Coping with Semantic Variation Points in Domain-Specific Modeling Languages

Florent Latombe*, Xavier Crégut*, Julien Deantoni†, Marc Pantel* and Benoit Combemale‡
* University of Toulouse, IRIT, Toulouse, France
first.last@irit.fr
† Univ. Nice Sophia Antipolis, CNRS, I3S, Inria, Sophia Antipolis, France
first.last@polytech.unice.fr
‡ University of Rennes I, IRISA, Inria, Rennes, France
first.last@irisa.fr

***Abstract*—Even if they exhibit differences, many Domain-Specific Modeling Languages (DSMLs) share elements from their concepts, notations and semantics. StateCharts is a well-known family of DSMLs that share many concepts but exhibit notational differences and many execution semantics variants (called Semantic Variation Points – SVPs –). For instance, when two conflicting transitions in a state machine are enabled by the same event occurrence, which transition is fired depends on the language variant (Harel original StateCharts, UML, Rhapsody, etc.) supported by the execution tool. Tools usually provide only one implementation of SVPs. It complicates communication both between tools and end-users, and hinders the co-existence of multiple variants. More generally, Language Workbenches dedicated to the specification and implementation of eXecutable Domain-Specific Modeling Languages (xDSMLs) often do not offer the tools and facilities to manage these SVPs, making it a time-consuming and troublesome activity. In this paper, we describe a modularized approach to the operational execution semantics of xDSMLs and show how it allows us to manage SVPs. We illustrate this proposal on StateCharts.**

I. Introduction

Domain-Specific Modeling Languages (DSMLs) provide user-friendly abstractions for domain experts who are not necessarily well-versed in the use of programming languages even when limited to domain-specific libraries (*i.e.*, API dedicated to a domain like security, fault tolerance, etc.). A DSML is called executable (xDSML) when it represents systems which evolve during time (*i.e.*, it captures the behavior(s) of the systems). Being domain-specific eases the design of complex software-intensive systems whereas being executable enables early verification and validation of the systems [1].

Semantic Variation Points (SVPs) are language specification parts left intentionally under-specified to allow further language adaptation to specific uses. SVPs must be dealt with either through further refinement of the language specification (in the case of UML, through stereotypes or profiles) or by making arbitrary choices in the implementation. The latter is most often the case when accommodating specific types of execution platform (*e.g.*, distributed, highly-parallel, etc.), allowing various implementations of the concurrency concerns (*e.g.*, parallelism or thread interleaving). For instance, the concurrency in CPython, the reference implementation of Python, is subject to the Global Interpreter Lock (GIL), which prevents multithreaded programs from taking advantage of multiprocessor systems; while Jython, the Java implementation of Python supports parallelism through the Java Virtual Machine (JVM). Another example is fUML [2], whose specification delegates the notions of time, communications and concurrency to the tool implementors. Tool vendors are thus responsible for specifying and documenting the implemented solution.

SVPs are usually identified informally in the DSML syntax and semantics specification documents. In the context of this paper, we will draw the difference between a *Language* (the specification of a syntax and of a semantics that may contain SVPs) and its *Dialects* (which implement a language, making choices about some – possibly all – SVPs of the language). Tools commonly only provide one dialect, thus constraining the end-user to work with the selected specific implementation of SVPs, which may not be the best-suited for their needs. Besides, it also complicates the cooperation between tools, since they may implement SVPs differently, giving a different semantics to the same syntax. Two engineers with different backgrounds may also assume different meanings for the same model, which impairs communication. Finally, large projects may need to use several dialects cooperatively, which means that this issue cannot be simply reduced to the choice of a unique tool: one dialect with an associated tool may be the best fit for a particular aspect of a system, but other ones may be better-suited for other aspects of the system.

