A Design Pattern for Executable DSML

Benoit Combemale, Xavier Crégut, Marc Pantel

A Design Pattern for Executable DSML

Benoit Combemale, Xavier Crégut, and Marc Pantel

November 28, 2010

Abstract

Model executability is now a key concern in model-driven engineering, mainly to support early validation and verification (V&V). Some approaches have allowed to weave executability into metamodels, defining executable domain-specific modeling languages (DSML). Then, model validation may be achieved by 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 not available to help the language designer in the definition of such an execution semantics and related support 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 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 DSML. It favors flexibility and improves reusability in the definition of semantics-based tools for DSML. We illustrate how this pattern can be applied to V&V and models at runtime, and give insights on the development of generic and generative tools for model animators.

1 Introduction

The use of Domain Specific Modeling Languages (DSML) is a key feature of Model Driven Engineering (MDE). 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 DSML as a class diagram (called *metamodel*). Metamodels capture domain specific concepts and their relationships. Initially used for communication purpose, they are now an essential part for DSML specific tool development. By annotating such a metamodel, syntactic tools may be partially or totally generated (*e.g.,* GMF[1] and TMF[2] for graphical and textual model editors).

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

[2]The Eclipse Textual Modeling Framework, cf. `http://www.eclipse.org/modeling/tmf`

Model executability is now a key concern in MDE, mainly to support early validation and verification (V&V) in the development process. In this purpose, we are targeting systems which are time (i.e. partial order relation between events) and context dependent. Their models provide a representation of subject system behavior (*e.g.,* software, hardware or even physical systems). The execution of a system relates its evolution during time according to the change in its environment (external stimulus) and its internal state. Recently, several possibilities have been explored to implement the execution semantics of DSML (cf. Section 5 for related works). Basically, it proposes to map the abstract syntax, defined by the metamodel, to a semantic domain [1]. Most proposals translate models into an existing semantic domain in order to reuse available tools (*e.g.,* simulator, or model-checker). This usually leads to complex transformations that implement the semantic mapping but provide execution results only in the target technical space. Getting back the results in the source language is difficult and usually requires to extend its abstract syntax.

Otherwise, other approaches propose to weave executability into metamodels using an action language. Similarly, endogenous model transformations[3], including graph transformations, were widely investigated to give a declarative specification of the execution semantics. These approaches allow a more intuitive definition of executable DSML. The semantic domain is an extension of the abstract syntax, and the semantic mapping is defined using an action language. In doing so, one has indeed 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 (*e.g.,* simulator and model checker).

In both cases, the definition of DSML is facing today hard methodological problems in the specification of execution semantics. They are often empirically defined without any uniformity and underlying best practices. For example, the information capturing the state of a running model, 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 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 different 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 redefined without any capitalization, and without any guidances to ease the error prone and time consuming development.

In this paper we introduce a general, reusable and tool-supported approach to assist a DSML designer in the definition of an execution semantics and its related tools. This approach relies on capturing the different concerns involved in the definition of an executable DSML. Generic and specific information are reified, providing a behavioral design pattern to support executability into DSML: the *Executable DSML* pattern. It addresses several common use cases (like V&V and models at runtime) relying on execution semantics. Based on this pattern, generic and generative approaches are pro-

[3]The source and target models conform to the same metamodel.

posed to partially or even totally automate the definition of DSML tools for V&V and models at runtime. This work has been applied in the TOPCASED project[4], 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 DSML considered in TOPCASED. In this context, we firstly 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 DSML. We also present generative tools to automate the definition of domain specific 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, and to use models at runtime.

The remainder of this paper is structured as follows. Section 2 introduces the *Executable DSML* pattern illustrated with model simulation for validation. Section 3 details this use of the pattern and proposes both a MoC-specific model execution framework, and the associated generative approach to ease the definition of graphical model animators. Section 4 presents other applications for model execution, mainly for model verification and for adaptable systems using models at runtime. Section 5 summarizes related works. Finally, Section 6 concludes and gives insights on perspective.

2 A Metamodeling Pattern for Model Execution

Similarly to [2], we follow in this section the common pattern description format. We rely on the UML State Machines model animation example in order to introduce the requirements for model execution at a conceptual level [3]. This example is further detailed in Section 3 as a detailed example of the pattern use.

2.1 Motivation

MDE advocates the use of models and thus DSML to potentially derive an application from its requirements (or abstract description) that are expressed using those models. The semantics of the DSML is then enclosed (generally hard-coded) in the transformations and tools used during this process. Our purpose is to make the definition of the DSML semantics explicit, including the semantic domain and mapping as advocated in [1]. The *Executable DSML* pattern thus addresses the problem of the execution of models.

The designer of a model that describes a system behavior usually needs to animate it to check that it behaves as expected. Unfortunately, the model's 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 Machines define the concepts of `Region`, `State`, `Transition`, `Event`, etc. but lack the notions of active states in a region, and of fireable transitions (see figure 1). Attributes are also missing to store the sequence of events received by a state machine. Furthermore, during model animation, the designer

[4]Toolkit in OPen-source for Critical Applications & SystEms Development, cf. `http://www.topcased.org`

