattern and proposes both a reusable MoC-specific model execution framework, and the associated generative approach to ease the definition of DSML-specific graphical model animators. Section 4 presents another application of the pattern to ease the definition and verification of translations required to reuse specific existing V&V tools. Section 5 summarizes related works. Section 6 concludes and gives insights on perspective.

2 A Metamodeling Pattern for Model Execution

In this section, we follow the common design pattern description format used in [**?**]. We rely on the model simulation and graphical animation of UML State Machine diagrams [**?**] in order to introduce the requirements for model execution at a conceptual level. This example, a small subset of the TOPCASED model simulator and graphical animator, is further detailed in Section 3.

2.1 Motivation

As explained in the introduction, the DSML semantics is usually enclosed (generally hard-coded) in the execution and transformation functions hidden in the system development tools. Our purpose is to make its definition explicit, including the semantic domain and the mapping as advocated in [**?**].

The designer of a model that describes a system behavior usually needs to simulate and animate it to check whether it behaves as expected. Unfortunately, the metamodel does not generally describe all the information that has to be managed at execution time (i.e. the semantic domain). For example, UML State Machine defines the concepts of `Region`, `State`, `Transition`, `Event`, etc. but lacks the notions of active

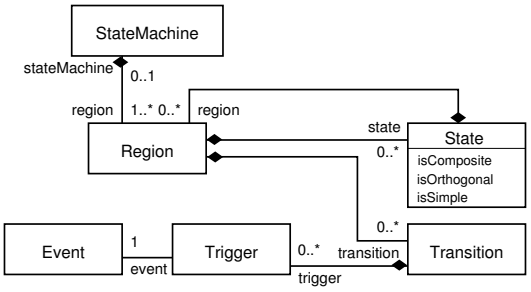


Figure 1: Subset of the UML StateMachine Metamodel that focuses on the elements that are used in this article.

states in a region, or of fireable transitions (cf. Figure 1). Also, no elements are available to store the sequence of events received by a state machine. Furthermore, during model animation, the designer has to simulate the behavior of the system environment through stimuli. The UML State Machine designer will inject UML events in a state machine that will trigger fireable transitions and change the current states of the regions. Obviously, the way the system reacts to the stimuli defines its execution semantics. This reaction updates the execution related data according to the current state of the model and the received stimulus. In the end, the designer may want to replay the same execution, for example, to check whether defects have been corrected or not, or to be able to perform non regression tests. Scenarios are then useful to describe a sequence of stimuli.

We have highlighted that model execution requires the extension of a DSML metamodel with:

- the definition of information managed during execution,
- the definition of the stimuli that trigger the evolution of the model,
- the organization of stimuli as scenarios,
- the definition of an execution semantics (or transition function) that describes how the model state evolves when a stimulus occurs.

An *executable DSML* (xDSML) is a DSML which defines the execution of its conforming models for a particular purpose. Therefore, an executable DSML at least includes the definition of its language abstract syntax, and its execution semantics (including semantic domain and semantic mapping related information)[5].

We propose to reify execution related elements to make them explicit and manageable. We aim to provide flexibility, evolvability and interoperability in the semantics definition. Furthermore such elements must ease the development of tools related to model execution, for example V&V tools.

2.2 Structure

Figure 2 shows the structure of the proposed *Executable DSML* pattern. It is built from four structural parts (detailed in the next subsection) that are woven together using the «`merge`» and «`import`» predefined package operators of MOF [**?**]. These parts organize the data related to the DSML and its execution semantics. A fifth part called *Semantics* provides the execution semantics itself relying on the previous

[5]In MDE, model execution is relying on the abstract syntax of the language used to build it, but not (necessarily) on the concrete syntax. Therefore, we let consequences of model execution on the concrete syntax as perspectives. Nevertheless, Section 3.2 explains how model execution changes are reflected to the graphical concrete syntax used by model animators.

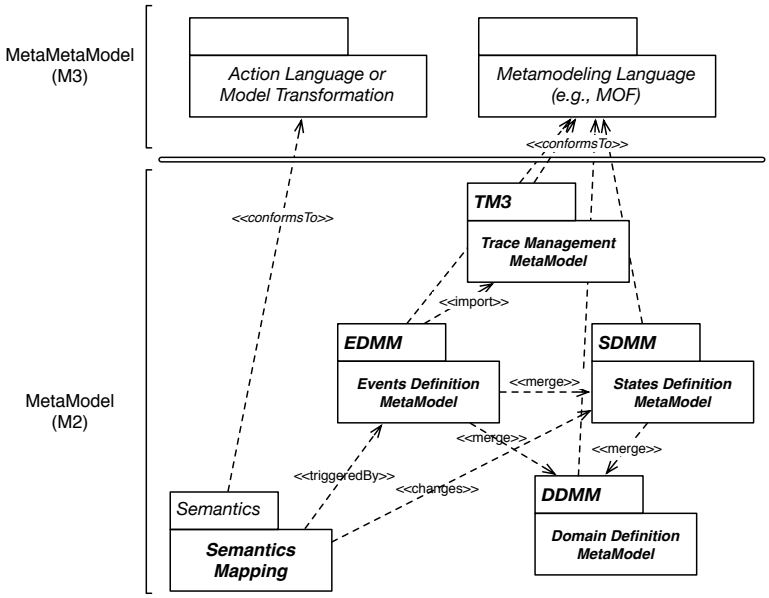


Figure 2: The *Executable DSML* Pattern

four parts (i.e., the semantic mapping based on the previous reification of the semantic domain information). As it is a pattern to organize data at the metamodel level (i.e., a *metamodeling pattern*, as motivated in [**?**]), the structure shows dependencies between packages that represent parts of a metamodel. This pattern is architectural like *MVC* or *3-tiers*. It emphasizes the common structure that a metamodel for an xDSML should use in order to define the language semantics. In addition to provide guidelines in language definition, the purpose is to be able to define generic and generative tools relying on that architecture.