This paper describes an approach to the specification of the execution semantics of an xDSML that eases the specification and management of SVPs. The approach is implemented in the GEMOC Studio[1]. It is illustrated with Finite State Machines (FSMs), a common formalism to represent complex system behaviors. StateCharts [3] is a common xDSML to model FSMs that exhibits many SVPs, as shown by a comparative study of its most popular versions [4].

Modern systems (IoT, etc.) increasingly rely on concurrency, and modern platforms provide more and more concrete concurrency facilities (many-core, GPU, etc.). xDSMLs must provide sophisticated concurrency concepts with well-defined semantics, to enable early concurrency-aware analyses, and

[1] http://www.gemoc.org

SVPs related to the concurrency concerns to allow refinements to specific platforms. But the concurrency concerns and SVPs of xDSMLs are usually implicit, embedded in the underlying execution environment or in the meta-languages used to define the xDSMLs. The GEMOC executable metamodeling approach makes explicit the concurrency aspects in an xDSML specification, relying on event structures [5] which define a partial ordering over a set of events. This paper advocates that this paradigm is not only suitable for specifying the concurrency concerns, but also for defining many of the concurrency-related SVPs by leaving some parts of the semantics under-constrained. Then, implementing these SVPs consist in reducing the partial ordering of the event structure with additional constraints.

This paper is structured as follows: Section II summarizes the StateCharts example. Section III describes the GEMOC executable modeling approach and illustrates how it is particularly suitable for the specification of SVPs. The implementation in the GEMOC Studio is described in Section IV. Finally, Section V gives related work and Section VI concludes and proposes insights on future works.

II. Semantic Variation Points of StateCharts

The StateCharts formalism is an extension of Finite State Machines (FSMs) with hierarchy, concurrency and broadcast communications [3]. A comparative study by M. Crane and J. Dingel [4] identifies and compares some of the existing StateCharts dialects. For the purpose of this paper, we will consider the original Harel Statecharts (denominated as Original) [6] and the UML StateCharts [7]. The study also classifies the differences between these dialects into three categories: notation (the least critical, a notation translation can ensure compatibility); well-formedness (when a formalism has an exclusive construct, model refactoring can most of the time make it compatible with the other formalisms); and semantics (the most critical one). For the purpose of this paper, we only consider the semantic differences between the dialects, and assume that notation and well-formedness issues have been solved beforehand.

A. Common Syntax of Statecharts

In StateCharts, an FSM is composed of *Regions* and *Events*. Each *Region* is composed of *States* and *Transitions* between these *States*. A *State* can be composed of *Regions*, thus defining a hierarchy between the *States*. Each *Region* has an initial *State*. A *Transition* can have a *Guard* (predicate expression) and an *Effect* (collection of behavior expressions named *Actions*). A *Transition* has a *Trigger* which is a reference to an *Event*.

B. Common Semantics of Statecharts

The bulk of the semantics is common to all three StateCharts dialects. When an *Event* occurs, it enables all the *Transitions* whose *Trigger* references the occurring *Event*. Depending on their *Guard* evaluation result, they may then be fired. When a *Transition* is fired, its source *State* is exited, its *Effect* is executed and its target *State* is entered.

C. Semantic Variation Points

We consider the three following SVPs:

- Simultaneous *Events*: Can different *Events* occur at the same time? In the original formalism, they can, and they are dealt with simultaneously (concurrently). The UML formalism adheres to the concept of run-to-completion (RTC) which means that *Event* occurrences are handled one by one (possibly with priorities), thus simultaneous *Events* are not allowed.
- Order of execution of *Actions*: When a *Transition* is fired, how are the *Actions* that compose its *Effect* executed? In *parallel* in the original formalism and in *sequence* in the UML version.
- Priorities of conflicting *Transitions*: When several *Transitions* are enabled by the same *Event* occurrence, how are they handled if their executions conflict (e.g. their application would lead to an inconsistent model state) ? In the original formalism, priority is given to the *Transition* which is highest in the hierarchy, while in UML, priority is given to the *Transition* which is lowest in the hierarchy.