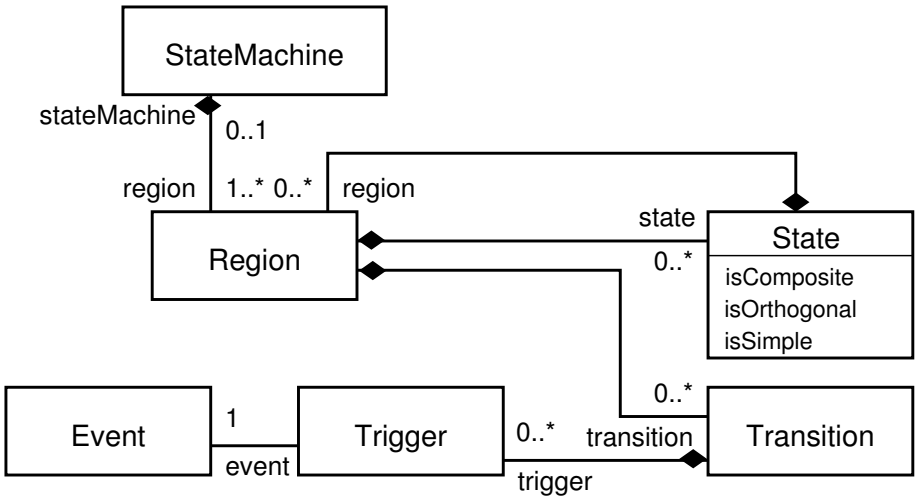

Figure 1: Subset of the UML StatMachine Metamodel. It only shows the elements that are used in this article.

has to emulate the behavior of the system environment through stimuli. In the case of UML State Machines, the 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 operational semantics. This reaction updates the execution related data according to the current state of the model and the received stimulus. Finally, the designer may want to replay the same execution, for example, to check whether or not defects have been corrected, 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.

We propose to reify those elements to make them explicit and manageable and thus be able to use them in generic and generative tools. We aim to provide flexibility, evolvability and interoperability in the semantics definition.

2.2 Structure

Fig. 2 shows the structure of the proposed pattern. It is built from four structural parts (detailed in the next subsection) that are woven together, that expresses the data related

to the DSML and its semantics, and a fifth one *Semantics* that provides the executable semantics relying on the previous four. As it is a metamodeling pattern, the structure shows dependencies between packages that represent metamodels. This pattern is architectural like *MVC* or *3-tiers*. It emphasizes the common structure that all metamodels for executable DSML should use in order to define the language semantics. The purpose is then to be able to define common, generic and generative tools relying on that architecture.

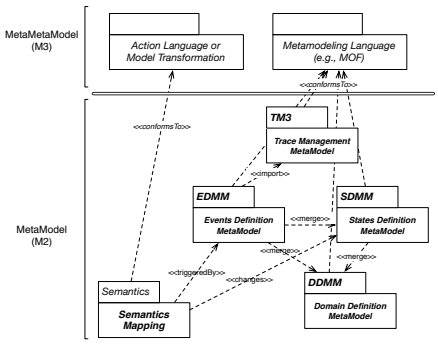

Figure 2: A pattern for model executability

2.3 Participants

2.3.1 Domain Definition MetaModel (DDMM)

It provides the key concepts of the language (representing the considered domain) and their relationships. It is the usual metamodel used to define the modeling language that may be enriched with structural constraints. It is the metamodel defined by standardization organisations. For instance, the UML metamodel defined by the OMG and the associated OCL constraints corresponds to the DDMM. It does not usually contain any execution-related information. For instance, the UML DDMM does not formalize the notion of *current state* nor *fireable transition*. Thus, even if a diagram describes the behavior of a system it is not executable as such.

2.3.2 State Definition MetaModel (SDMM).

During the execution of a model, additional data is usually mandatory for expressing the execution itself. Such data must be manipulated and recorded (in the form of metatype instances). For example, each active UML region must have at least an active state and a state machine must store the sequence of received events. These execution related data make up the SDMM. The SDMM is built on top of the DDMM. These data are related to the semantic domain. For example, the UML State Machines SDMM may add a reference from `Region` to `State` to record the active states of the region. Thus Fig. 2 uses the « `merge` » predefined package operator [4] to relate them.

2.3.3 Event Definition MetaModel (EDMM).

It reifies the concrete stimuli of the DSML as subtypes of the common abstract `RuntimeEvent` metatype. Concrete EDMM events add properties in relation to, and/or redefine properties of, events related to the formal semantics to be supported. As an illustration, a UML EDMM event consists of communicating an UML event to a state machine. When an UML event is handled by the state machine, it will fire the transitions that it triggers. The same merge package operator is again used to relate this part to the previous ones.

2.3.4 Trace Management MetaModel (TM3).

The common abstract `RuntimeEvent` metatype is defined in the TM3 and then imported in the EDMM. The «import» package operator from [4] is used to relate both metamodel parts and expresses the reuse of common MoC elements. The TM3 is independent of any DSML. It defines three main metatypes called `Trace`, `Scenario` and `RuntimeEvent` (cf. Fig. 3). `RuntimeEvent` is an abstract metaclass which reifies the concept of stimulus. This metatype is an abstraction for any kind of semantic related event. It should not be reduced to a mere hardware event, that it can represent, but also memory events like reading or writing, communication event like sending or receiving, clock event like a tick, function event like computation results given parameters, etc. It has executability-related features, like (partially ordered) dates of occurrence (i.e., symbolic representation of the time when the event occurs). `RuntimeEvent` instances fall into two categories, which are modeled by the `RuntimeEventKind` enumeration. Exogenous events are produced by the environment, while endogenous events are produced internally by the system in response to another event (cf. `cause` in Fig. 3). As stated by the OCL constraint in Fig. 3, a scenario is made of exogenous events whereas a trace corresponds to one possible execution of a scenario and is thus composed of any kind of events. A more sophisticated trace management metamodel or a "standard" one may be integrated in our pattern. The use of the UML Testing Profile [5] is under investigation. Any `RuntimeEvent` that triggers a semantics action that induces a state change should have a reference to its source and target states information in the SDMM.