2.3 Participants

2.3.1 Domain Definition MetaModel (DDMM)

It is the usual metamodel used by standardization bodies to define the modeling language. It provides the key concepts of the language (representing the considered domain) and their relationships extended with structural constraints. For instance, the UML metamodel defined by the OMG using MOF is a DDMM. A small subset of this metamodel is shown on Figure 1. Usually, the DDMM does not contain all the execution-related information. For instance, the UML DDMM does not formalize the notions of *active state* nor *event queue*. Thus, even if a model describes the implicit potential behavior of a system, it does not usually provide explicitly the elements for its execution.

2.3.2 State Definition MetaModel (SDMM)

During the execution of a model, additional data is usually mandatory for expressing the execution itself (a.k.a. dynamic information). Such data must be manipulated and recorded (in the form of metaclass instances). For example, each active UML region must have one active state and a state machine must store the sequence of received events. These execution related data make up the SDMM, and are related to the semantic domain: the data required to express the execution semantics. Thus the SDMM is built

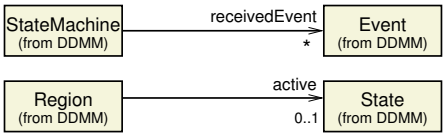


Figure 3: One possible SDMM for UML State Machines

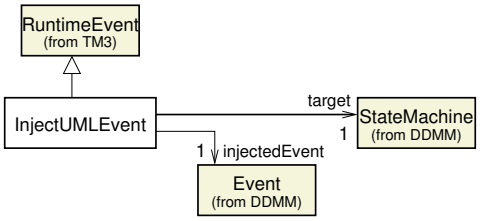


Figure 4: One possible EDMM for UML State Machines

on top of the DDMM. For example, the UML State Machines SDMM (cf. Figure 3) may add a reference from `Region` to `State` (both defined in the DDMM) to record the active state of one region.

2.3.3 Event Definition MetaModel (EDMM)

The EDMM of a given DSML specifies the concrete stimuli (called runtime events) that drive the execution of a model that conforms to this DSML. These stimuli are not only concrete system hardware events, but also more abstract software events like storage events for reading or writing, communication events for sending or receiving, clock events as ticks, function events like computation results given parameters, etc. Concrete stimuli define properties of events related to the formal execution semantics to be supported.

As an illustration, the runtime event we consider for the UML State Machine stores an UML event (an instance of `Event`, see Figure 1) in a state machine queue (cf. Figure 4). When the UML event in the queue is handled by the state machine, it fires the transitions that it triggers.

2.3.4 Trace Management MetaModel (TM3)

The TM3 is specific to a particular MoC and is reused for all DSMLs using this MoC. As an example, Figure 5 shows a simplified TM3 dedicated to discrete-events system modeling [**?**]. It defines three main metaclasses called `Trace`, `Scenario` and `RuntimeEvent`. `RuntimeEvent` is an abstract metaclass which reifies the concept of stimulus. It is an abstraction for any kind of semantic related stimulus defined in the EDMM. To this end, `RuntimeEvent` is imported in the EDMM, and all the concrete runtime events must inherits from it. This metaclass has executability-related features, like (partially ordered) dates of occurrence (i.e., symbolic representation of the time when the runtime event occurs). Any `RuntimeEvent` that triggers a semantic action involving a state change should have a reference to its source and target states information in the SDMM. `RuntimeEvent` instances fall into two categories, which are modeled by the `RuntimeEventKind` enumeration. Exogenous runtime events are injected by the environment, while endogenous runtime events are produced internally by the system in response to another runtime event (cf. `cause` in Figure 5). As stated by the OCL constraint in Figure 5, a scenario is made of exogenous runtime events whereas a trace corresponds to one possible execution of a scenario and is thus composed of any kind of runtime events. A more sophisticated trace management metamodel or a "standard" one (like the UML Testing Profile [**?**]) may be integrated in our pattern.

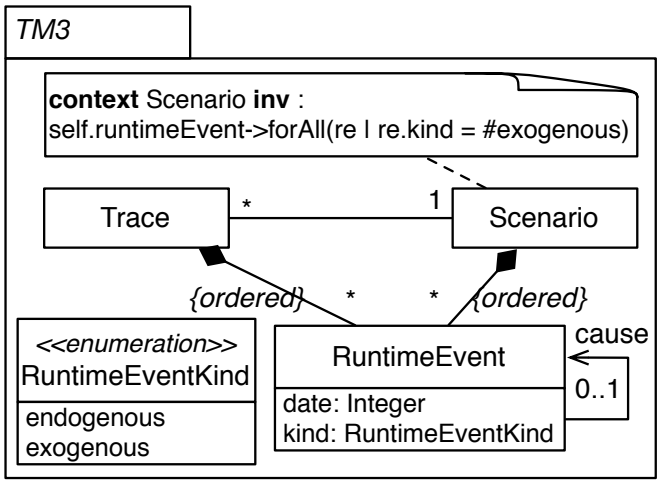


Figure 5: A simplified TM3 for Discrete-Events Modeling

2.3.5 Semantics

The last and key participant is the package *Semantics*. It abstracts both the semantic mapping [**?**] (DSML-specific part) and the interactions with the environment (MoC-specific part).

It describes how the running model (SDMM) evolves according to the stimuli defined in the EDMM. An important point in applying the pattern is to define the content of the package *Semantics* that depends on the application context. On the one hand the semantic mapping may be explicitly defined as a transition function and thus conforms to an action language (a.k.a. operational semantics). In this case, the four previous participants correspond to the semantic domain. Such kind of semantic mapping is usually used for model graphical animation (cf. Section 3). On the other hand, the semantic mapping may be implicitly defined thanks to a translation to another language (a.k.a. translational semantics). Consequently SDMM and EDMM do not correspond to the semantic domain but help in defining the mapping, and in getting results back (cf. Section 4). This second approach is usually applied to map the pattern on the internal data of an already existing semantics or on the concepts of another executable language.