Let us consider the example model from Figure 1, representing a simple music player.

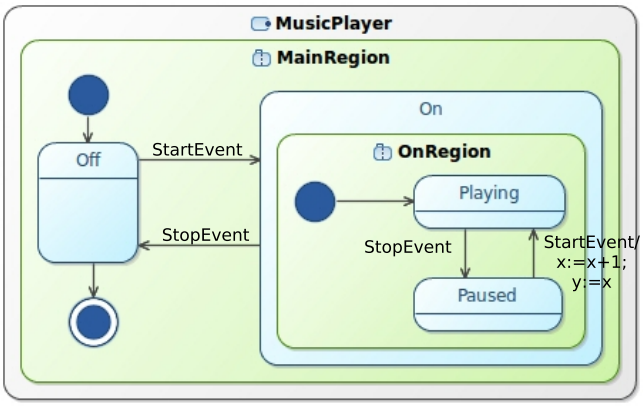

Fig. 1. Example model of Statecharts representing a simple music player.

The following scenarios illustrate the 3 targeted SVPs:

- When in *State* “Off”, “StartEvent” and “StopEvent” occur.
  - Original: they are dealt with concurrently, resulting in the *Transition* from “Off” to “On” being fired.
  - UML: not supported.
- When, in *States* “On” and “Paused”, the *Event* “StartEvent” occurs (assuming x and y both start at 0), triggering the firing of the *Transition* from “Paused” to “Playing”.
  - Original: when reaching *State* “Playing”, the values of x and y are respectively 1 and 0.
  - UML: when reaching *State* “Playing”, the values of x and y are respectively 1 and 1.

- When, in *States* “On” and “Playing”, the *Event* “StopEvent” occurs.
  - Original: the *Transition* from “On” to “Off” is fired.
  - UML: the *Transition* from “Playing” to “Paused” is fired.

So far, these differences have to be deduced from the documentation of the respective tools or specifications, or through tests of the tools. It is not explicit, when considering the model, what the associated version of the semantics is.

III. Contribution and application to StateCharts

This section describes the GEMOC executable metamodeling approach and how we propose to deal with SVPs. We illustrate it on StateCharts.

A. GEMOC Executable Metamodeling Approach

The language and modeling communities specify languages with an Abstract Syntax (AS), one or more Concrete Syntaxes (CSs) and mappings from the AS to one or more Semantic Domains (SDs) [8]. Based on this proposal and work from the concurrency theory community, previous contributions [9, 10] have proposed to cross-fertilize both approaches to specify *Concurrency-aware xDSMLs* by making the concurrency concerns explicit and suitable for analyses like determinism or deadlock freeness. The mapping between the AS and the SD, inspired from Structured Operational Semantics (SOS) [11], is split into two parts: the *Semantic Rules*, dedicated to sequential actions (not to be confused with StateCharts *Actions*) specified in an operational manner; and a declarative *Concurrency Model* that orchestrates the Semantic Rules. A *Communication Protocol* defines the relations between the Semantic Rules and the Concurrency Model. The class diagram from Figure 2 summarizes the architecture of the specification of concurrency-aware xDSMLs execution semantics in GEMOC.

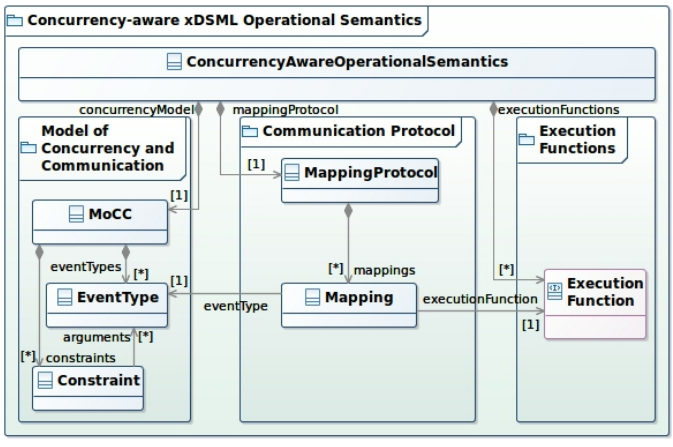

Fig. 2. Class Diagram of our approach towards concurrency-aware operational semantics of an xDSML.