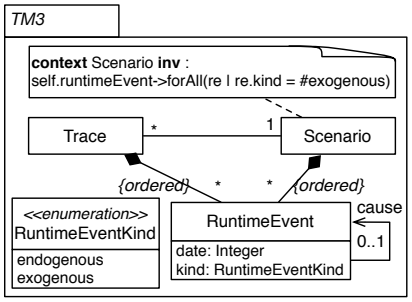

Figure 3: Content for a discrete-event TM3

2.3.5 Semantics.

The last and key participant is the *Semantics* package that abstracts both the semantic mapping [1] (DSML-specific part) and the interactions with the environment (i.e., the model of computation – 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 *Semantics* package 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. In this case, the four previous participants correspond to the semantics domain. Such kind of semantic mapping is usually used for model animation (Section 3) or models at runtime (Section 4), interactions being respectively handled by a model of computation or a control loop. On the other hand, the semantic mapping may be implicitly defined thanks to a translation to another language. Consequently SDMM and EDMM do not correspond to the semantic domain but helps in defining the mapping, and in getting results back (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 that may be equipped or not with the pattern.

2.4 Consequences

According to our pattern, an executable DSML is supported by an executable metamodel MM_x structured as three domain-specific parts and one MoC-specific part:

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

The MM_x metamodel 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 independent of any DSML. Thus, a semantics is defined by a triplet (SDMM, EDMM, *Semantics*). These different parts should not be defined independently one from the other in order to reduce the risks of inconsistencies. Redundancy analysis should also be conducted as in any other data model. 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-based tools.

Definition of the Semantics

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

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

- *It allows to factorize commonalities.* It eases the definition of a family of semantics for a single language or the semantics of a family of languages because the structure of MM_x allows to factorize commonalities. In particular, semantics variation points (like in UML) lead to different but similar semantics definitions. In most cases, SDMM and EDMM are the same and only the *Semantics* package 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 *Semantics* package are dependent of the user purpose during models execution. For instance, the user may prefer to carry out more abstract execution with fewer 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.* Relying on this pattern, we have proposed in [6] a particular one 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 is a minimal mandatory subset that allows to express the semantics relevant for the user, according to the substitutability principle [7].

- *The Semantics package definition is postponed.* As discussed in section 2.3, the semantic mapping and the interaction with the environment are not described in the pattern. According to the purpose of empowering a DSML with execution, the content of the Semantics package may be detailed. For example, Section 3 shows a MoC-specific framework for model execution. In most cases, for DSMLs, the architecture of the MM_x will ease the definition of the *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) which standard metamodel (DDMM) and semantics can be extremely complex. The introduction of purpose specific metamodels allows to ease the definition of the semantics of the subset of the language that the end user wants to assess.

Definition of the Execution-Based Tools

- *The formalization of pattern elements favors the definition of generic and generative execution-based tools.* Examples are given in Section 3.

- *Several models of computation (MoC) 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 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 events. Nevertheless, the *Semantics* can implement any MoC.

- *Cosimulation and models at runtime* can be integrated. The *Semantics* package can also be implemented as a wrapper over, either real physical systems which sensors and actuators are mapped to MM_x directly or through software layers, or existing softwares. Several DSML can also be integrated through shared data in their MM_x and synchronization/cooperation in their *Semantics* packages. This will be developed in Section 4.

- *It favors interoperability between the various semantics based tools for a given DSML.* Different kinds of tools may be based on the same executable DSML (*e.g.,* model animator, *model-checking* based verification tool, etc). The use of the same definitions thanks to the separation between MM_x and the *Semantics* package, allows to share data between tools (i.e. a counter example provided by a verification tool can be analyzed using a model animator). However, this relies only on structural similarities and also requires to assess the compatibility of both *Semantics* packages (i.e. by checking the bisimilarity of the transition relations, see Section 4). The definition of an explicit reference semantics for each DSML would be an even stronger point in that purpose.

3 Application: Model Execution for Graphical Animators

The previous pattern was designed and experimented with in the TOPCASED project. MDE 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 build graphical model editors. The first aim of model animation in TOPCASED was to be able to animate models built thanks to those graphical editors so that their designer may visually validate their models. The current graphical model animators for SYSML/UML State Machines and SAM[5] have been developed by ATOS ORIGIN relying on the *Executable DSML* pattern thus reaching a higher level of abstraction and reuse. The graphical user interface was initially hand-coded based on the corresponding graphical editor. Then those model animators have

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

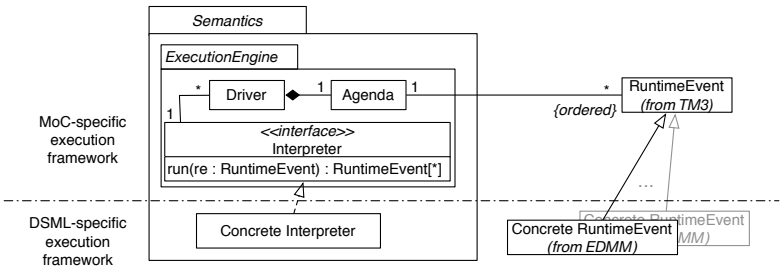

Figure 4: The TOPCASED Model Execution Framework

been refactored and several generative tools [8] have been developed to ease the development of model animators. This has drastically reduced the animators development time (only a couple of hours has been spent to retrofit/redevelop the UML State Machines animator using the already developed semantics). This section will present both the discrete event based implementation of the *Semantics* package and the generative approach used for the graphical interface.