2.4 Consequences

According to the *Executable DSML* pattern, an xDSML is supported by an executable metamodel MM_x structured as three DSML-specific parts (DDMM, SDMM, and EDMM) and one MoC-specific part (TM3):

$$MM_x = \{DDMM, SDMM, EDMM\} \cup \{TM3\}$$

MM_x reifies the elements involved in model execution. The DDMM is the starting point. It is usually standardized and cannot be changed in order to preserve interoperability. The TM3 is shared by any DSMLs relying on the same MoC. Thus, a semantics is defined by a triplet (SDMM, EDMM, *Semantics*). The SDMM and the EDMM introduce the needed information to express the execution semantics (i.e. the semantic domain) whereas the package *Semantics* implements the semantic mapping. These three different parts should not be defined independently in order to reduce the risks of inconsistencies. Any change in this triplet entails a new semantics. In order to reduce these risks, we propose through the use of this pattern to reify the various aspects linked to the definition of the execution semantics in order to allow systematic specification, analysis and validation of an executable DSML metamodel.

Applying this pattern produces several consequences, both for the definition of the semantics, and for the definition of the execution-related tools.

2.4.1 Definition of the Semantics

- *The pattern allows a modular implementation of the execution semantics* (i.e., an implementation that is separated out, encapsulated, and easily replaceable) with respect to the core language metamodel, the DDMM. The specification of the DSML semantics is split in two parts: first, a generic MoC based on the TM3, and shared with other DSMLs; and then DSML specific elements based on the SDMM and EDMM. This strong property provides several benefits described here after.

- *It favors the evolvability of the semantics during the DSML lifetime* thanks to the separation of concerns involved in the definition of an execution semantics.

- *It eases the factorization of commonalities.* The pattern favors the definition of a family of semantics for a single language as well as the semantics of a family of languages. For example, semantic variation points (like in UML) lead to different but similar semantics definitions. In most cases, SDMM and EDMM are the same and only the package *Semantics* has to be adapted.

- *It provides flexibility in the association of semantics to a given DSML in order to define several purpose driven semantics for the same DSML.* Obviously, runtime information (SDMM), concrete runtime events (EDMM) and the package *Semantics* are dependent on the user purpose during the execution of models. For instance, the user may prefer to carry out more abstract execution with fewer runtime events and/or runtime information that demonstrates one aspect of the system under assessment or the user may want to define a fine-grained semantics that exhibits most aspects of the system. Each semantics will have its own set of events in the EDMM and states in the SDMM.

- *No specific method is enforced to apply the pattern.* Nevertheless, we have proposed in [**?**] a method for the definition of DSML execution semantics dedicated to verification activities. It advocates a property driven approach: only runtime information and events required to evaluate properties of interest to the end user are described. In doing so, the EDMM and SDMM are a minimal mandatory subset of data to express the semantics relevant for the user, as advocated by the substitutability principle [**?**].

- *The definition of the package* Semantics *is postponed.* The pattern is mainly an architectural pattern that helps in structuring information required to make a DSML executable while ensuring interoperability between tools based on this DSML. Thus, the semantic mapping and the interaction with the environment are not described in the pattern (as discussed in Section 2.3). According to the purpose of empowering a DSML with execution, the content of the package *Semantics* may be detailed. For example, Section 3 shows a MoC-specific framework for model execution. In most cases, the architecture of the MM_x eases the definition of the package *Semantics*. However, for scalability, efficiency, and some time readability purposes, it might be useful to introduce a new metamodel not relying on the standard DDMM. For example, the use of matrices to encode Petri nets instead of graphs is mandatory to allow the execution of huge models. This is also true in the case of General Purpose Modeling Languages (GPML) whose standard metamodel (DDMM) and semantics can be extremely complex. The introduction of purpose specific metamodels allows to ease the definition of the semantics for a subset of the language that the end user wants to assess.

- *Semantics is discrete event oriented.* The EDMM part of the pattern stresses the use of discrete events to represent system stimuli. It may not be well-suited for all systems, like continuous one. Nevertheless, we can notice that when one wants to observe a continuous system, a discretization

(on events or time) is performed. Thus, the pattern is still applicable as this is done in PTOLEMY II [**?**] for example. Time may be managed continuously as part of the MoC or discretized as runtime events.

2.4.2 Definition of the Execution-Related Tools

- *The formalization of pattern elements favors the definition of generic and generative execution-based tools.* Examples are given in Section 3 and Section 4.
- *Several models of computation (MoCs) may be used to support symbolic execution semantics.* The description of the EDMM and TM3 might give the impression that the semantics is restricted to a discrete event MoC. In fact, these parts of the pattern define the discrete observations and interactions between the user/environment and the system, but any MoC can be used, including continuous ones. Our aim is to describe systems that in the end will be managed by either discrete software systems or human end users. Both can only handle a finite discrete history of the system. Thus the MM_x architecture is strongly based on the user point of view: observation of the interaction between the model and its environment (depicted by the model state) at some key points in time represented by the runtime events. However, the package *Semantics* can implement any MoC or abstract the translation to an existing one.
- *Cosimulation and models at runtime* can be integrated. The package *Semantics* can also be implemented as a wrapper over, either real physical systems in which sensors and actuators are mapped to MM_x directly or through software layers, or existing softwares and execution engines. Several DSMLs can also be integrated through shared data in their MM_x and synchronization/cooperation in their packages *Semantics*.
- *It favors interoperability between the various semantics-related tools for a given DSML.* Different kinds of tools may be based on the same executable DSML (*e.g.,* model simulator and graphical animator, *model-checking* based verification tool, etc.). The separation between MM_x and the package *Semantics* makes possible to share data between tools (i.e., a counter example provided by a verification tool can be analyzed using a graphical animator). However, this relies only on structural similarities and also requires to assess the compatibility of both packages *Semantics* (i.e., by checking the bisimilarity of the transition relations, see Section 4).
- *The logical view promoted by the pattern should be supported at the implementation level.* When implementing the pattern, the logical view enforced by the pattern to represent information related to the execution semantics may be lost. Indeed, inheritance may be used to implement the «`merge`» operator but leads to a strong dependency. If the *Decorator* pattern [**?**] avoids this drawback, high level paradigms like aspects [**?**] are preferred because they ensure that the separation of concerns in the logical view of the pattern is preserved at the implementation level.