The Semantic Rules specify the sequential evolutions of the model state at runtime (not to be confused with StateCharts *States*). The *Execution Data* (ED) represent at the language level the runtime state of the model. *Execution Functions* (EF) specify how the ED evolve during execution. For instance, a *Region* has a “currentState” reference in its ED which represents its current *State* during the execution. It is updated by the EF “fire” on *Transitions*.

The Concurrency Model is captured by the *Model of Concurrency and Communication* (MoCC). The MoCC is an *EventType* structure, specifying at the language level how, for a given model, the *event structure* [5] defining its *concurrent control flow* is built from a partial ordering over events (events of the event structure, not to be confused with StateCharts *Events*). This event structure represents all the possible execution paths of the *model* (including all possible interleavings of events occurring concurrently). Figure 3 shows the simplified event structure corresponding to the example from Figure 1. In this representation, a node is a *configuration* : an unordered set of event occurrences that have happened at this point in the execution. Event structures allow focusing on the concurrency, synchronization and the, possibly timed, causalities between actions. The actions themselves are opaque, thus the data manipulation they realize are abstracted away. In other words, it allows specifying the pure concurrent control flow of a model in order to ease reasoning on it. Note that in the case of StateCharts, the nature of the language (with 3 different possible inputs at every steps, depending on which events occur) makes the Event Structure very large and difficult to represent.

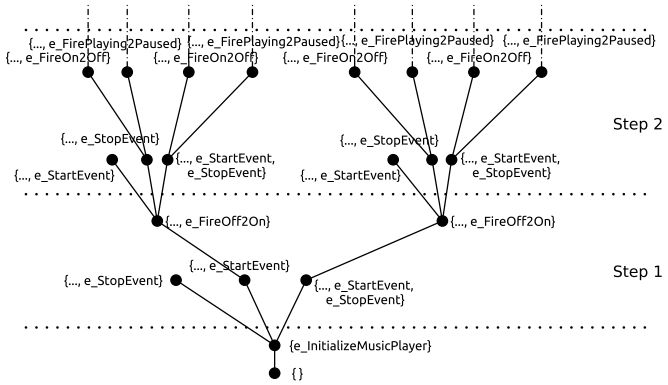

Fig. 3. Simplified event structure of the MusicPlayer example from Figure 1. “...” in a configuration represents the preceding configuration. The figure is limited to two steps of execution for representation purposes: every step offers three possibilities (not considering steps where none of the events occur), leading to three different execution paths.

The full semantics is built thanks to the *Communication Protocol* which maps some of the *EventTypes* from the MoCC to the EFs. This means that, at the model level, when an event occurs, it triggers the execution of the associated EF on an element of the model. For instance, if the Communication Protocol specifies that the *EventType* “et_FireTransition” is mapped to `Transition::fire()`, then the event “e_FireOn2Off” (instance of “et_FireTransition” for the *Transition* from “On” to “Off”, “On2Off”) triggers the execution of `On2Off .fire()`. Figure 4 shows the different concerns, at the model level, of our example: the bottom part of the Event Structure

(MoCC at the model level), the Execution Functions and the Communication Protocol (both at the model level). The

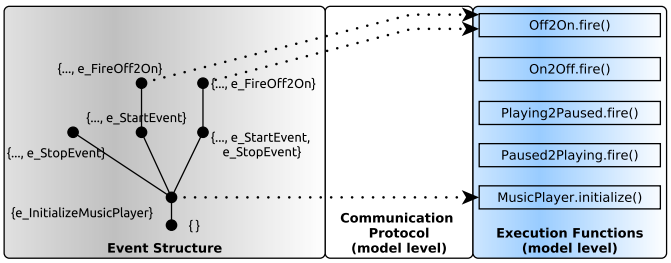

Fig. 4. Simplified view of the model-level specifications of our approach applied to the example StateCharts model of Figure 1.