3.1 MoC-specific Framework for Model Execution

The TOPCASED project aims at developing an open source CASE environment for the design of safety critical applications and systems. It addresses the domains of aeronautics, aerospace and more generally transportation systems. Representation of dynamic behaviors and of real-time features prevails in the description of such systems. During the design of their software parts, discrete (synchronous or asynchronous) modeling [9] is the most adequate way to represent them. Thus, in this context, only the discrete event MoC will be used for the model execution of any DSML.

We have applied the *Executable DSML* pattern for model execution and developed a framework (cf. Fig. 4) 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 DSML (top of Fig. 4) as it only depends on the TM3 (`RuntimeEvent`) and the `Interpreter` interface from the Semantics package which abstracts the transition function that will be provided by DSML specific Semantics packages. Its `run` method updates the dynamic information of the model according to one EDMM event and returns the list of generated endogenous EDMM events. For a particular DSML, one has to provide both the implementation of the `Interpreter` and the concrete EDMM events (bottom of Fig. 4).

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 EDMM 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 EDMM event from the agenda and asks the interpreter (`Interpreter`) to han-

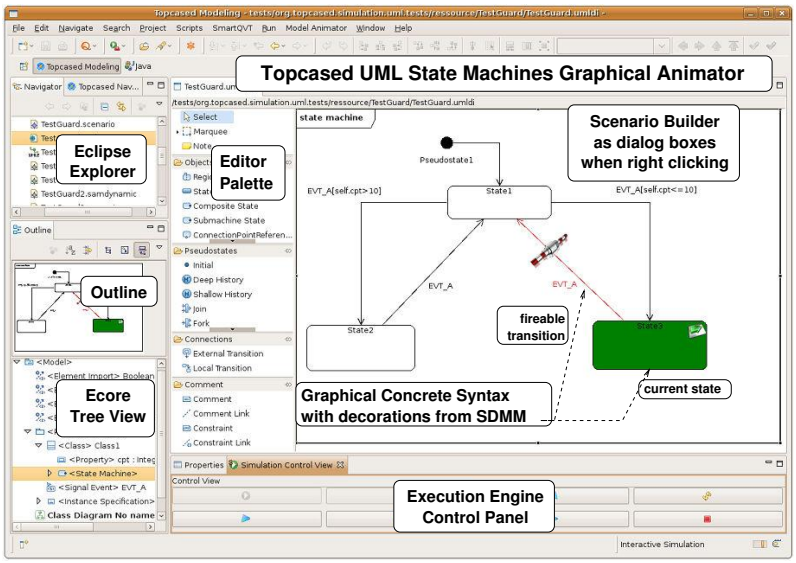

Figure 5: The TOPCASED Animator for UML State Machines

dle it. The generated endogenous EDMM events are then added to the agenda. The driver provides an API for external components that allows both batch and interactive execution.

For each execution semantics of a given DSML, the `Concrete Interpreter` must implement `Interpreter` (cf. Fig. 4). For the TOPCASED animators, the `run` method was initially hand-coded using JAVA and the EMF API. Then, it was implemented using SMARTQVT,[6] an open source transformation language that implements the OMG QVT specification [10] and 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. related works in Section 5).

3.2 Generative Technologies for Model Animation

Fig. 5 shows the components of TOPCASED Eclipse-based animators for the UML State Machines. The main view displays the graphical syntax. This one is decorated with dynamic information: current states are drawn with a green background, fireable transitions are in red with an icon. Other decorations like gauges or progress bars could be used. This visualisation has been chosen because it allows the user to view dynamic information directly on the domain model he has built. Nevertheless, other specific visualisation tools could be developed. The Eclipse Explorer, the outline of the diagram and the tree view of the underlying Ecore model are on the left side. The bottom view is the *Control Panel* that manages the execution (start/stop the simulation, move forth/back in the trace, etc.). During interactive animation, the user may insert

[6] Cf. `http://smartqvt.elibel.tm.fr/`

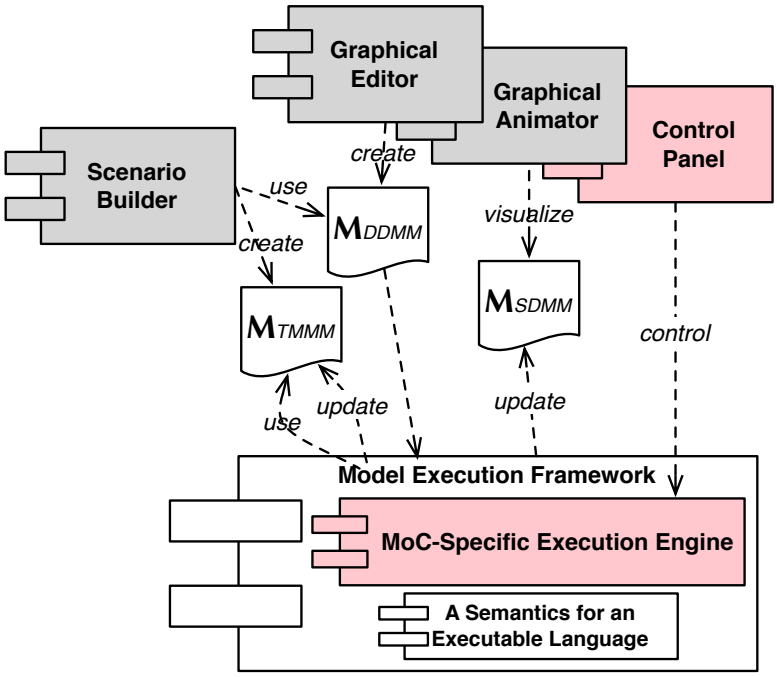

Figure 6: Interactions between Simulator Components

an EDMM event by clicking of the concerned graphical element. A dialog box prompts the user for the UML event name. This one is part of the *Scenario Builder* which purpose is to manage (*e.g.*, define, save and load) a scenario, before the simulation (based on the requirements, a test case generator, or a counter-example provided by a model-checker), or during the simulation, in an interactive fashion.