The pattern introduced in this paper offers a methodology to define executable DSMLs. Then, its explicit architecture can be used to provide generic and generative approaches to ease the development of dedicated execution related tools.

3 Integrating V&V by Interpretation: Application to *Graphical Model Animation*

The *Executable DSML* pattern was designed and experimented in the TOPCASED project. This project aims at developing an open source CASE environment for the design of safety critical systems. MDE

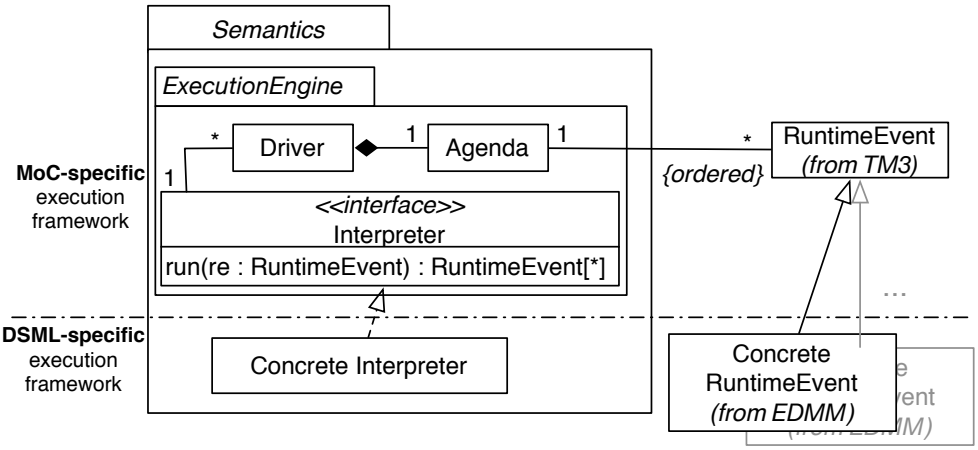


Figure 6: The TOPCASED Model Execution Framework

is strongly used in the TOPCASED project. In particular, each language it provides is defined thanks to a metamodel. Furthermore, a graphical editor generator has been developed and has been used to automatically build graphical model editors. The first aim of model animation in TOPCASED is to animate models according to their layout in the graphical editors so that their designer may visually validate their models behaviors. The current graphical model animators for SYSML/UML Class, State Machine and Activity diagrams including OCL and SAM[6] have been developed by ATOS relying on the *Executable DSML* pattern thus reaching a higher level of abstraction and reuse. The modularity and separation of concerns provided by the pattern allowed to share the MoC-specific execution engine between the various DSMLs. Moreover, several generative tools have been developed to ease the development of graphical animators [**?**]. Their development time was drastically reduced: only a couple of hours has been spent to retrofit and redevelop the UML State Machine animator using the already developed semantics. This section presents both the discrete event based implementation of the package *Semantics* and the generative approach used for the graphical interface.

3.1 A MoC-specific Framework for Model Execution

The TOPCASED project addresses the domains of aeronautics, aerospace and more generally transportation systems. Dynamic behaviors and real-time features prevails in the design of such systems. During the design of their software parts, discrete (synchronous or asynchronous) modeling [**?**] is the most adequate way to represent them. Thus, in this context, only the discrete event MoC is used for the model execution of any DSMLs.

We have applied the *Executable DSML* pattern for model execution and developed a framework (cf. Figure 6) included in the TOPCASED toolkit. The *execution engine* is the core of the framework. It implements a model execution engine for a discrete event based MoC. It is independent of any DSMLs (top of Figure 6) as it only depends on the TM3 (`RuntimeEvent`) and the `Interpreter` interface from the package *Semantics* which abstracts the transition function that will be provided by DSML-specific packages *Semantics*. Its `run` method updates the dynamic information of the model defined in the SDMM according to one runtime event (instance of the events defined in EDMM) and returns the list of generated endogenous runtime events. For a particular DSML, one has to provide both the implementation of the `Interpreter` and the concrete runtime events in EDMM (bottom of Figure 6).

[6]SAM is an AIRBUS CORPORATE specific structural analysis modeling language used for the A350 plane.