Communication Protocol allows the MoCC and the Semantic Rules to remain independent, enabling modular changes in either part.

B. Coping with Semantic Variation Points

If several execution paths are allowed at a point in the event structure, it means that there is either *Concurrency* or *Conflict*. The first one means that other events are happening concurrently (interleaved or in parallel), in which case the execution paths will eventually merge. It does not mean that the executed model reaches the same state, but instead that in terms of pure control flow (independent of any data from the model) it is at the same point in the execution. Our example model is sequential, so there is no need for parallelism or interleavings. The second one means that there is a disjunction among the possible execution paths, which ultimately results in different final configurations of the event structure. Conflicts are a sign of nondeterminism in the semantics of the language, which means that either the language is indeterministic by intention, or that there is a SVP pertaining to the concurrency concern of the language. In the case of StateCharts, depending on the *Events* occurring, the execution will go one way or another, therefore there are many conflicts to represent the possible inputs (*Events* occurring).

SVPs can occur in any parts of the execution semantics. The ones in Semantic Rules target the model runtime state representation and/or evolution. For instance, changing a List into a Stack to have a LIFO policy instead of FIFO, or incrementing a value twice instead of once to double a ressource consumption. In StateCharts, a *Transition Effect* is a collection of *Actions* executed in parallel (in the original formalism) or in sequence (in UML). This SVP can be implemented in the EF that executes the *Effect* of the *Transition* being fired, with a parallel or sequential implementation. Another solution consists in providing both versions of the EF, but in changing the Communication Protocol to use the sequential one or the parallel one depending on the dialect we wish to use. The imperative nature of these implementations makes them complex to manage.

Thus, we propose to rely on the language's declarative MoCC to represent the superset of possibilities for all allowed concurrency-related SVPs, and to refine the MoCC for each dialect by removing the execution paths that do not correspond to the expected semantics. Nondeterminisms in the MoCC (resulting in conflicts in a model event structure) can be seen as potential SVPs. SVP implementations restrain the number of outgoing execution paths at a conflict point. In StateCharts (where a scenario, user, or environment drives the execution), there will be a lot of nondeterminisms left in the MoCC even after implementing the SVPs of the various dialects (to represent the possible inputs, *Events* being allowed to occur at every step of the execution). SVP implementations cannot add new *EventTypes* in the MoCC, as it would result in new execution paths that were not initially present in the language's MoCC specification.

For instance, let us consider the SVP concerning simultaneous *Events*. The part of the event structure corresponding to this situation is represented in Figure 5. For this particular

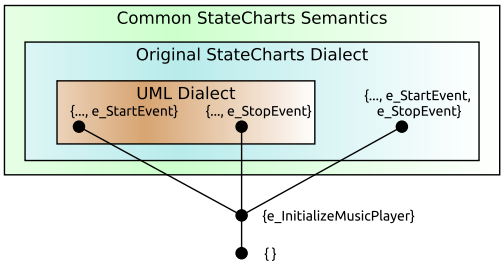

Fig. 5. Simplified event structure showing the Semantic Variation Point of simultaneous *Event* occurrences.

SVP, the UML dialect implements a subset of the original formalism's possibilities. Individual *Events* "e_StartEvent" and "e_StopEvent" are possible in both situations, but having both "e_StartEvent" and "e_StopEvent" is only allowed in the original formalism. The part of the MoCC responsible for implementing this SVP in UML is an extension of the part of the MoCC for the original formalism, with the addition of a constraint preventing simultaneous *Event* occurrences.