Fig. 6 presents 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, he can build a scenario (M_{EDMM}) with the *Scenario Builder*. Inside the editor, he can launch the animator. This triggers the creation and initialisation of the dynamic model (M_{SDMM}) from the M_{DDMM}. A scenario (M_{EDMM}) may be used to initialize the execution engine agenda (also a M_{EDMM}, the trace). Then, while the simulation is running and EDMM events are handled, the execution engine updates the M_{SDMM} and the trace (M_{EDMM}). Finally, as the dynamic model is modified, the associated graphical decorations are updated thanks to the EMF notifications.

Fig. 7 highlights which parts of the *Executable DSML* pattern are used to derive the animators' components. Some implementation insights are given hereafter (see [8] for further details). The graphical editor is built from the DDMM. It is generated from a TOPCASED configuration file that maps graphical elements to Ecore elements. The animator (in fact the visualisation of runtime 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 M_{SDMM} changes. It is currently

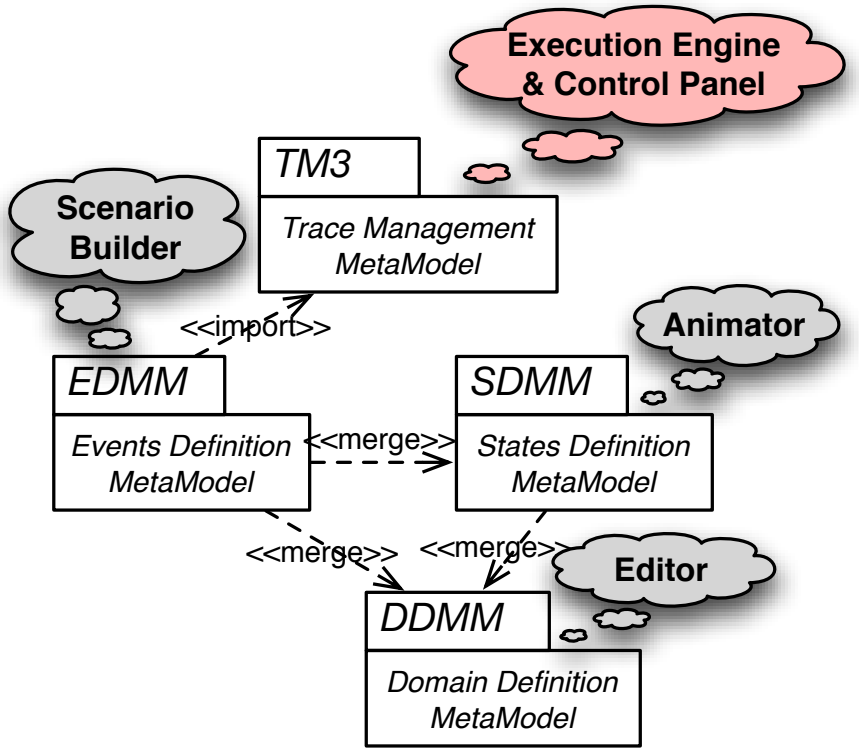

Figure 7: DSML-based Tooling for Model Simulation

hand-coded but the definition of a configuration file based on that of the editor is part of our future works.

The *Scenario Builder* relies on the EDMM (and thus on the TM3). It mainly consists of defining a concrete syntax for the EDMM of a given DSML. The underlying tooling can then be built using generative approaches such as GMF or TMF. In interactive mode, the dialog boxes are generated from the EDMM 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.

The example that we have provided is a simple case where the animation view is derived from the classical graphical model editor by adding additional decorators whose contents are based one the values in the SDMM. This case is quite common and allows to reuse the existing graphical editor and to generate parts of the decorators. But the pattern does not require to rely on the editor view and other views can be defined for the animation based on the content of the SDMM, EDMM and TM3.

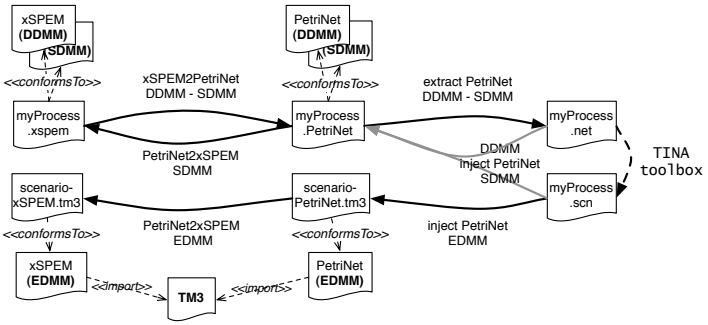

Figure 8: Pattern-Based Approach for Model Verification applied to XSPEM models

4 Other applications

We presented in Section 3 the use of the *Executable DSML* pattern for the development of design model validation tools. The provided *Semantics* package implements a kind of operational semantics which uses MM_x as the semantics domain. We now highlight how the elements of the *Executable DSML* pattern can be applied seamlessly to other use cases, both at system design and runtime and both for operational and translational kinds of semantics. Once again, the key point is the definition of the *Semantics* package for each kind of application.