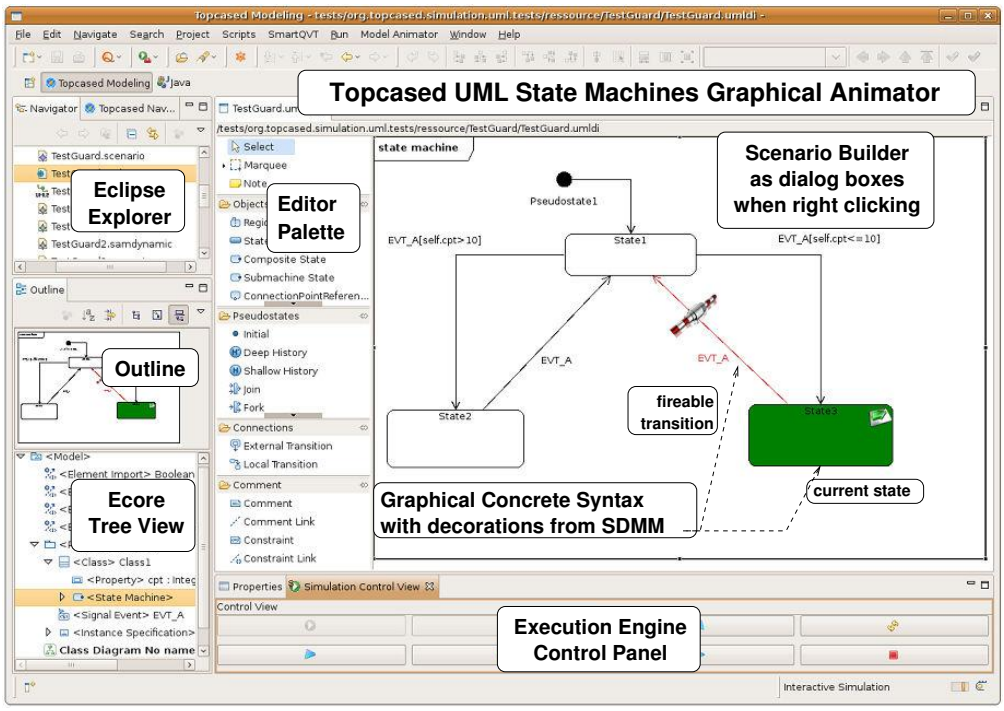


Figure 7: The Initially Generated UML State Machine TOPCASED Animator

Besides the interpreter, the execution engine is composed of two main components which implement the discrete events MoC: `Agenda` and `Driver`. The agenda (`Agenda`) stores the runtime events (`RuntimeEvent`) corresponding to one particular execution. These events are ordered according to their occurring date. The agenda provides the API required by the driver to handle the events (e.g., retrieving the next event and adding a new event).

The driver (`Driver`) controls the execution. It contains a `step` method, which gets the next runtime event from the agenda and asks the interpreter (`Interpreter`) to handle it. The generated endogenous runtime events are then added to the agenda. The driver provides an API that allows both batch and interactive execution.

For each execution semantics of a given DSML, a `Concrete Interpreter` must implement `Interpreter` (cf. Figure 6). For the TOPCASED animators, the `run` method was initially hand-coded using JAVA and the EMF API. Then, it was implemented using SMARTQVT[7], an open source implementation of the OMG QVT specification [**?**] that generates JAVA code relying on the EMF API. SMARTQVT mainly eases the navigation on model elements. Any other techniques for semantics implementation may be considered (cf. Introduction).

3.2 A Generative Approach for Model Animation

Figure 7 shows the components of the TOPCASED Eclipse-based animators for UML State Machine. The main view displays the graphical syntax which is reused from the DSML editor. This one is decorated with dynamic information (coming from the SDMM): active states are highlighted with a green background, fireable transitions are in red with an icon. Other decorations like gauges or progress bars

[7] Cf. `http://sourceforge.net/projects/smartqvt/`

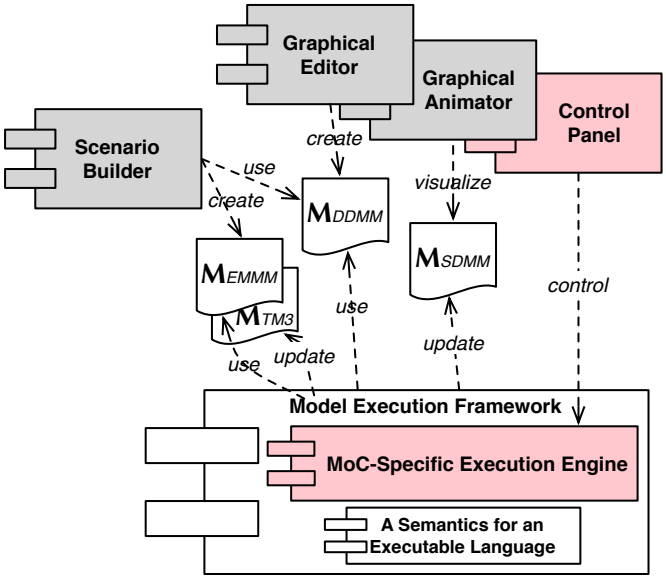


Figure 8: Interactions between Simulator Components

could be used. This visualization has been chosen because it allows the user to view dynamic information directly on the domain model he has built. Nevertheless, other specific visualization tools could be developed. The *Eclipse Explorer*, the outline of the diagram and the tree view of the underlying model are on the left side. The bottom view is the *Control Panel* that manages the execution (start/stop the animation, move forth/back in the trace, etc.). During interactive animation, the user may inject an instance of a runtime event by clicking on the concerned graphical element. A dialog box prompts the user for the UML event name. It is part of the *Scenario Builder* of which the purpose is to manage a scenario before the animation starts (based on the requirements, a test case generator, or a counter-example provided by a model-checker), or during the animation in an interactive fashion.

Figure 8 shows the data flow between the animator components and the concerns modeled in the MM_x. The end user first edits the domain model (M_{DDMM}) using the editor. Then, the end user can build a scenario (M_{EDMM} and M_{TM3}) with the *Scenario Builder*. Inside the editor, the end user can launch the animator. This triggers the creation and the initialization of the dynamic model (M_{SDMM}) from the M_{DDMM}, and according to the DSML execution semantics. A scenario or a trace (M_{EDMM} and M_{TM3}) may be used to initialize the execution engine agenda. Then, while the simulation is running and runtime event instances are handled, the execution engine updates the M_{SDMM} and the trace (M_{EDMM} and M_{TM3}). Finally, each time the dynamic model is modified, the associated graphical decorations are updated thanks to the EMF notifications.

Figure 9 highlights the parts of the *Executable DSML* pattern used to derive the animators' components. Some implementation insights are given hereafter (see [**?**] for further details). The graphical *editor* is built from the DDMM. It is generated from a TOPCASED configuration file that maps graphical elements to the metamodel elements. The *animator*'s view (i.e., the visualization of dynamic information) is based on the SDMM. It is implemented using the decorators provided by GMF and relies on the EMF notifications to update the graphical representation when the running model is changing.

The *Scenario Builder* relies on the EDMM (and thus on the TM3). It mainly consists in defining a concrete syntax for the runtime events defined in the EDMM of a given DSML to provide the way of building scenarios. The underlying tooling can then be built using generative approaches such as GMF

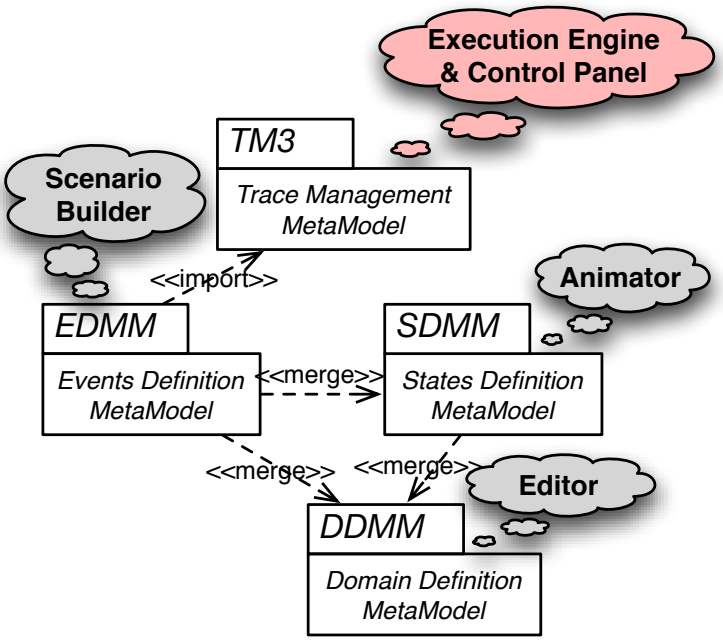


Figure 9: Pattern-based Tooling for Model Simulation

or TMF. In interactive mode, the dialog boxes are generated from the runtime event attributes.

The TM3 is used by the execution engine that stores all the events in the agenda as a trace. It is also used by the *control panel* to go back and forth in that trace.

For this animator, the view is derived from the classical graphical model editor by adding additional decorators whose contents are based on the values in the model state. This case is quite common to reuse the existing graphical editor and to add decorators. Nevertheless, the pattern does not require to rely on the editor view. Other views can be defined for model animation based on the content of the SDMM, EDMM and TM3.

4 Integrating V&V by Translation: Application to Model Checking

Section 3 presented the use of the *Executable DSML* pattern as a methodology to implement an operational semantics and to ease the development of the associated model validation tools. The provided package *Semantics* implements an operational semantics which uses MM_x as the semantic domain by interpreting the model. This section highlights how the *Executable DSML* pattern can be seamlessly applied to other use cases, in particular for translational semantics to reuse an existing V&V tool. Once again, the key point is the definition of the package *Semantics*.

Verification activities ensure that the final system or one of its models is compliant with respect to another model (usually a property or more abstract model). We are only considering the system behavior at runtime, models are thus executable. We have focused our experiments on model checking, but other technologies could be integrated using the same approach. A model checker builds a graph of all possible accessible states taking into account all possible events triggering the transition function. Then it assesses the compliance for all states, or traces (i.e., paths), in the graph. In case of failure it returns a state, or a trace, that does not comply to the property or does not refine the previous model. These elements

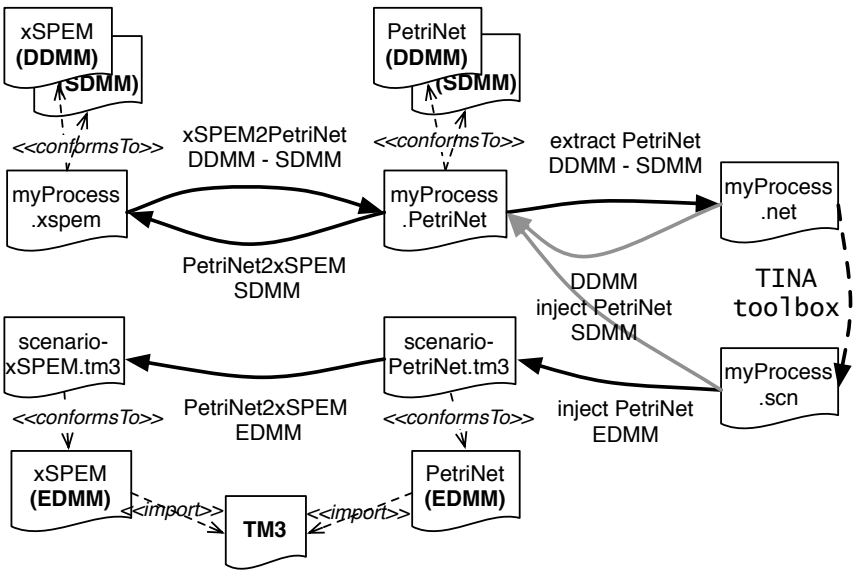


Figure 10: Pattern-based Approach for Model Verification (applied to xSPEM models)

can be represented as instances of the metaclasses in SDMM, EDMM and TM3; and implemented with the previous discrete event based MoC. However, model checkers are very complex tools, thus it is more efficient to integrate existing ones in the package *Semantics* through back and forth model transformations between the DSML and the model checker dedicated modeling language (*e.g.,* Petri nets as illustrated in Figure 10). In this approach, such model transformations in the package *Semantics* implement a translational semantics of the source language according to the semantics of the target language.