4.1 Verification of executable models at system design

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 dynamic behavior during runtime, models are thus executable. We have focused our experiments on model checking, but other technologies could be integrated using the same approach. It 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 paths, in the graph. In case of failure it returns a state, or a path, that does not comply to the property or does not refine the previous model. These elements can be naturally represented using the metatypes in SDMM, EDMM and TM3; and implemented with the previous discrete event MoC. However, model checkers are very complex tools, thus it is more efficient to integrate existing ones in the *Semantics* package through back and forth model transformations between the DSML and the model checker dedicated modeling language (*e.g.,* Petri nets) as illustrated by Fig. 8.

This kind of integration is common practice since the early 80's, however it usually relies on ad-hoc solutions and suffers from the lack of processes, methods and tools. We experimented the use of the pattern in TOPCASED first with the XSPEM extension of SPEM [11], then with the LADDER PLC specification language; both were translated to PETRINET and verified using the TINA toolset [12, 13]. The structure

of the pattern allows to define guidelines for the specification and verification of the transformations. First, the MM_x architecture must be used for both source and target DSML. Then, the source DDMM is mapped to the target one (PETRINET DDMM in Fig. 8) to handle the structural parts of the model (transformation *xSPEM2PetriNet* – DDMM *part*). The SDMM provides the initial state (*xSPEM2PetriNet* – SDMM *part*). The state or path resulting from a verification failure is expressed using the generic TM3, the target SDMM and EDMM. The backward transformation will push the results in the end user domain using the generic TM3, the source SDMM (*PetriNet2xSPEM* – SDMM *part*) and EDMM (*PetriNet2xSPEM* – EDMM *part*). This implementation of the *Semantics* package is thus a kind of translational semantics to a semantics domain not related to the DSML. The injectors and extractors 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 or denotational semantics. These transformations are correct if the entailed relation between the events in both EDMM is a bi-simulation (see [14] for details).

4.2 Executable models at runtime

At runtime, we must consider the real system and its environment which encompass physical and hardware parts, operating systems, middlewares, classical softwares and end users. Executable models can be involved in the observation and management of the system and its environment; and even be the real system itself.

An executable observer model will only collect abstract information related to the evolution of the system. Sensors are used to observe the evolution of physical systems and their environment (*e.g.,* speed of a vehicle, position of a GPS, temperature of a CPU, load of a network, etc). The assignment of their values to attributes of SDMM and EDMM metatypes provides the user with the chosen abstract view. It can be done directly (*e.g.,* shared memories, interruptions, etc), or through operating systems, middlewares or APIs. SDMM conforming models are snapshots of the system evolution that are produced when events from EDMM conforming models occur. The physical behavior of the system thus embodies parts of the *Semantics* package. This package is thus not only a software model representing a mathematical abstraction of the real world, but also the application of physical laws.

An executable manager model will take decision based on the collected information, and issue stimuli toward the real system. Actuators are used to control the execution of physical systems (*e.g.,* engine fuel injection, brake fluid pressure pump, video mode selection, download protocols, number of active threads, etc). Mapping the actuators to elements from SDMM and EDMM provides the user with a model based control over the system behavior. Depending on the kind of actuators, metatypes or their attributes from the SDMM or the EDMM will be used. This mapping is done in the same manner as for sensors. We now mix two implementations of the *Semantics* package: a classical software one for the manager, and a real world physical one for the system feedback. This can be extended to as many implementations as DSML and physical parts are available in the system. These implementations must act in a more or less synchronized manner by sharing parts of their MM_x (i.e. use of a single or multiple

traces), and the associated *Agenda* and *Driver*. This point is a key design solution for the integration of the various parts of the *Semantics* package that will be explored in our future works.

Let us consider a simplified vehicle ABS system. The executable model can be written using a mixed control flow/data flow executable modeling language similar to Simulink. The car driver pushes the brake pedal which is a shared memory sensor, the corresponding attribute of the SDMM is updated regularly as part of the physical system semantics. The executable model is triggered periodically by the *Driver*, it consults this attribute and computes the required brake pressure depending on the vehicle speed and the wheel acceleration, the brake pressure is a shared memory controlled actuator whose value is converted by a Digital Analog Converter to a motor input in the brake fluid pump. The vehicle is driving on an icy road, the wheel acceleration modification sensors signals that the wheels are braking faster than expected. This sensor is an event producer that will push an urgent EDMM event to the executable model *Agenda*. The executable model will thus compute another value for the brake pressure to avoid the skidding of the vehicle.

Given an existing physical system and one of its usual DDMM model, the pattern can be used in two different ways: either the sensors and actuators are already available and the pattern allows to make the link with the models that the end user will observe and manage during the system execution; or the pattern allows to define what the user wants to observe and manage. It is then possible to derive the mandatory sensors and actuators that must be added to the physical system.

In the case of pure software system running on hardware, operating systems and middlewares, the executable model can fully embody the system. The SDMM and EDMM then represent the interaction with the user or the reasoning engine. The *Semantics* package implementation can rely on the discrete event MoC presented in Section 3. This last point was experimented in the MOCAS library[7], a UML State Machines interpreter that may be embedded in cell phones. The main difference the previously described implementation, is that the MoC uses physical time instead simulated one.

5 Related Work

5.1 Definition of the Semantics

The executable metamodeling pattern introduced in this paper relies on a semantics interface (cf. `Interpreter` in Fig. 4), allowing the systematic use of all available techniques to implement it. The extensive works on programming languages semantics have provided a taxonomy of the different techniques used to express a semantic according to different needs [15]. This concern is much more recent for modeling languages. Nevertheless, several possibilities have been explored to implement the executable semantics of DSML [14], and can be considered in the use of the *Executable DSML* pattern.

The first one is to use (the action language of) an executable metamodeling language to express directly the executable semantics like a set of operations for each

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

concept (e.g., Kermeta [16], xOCL [17], MOF action languages [18] or even JAVA with the EMF API). Such languages can offer high-level paradigms for the separation of concerns (*e.g.,* the aspect-oriented paradigm in Kermeta [19]), and then explicitly support the separation induced by the *Executable DSML* pattern.