This kind of integration is a common practice since the early 80's, however it usually relies on ad-hoc solutions and suffers from the lack of methods and tools. We experimented the use of the pattern in TOPCASED first with XSPEM [**?**], an executable extension of SPEM [**?**], then with the LADDER PLC specification language [**?**]; both were translated to PETRINET and verified using the TINA toolset [**?**]. In the remainder of this section we show how the structure of the pattern provides guidelines for the specification and verification of the transformations.

First, the MM_x architecture must be used for both source and target DSMLs. Since the pattern proposes an explicit architecture of the information needed by all state-based languages, an existing language that does not use initially the pattern could be refactored accordingly. As an example, Figure 11 proposes a simplified version of the Petri nets metamodel used in our experiments to translate the XSPEM and LADDER PLC languages. The architecture of this metamodel has been refactored to highlight the underlying pattern. *PetriNet_DDMM* defines a Petri net as composed of transitions and places connected using arcs. *PetriNet_SDMM* merges the previous one by the marking (i.e., the number of tokens in each place) to capture the state of a running Petri net. *PetriNet_EDMM* defines one runtime event: firing one transition from the Petri net DDMM model.

Then, the architecture of the source and target languages are used as guidelines to specify the transformations between them. The source DDMM is mapped to the target one (PETRINETDDMM in Figure 10) to handle the structural parts of the model (transformation *xSPEM2PetriNet – DDMM part*). The SDMM provides the initial state (*xSPEM2PetriNet – SDMM part*). The state or trace resulting from a verification failure is expressed using the TM3, as well as the target SDMM and EDMM. The backward

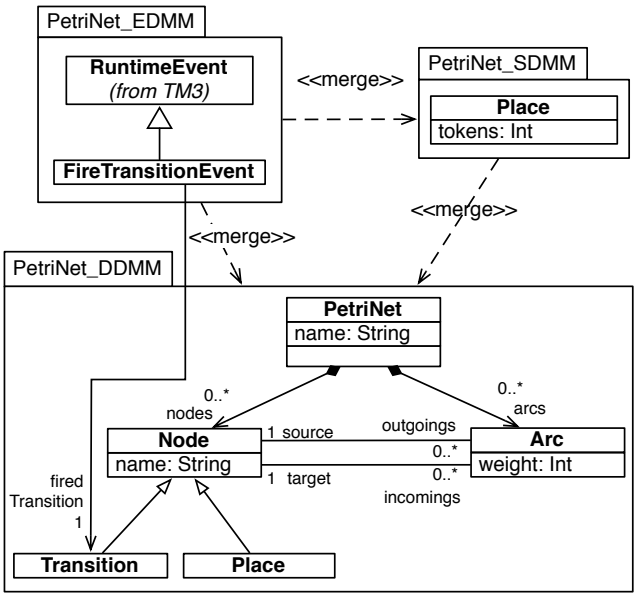


Figure 11: A PETRINET Metamodel

transformation pushes the results in the end user domain using the TM3, as well as the source SDMM (*PetriNet2xSPEM – SDMM part*) and EDMM (*PetriNet2xSPEM – EDMM part*). Thanks to the *Executable DSML* pattern, such backward transformation can even be automatically defined with a generic tool as we proposed in [**?**] for tracing executions back to a DSML's operational semantics.

This implementation of the package *Semantics* is thus a kind of translational semantics to a semantic domain not related to the DSML. The injectors (*text-to-model* translations) and extractors (*model-to-text* translations) allow to map the various parts of the PetriNet models to the input and output of the TINA toolset. The use of the MM_x architecture on both sides of the translation provides a much better result feedback than with usual translational semantics.

Of course, these transformations must preserve the intended semantics: the entailed relation between the events in both EDMM must be a (weak) bi-simulation. The pattern provides guidelines to establish the proof of such relation, as we experienced in [**?**] to prove the transformation from XSPEM to PETRINET.

5 Related Work

A language semantics is usually defined operationally or by translation (cf. Section 1). Both may be considered with the *Executable DSML* pattern and correspond to the technology used to implement the semantic mapping in the package *Semantics*. The objective of the *Executable DSML* pattern is twofold: it provides a methodological framework to ease the use of such approaches as well as an architectural framework to be supported by generative and generic tools.

Such *an engineering of semantics for DSMLs* is at the heart of some previous work. Sadilek et al. have followed a similar purpose to ours in the EPROVIDE project: provide execution power to DSMLs and ease the development of related tools [**?**, **?**]. Their framework enables the implementation of a DSML semantics using various technologies (including JAVA, PROLOG, ASM, QVT). They have prototyped its use for PetriNet and SDL DSMLs. In such a context, the *Executable DSML* pattern assists the DSML

designer in structuring the dynamic information and provides a more abstract basis to build graphical model animators without explicitly relying on APIs.

In [**?**] the authors propose a reification of the dynamic information according to the execution semantics specified thanks to UML Activity Diagrams. The *Executable DSML* pattern complements this approach by reifying runtime events to assist the implementation of both operational and translational semantics, and to be explicitly manageable by V&V tools.

Otherwise, several existing tools support edition and execution of models described in an automata-like notation (mainly for simulation purpose). Let us mention, among the more popular ones *StateMate*[8], *Uppaal*[9], the Stateflow module in the *The MathWorks* Simulink framework[10], the Finite State Machine (FSM) model of computation of *Ptolemy II*[11], and the UML State Machines from most software tools editors. Based on simple or timed automata, these tools provide graphical visualization of simulations highlighting active states and fireable transitions, coupled with execution trace visualization and recording. These existing tools have been defined manually for a specific DSML. They embed their own semantics making it difficult to ensure their compatibility. Moreover, the interoperability between the various tools is difficult due to their own characterization of the states (i.e., the SDMM), without any clear reification. The dynamic information are generally deeply embedded in the tools, potentially with different choices or abstraction levels. The *Executable DSML* pattern proposes to explicitly reify such information at the metamodel level to improve flexibility and reuse. The pattern provides also the way to integrate an existing execution engine by implementing the translation as part of the package *Semantics*.

6 Conclusion and Perspectives

The *Executable DSML* pattern introduced in this article provides a clear separation of concerns in metamodels for the recurring and general problem of model execution. Static and dynamic information from running models are reified in separate parts, respectively the *Domain Definition MetaModel* (DDMM) and the *States Definition MetaModel* (SDMM). Specific events triggering the evolution of the running model are also reified in the *Events Definition MetaModel* (EDMM). The model of computations (MoC) and the transition function (as execution or translation functions) are abstracted by the package *Semantics*. Finally, MoC-specific information such as trace and scenario are also reified in the *Trace Management MetaModel* (TM3).

The *Executable DSML* pattern that we propose provides a methodological framework to ease and assist the definition of language semantics. The *Executable DSML* pattern favors also the definition of generic and generative technologies to ease the development of semantics-based tools for DSML such as model animators.

The TOPCASED toolkit embeds several DSMLs such as UML/SysML, for which the definition of simulators is a key point. Each of them has been extended to be executable, relying on the *Executable DSML* pattern. Based on the same discrete event MoC, a flexible execution framework has been defined relying on the same TM3, thus extending the pattern with a common package *Semantics*. The DSML-specific components such as graphical animator and scenario builder were generated, based respectively on the SDMM and the EDMM. Thus, the development of such simulators was well-structured, and the approach provided a great uniformity and time saving. After a detailed presentation of its application, we gave a short presentation of another use of the *Executable DSML* pattern in the context of the TOPCASED project to integrate existing V&V tools. Finally, the proposed pattern has the intent to develop a general toolkit that combines the different V&V approaches presented in this paper. For instance, counter-examples provided by a model checker may be reused in the animators to visualize the fault.

[8] `http://www-01.ibm.com/software/awdtools/statemate`
[9] `http://www.uppaal.com`
[10] `http://www.mathworks.com/products/simulink`
[11] `http://ptolemy.berkeley.edu/ptolemyII`

Similarly to object-oriented modeling (OOM), this paper emphasizes the need for design patterns in metamodeling to capitalize experiences for recurring problems. An illustrative case is used as a guideline. For example, almost no systematic methods are available to help the language designer in the definition of an execution semantics and its tool support using metamodel. Moreover, in the same way that design patterns in OOM standardize the way designs are developed, design patterns in metamodeling should help the implementation of generative approaches.

The *Executable DSML* pattern introduces exciting perspectives and claims an engineering of semantics in MDE. The overall objective is to provide a flexible and general framework for model execution, supported by generic and generative tools to ease the development of DSML-specific tools. We are currently experimenting the pattern use to improve the specification and proof of correctness of semantic preserving model transformations. We are also extending the work on generation of model animators in several directions: the use of OCL to define sophisticated conditional breakpoints, the use of a temporal extension of OCL to define conditional breakpoints triggered by sequences of events and not only state contents, step-back facility (not included in the current execution framework) relying on bi-directional semantics implementation, the definition of an animator configuration model (extending the graphical editor configuration model of TOPCASED) to specify the decorators that must be added for a given semantics, and take into account of different MoCs in the same model with synchronization constructs.

Acknowledgment

This work was partially supported by the french government (DGCIS) through the FUI TOPCASED and ITEA2 OPEES project. The authors wish also to thank P. Farail and J.-P. Giacometi from Airbus for their helpful comments, the team of R. Faudou from Atos for their intensive development work in TOPCASED that provided the first validation of the pattern presented in this paper, and B. Berthomieu and F. Vernadat from the OLC team at the LAAS-CNRS for their cooperation in the definition of the model verification framework based on the pattern and on the TINA verification tool. And last but not least, the authors thank all careful proofreaders and especially B. Baudry for his stimulating proposals.

Contents

1 Introduction **3**

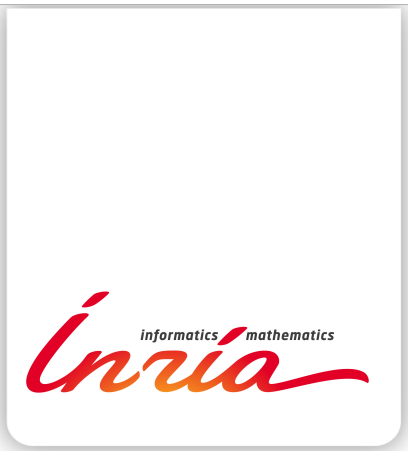
2 A Metamodeling Pattern for Model Execution **4**
2.1 Motivation 4
2.2 Structure 5
2.3 Participants 6
2.3.1 Domain Definition MetaModel (DDMM) 6
2.3.2 State Definition MetaModel (SDMM) 6
2.3.3 Event Definition MetaModel (EDMM) 7
2.3.4 Trace Management MetaModel (TM3) 7
2.3.5 Semantics 8
2.4 Consequences 8
2.4.1 Definition of the Semantics 9
2.4.2 Definition of the Execution-Related Tools 10

3 Integrating V&V by Interpretation: Application to *Graphical Model Animation* **10**
3.1 A MoC-specific Framework for Model Execution 11
3.2 A Generative Approach for Model Animation 12

4 Integrating V&V by Translation: Application to Model Checking **14**

5 Related Work **16**

6 Conclusion and Perspectives **17**



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399