A second way is to lay endogenous transformations over the abstract syntax that can be implemented using any model transformation language. As an example, [20] uses QVT [10] to express in-place rewriting rules that gradually compute the values of an OCL expression. TOPCASED currently rely on that approach using SMARTQVT.

Endogenous transformations have also been widely implemented through graph transformations [21], providing a declarative and rule-based technique to define an operational semantics (i.e. a kind of transition system). For example, AGG [22], ATOM3 [23], and GROOVE [24] can be used in that purpose. Kuske et al. [25] have used such an approach to define the executable semantics for some UML diagrams and their relationships. Hausmann [26] introduces the notion of dynamic metamodeling as a semantics description technique for Visual Modeling Languages. In [27], graph transformation rules are visually defined with collaboration diagrams. In [28], elementary transformations are represented as UML collaborations diagrams indicating the elements to add and/or to remove when it is applied. These transformations are embedded in the state of a UML activity diagram that controls the application order for the transformations. It thus looks like a graphical meta-programming language (called *Story Diagrams*) where actions are transformations based on graph rewriting and control structures are provided by the UML activity diagram.

The third technology to define the executable semantics of a DSML is called *translational semantics*. Unlike operational semantics, a translational semantics maps the model state into another (formally well defined) technical space. Thus, it relies on an existing semantics defined on the target technical space. It translates elements from the initial domain into elements of the formal target space. Thus, this translation provides the semantics to the initial domain. Translational semantics is used by the group pUML[8], called *Denotational Meta Modeling*, in order to formalize some UML diagrams [29].

As part of the MIC approach (Model-Integrated Computing), the ISIS laboratory promotes semantics anchoring [30] that is a kind of translational semantics. It consists of mapping the DSML elements into a semantics unit to define its semantics. We can notice that semantics units are defined using operational semantics, for example using Abstract State Machines (ASM). ASM are also used in [31] as an action language to operationally define the behavioral semantics of DSML, using an MDE framework that enables a transparent translation. ASM were also used in the AMMA platform to give the operational semantics of ATL and other modeling languages by Di Ruscio et al. in unpublished works. Similarly, [32, 33] aim at formally defining the semantics of a DSL by translating it to the Maude formal environment. The Maude tool is then used to express the operational semantics using rewriting rules. Recently, the same authors have used graph transformation in [34] to directly express the operational semantics through the abstract syntax of the DSL and then automatically translated into the Maude environment.

[8] *The precise UML group*, cf. `http://www.cs.york.ac.uk/puml/`

Numerous works use translational semantics, mainly to take advantage of the facilities and tools available in the target technical space (code generators, model-checkers, simulators, visualization tools, etc.). Nevertheless, an identified drawback is that it requires specialized knowledge and expertise in the target semantic domain by the DSL designer.

Operational and translational semantics may both be considered with the *Executable DSML* pattern. They correspond to the technology used to implement the mapping in the *Semantics* package. Taking advantage of these previous works, we claim for a flexible and general tool supported approach, relying on the existing semantics engineering. The main contribution of this work is to propose a metamodeling pattern for executable DSML relying on and combining the existing work on semantics definition.

5.2 Execution-Based Tools

Sadilek et al. have followed a similar purpose to ours in the EPROVIDE project: provide execution power to DSML [35, 36]. 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 information 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 must explicitly rely on APIs requiring a bit more work.

Conversely, in [37] the authors propose a reification of the dynamic information, according to the behavioral semantics specified thanks to UML Activity Diagrams. Nevertheless, only the dynamic information is reified, explicitly using inheritance to extend the domain definition and the UML Activity Diagram to implement the behavioral semantics. The *Executable DSML* pattern introduced in this paper proposes a more abstract and generic solution to implement executable DSML, considering the state of the art both to implement the behavioral semantics, and to extend the domain definition.

Soden et al. have proposed the MXF (Model eXecution Framework) eclipse project [38] 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.

Otherwise, several existing tools support edition and execution (mainly for simulation purpose) of models described in an automata-like notation. Let us mention, among the more popular ones *StateMate*[9], *Uppaal*[10], the Stateflow module in the *The MathWorks* Simulink framework[11], the Finite State Machine (FSM) model of computation of *Ptolemy II*[12], 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.

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

These existing tools have been defined for a given DSML. They embed their own semantics making 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 in our pattern), without any clear reification. Thus the dynamic information are generally deeply described in the tools, potentially with different choices or abstraction levels.

6 Conclusion and Perspectives

An executable metamodeling pattern introducing a clear separation of concerns into executable metamodels is proposed in this paper 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 and the real physical systems are represented through the Semantics package. Applying this pattern, we also propose an approach to help the DSML designer in the specification of an event-based execution semantics for DSML model animators. Finally, MoC-specific information such as trace and scenario are also reified in a separate metamodel called TM3 (standing for *Trace Management MetaModel*).

The *Executable DSML* pattern favors the definition of generic and generative technologies to ease the development of semantics-based tools for DSML such as model animators. After a detailed presentation of its application in the context of the TOPCASED project, we give a short presentation of other uses of our pattern for verification and models at runtime. The TOPCASED toolkit embed several DSML 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 our pattern. Based on the same discrete event MoC, a flexible execution framework has been defined relying on the TM3, thus extending the pattern with a common *Semantics* package. The DSML-specific components such as graphical animator and scenario builder were implemented, 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. The proposed framework has the intent to develop a general toolkit that combines the different V&V approaches presented in this paper. For instance, counter-example provided by model checker may be reused in the animators to visualize the fault.

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 a semantics engineering for MDE. The overall objective is to provide a flexible and general frameworks 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 extensions 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 models of computation in the same model with synchronization constructs.

Acknowledgments

This work was partially supported by the french government DGCIS through the FUI TOPCASED 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 Origin for their intensive development work in TOPCASED that provided the first validation of the model execution framework presented in this paper, F. Barbier and the MOVIES team from LIUPPA for the integration of the MOCAS framework inside TOPCASED relying on the pattern 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 thanks all careful proofreaders and especially B. Baudry for his stimulating proposals.

References

[1] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] OMG. *Unified Modeling Language (UML) 2.1.2 Superstructure*, 2007.

[4] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*, 2006.

[5] OMG. *UML Testing Profile 1.0 Specification*, 2005.

[6] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, Xavier Thirioux, and Francois Vernadat. A Property-Driven Approach to Formal Verification of Process Models. *LNBIP*, 2008.

[7] Marvin Minsky. Matter, mind, and models. *Semantic Information Processing*, pages 425–432, 1968.

[8] Xavier Crégut, Benoit Combemale, Marc Pantel, Raphael Faudoux, and Jonatas Pavei. Generative technologies for model animation in the TopCased platform. In *ECMFA 2010*, LNCS. Springer, June 2010.

[9] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. Mc Graw Hill, 2000.

[10] OMG. *MOF 2.0 Query/ View/ Transformation (QVT) Specification*, 2008.

[11] Reda Bendraou, Benoit Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an eXecutable SPEM2.0. In *14th APSEC*, Japan, December 2007. IEEE Computer Society.

[12] Benoît Combemale, Pierre-Loïc Garoche, Xavier Crégut, Xavier Thirioux, and François Vernadat. Towards a Formal Verification of Process Model's Properties SIMPLEPDL and TOCL Case Study. In *ICEIS*, 2007.

[13] Darlam Fabio Bender, Benoît Combemale, Xavier Crégut, Jean-Marie Farines, Bernard Berthomieu, and François Vernadat. Ladder Metamodeling and PLC Program Validation through Time Petri Nets. In *ECMDA-FA*, volume 5095 of *LNCS*, pages 121–136. Springer, 2008.

[14] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 4(6), 2009.

[15] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.

[16] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS'05*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[17] Tony Clark, Paul Sammut, and James Willans. SUPERLANGUAGES – Developing Languages and Applications with XMF. 2008.

[18] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. An action semantics for MOF 2.0. In *SAC'06*, pages 1304–1305. ACM, 2006.

[19] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218, may 2008.

[20] Slavisa Markovic and Thomas Baar. Semantics of OCL specified with QVT. *Software and System Modeling*, 7(4):399–422, 2008.

[21] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing, 1997.

[22] Gabriele Taentzer. AGG : A Graph Transformation Environment for Modeling and Validation of Software. In Springer, editor, *AGTIVE'03*, volume 3062 of *LNCS*, pages 446–453, 2003.

[23] Paolo Bottoni, Juan de Lara, and Esther Guerra. Action patterns for the incremental specification of the execution semantics of visual languages. In *VL/HCC*, pages 163–170. IEEE Computer Society, 2007.

[24] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink. Defining Object-Oriented Execution Semantics Using Graph Transformations. In *FMOODS'06*, volume 4037 of *LNCS*, pages 186–201. Springer, 2006.

[25] Sabine Kuske, Martin Gogolla, Ralf Kollmann, and Hans-Jörg Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In *IFM'02*, volume 2335 of *LNCS*, pages 11–28. Springer, 2002.

[26] Jan Hendrik Hausmann. *Dynamic Meta Modeling – A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, University of Paderborn, 2005.

[27] Gregor Engels, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *UML'00*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[28] T. Fischer, J. Niere, L. Torunski, and Albert Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In *TAGT'98*, volume 1764 of *LNCS*. Springer, 1998.

[29] Tony Clark, Andy Evans, and Stuart Kent. The Metamodelling Language Calculus: Foundation Semantics for UML. In *FASE'01*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[30] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson. Semantic anchoring with model transformations. In *ECMDA-FA'05*, volume 3748 of *LNCS*, pages 115–129, 2005.

[31] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages. *Autom. Softw. Eng.*, 16(3-4):415–454, 2009.

[32] José E. Rivera and Antonio Vallecillo. Adding behavioral semantics to models. In *EDOC 2007*, pages 169–180. IEEE Computer Society, October 2007.

[33] J. Raul Romero, Jose E. Rivera, Francisco Duran, and Antonio Vallecillo. Formal and Tool Support for Model Driven Engineering with Maude. *Journal of Object Technology, TOOLS EUROPE*, 6(9):187–207, 2007.

[34] José E. Rivera, Esther Guerra annd Juan de Lara, and Antonio Vallecillo. Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. In *SLE 2008*.

[35] Daniel A. Sadilek and Guido Wachsmuth. Prototyping visual interpreters and debuggers for domain-specific modelling languages. In *ECMDA-FA'08*, volume 5095 of *LNCS*, pages 63–78, 2008.

[36] Daniel A. Sadilek and Guido Wachsmuth. Using grammarware languages to define operational semantics of modelled languages. In *TOOLS'09*, volume 33 of *LNBIP*, pages 348–356. Springer, 2009.

[37] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA-FA 2007*, volume 4530 of *LNCS*, pages 157–171. Springer, 2007.

[38] Michael Soden and Hajo Eichler. Towards a model execution framework for Eclipse. In *BM-MDA'09*, pages 1–7. ACM, 2009.