The SVP concerning conflicting *Transition* is different: there is a disjunction between the original formalism and the UML one. Figure 6 shows an example event structure of a situation where, in the *State* "On" and "Playing", the *Event* "StopEvent" (or both "StartEvent" and "StopEvent") occur. In this case, the

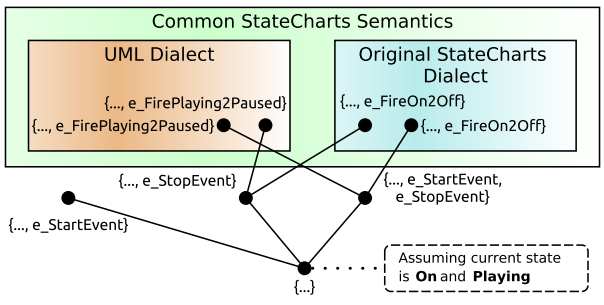

Fig. 6. Simplified event structure showing the Semantic Variation Point of conflicting *Transitions*.

original formalism gives priority to "On2Off", while UML

gives priority to "Playing2Paused". Therefore, the MoCC of the language representing the superset of possibilities for all allowed variants must allow both solutions. Each dialect then extends the MoCC of the language to add constraints resulting in the removal of some of the execution paths not corresponding to the implemented semantics.

The difference between specializing a language for a specific environment and implementing a Semantic Variation Point is blurry. SVPs sometimes represent adaptation points for a specific platform (distributed, highly parallel, ...). Therefore the event structure used to define the concurrency concerns in the semantics of xDSMLs is also a good fit for defining a language while still allowing dialects to be implemented by extending the concurrency model. SVPs can also be implemented in a modular way so that dialects are then realized by merging specific SVP implementations; similar to creating a new class in Aspect-Oriented Programming by extending an existing class and weaving existing aspects onto it.

IV. Implementation

The GEMOC executable metamodeling approach is implemented in a language workbench, the GEMOC Studio[2] based on previous work [10]. It is integrated in the Eclipse Modeling Framework (EMF) [12] to benefit from its large ecosystem.

The AS is specified with Ecore, the Eclipse Modeling Framework implementation of EMOF, and with the Object Constraint Language (OCL) for the static semantics. Both EMOF [13] and OCL [14] are standards from the Object Management Group (OMG).

The EFs are implemented with the Kermeta 3 Action Language (K3AL) [15], built on top of xTend [16]. K3AL allows the definition of aspects for Ecore metaclasses, allowing us to weave the additional classes, attributes, references and operation implementations specifying the ED and the EFs. K3AL, just like xTend, compiles into Java Bytecode and provides an executor based on the Java Reflection API to dynamically execute the EFs. Listing 1 shows the implementation of the `StateMachine.initialize()` Execution Function in K3AL.

Listing 1
Implementation of the Execution Function "initialize" of StateMachine in Kermeta 3.

```
@Aspect(className=StateMachine)
class StateMachineAspect {
 // Initializes each Region with its initial state
 def public void initialize() {
  _self.regions.forEach [ region |
    region.currentState = region.initialState
  ]
 }
}
```

MoCCML [17], a declarative meta-language designed to express constraints between events, is used to specify the MoCC. The definition of the *EventType* structure relies on the *Event Constraint Language* (ECL) [18], an extension of OCL which allows the definition of *EventTypes* for concepts from the AS. It can also use constraints defined in MoCCML to specify how the event structure at the model level is built. Listing 2 shows an excerpt from the MoCC of the fUML dialect of StateCharts. EventTypes are declared in the context of concepts from the AS, and the exclusion constraint implements the simultaneous events SVP for fUML. This relation ensures that events cannot occur simultaneously.

Listing 2
Excerpt from the MoCC of the UML dialect of StateCharts specified using ECL. The additional constraint "noSimultaneousEvents" implements the simultaneous events SVP.

```
import 'platform:/resource/org.gemoc.sample.
    statecharts.model/model/statecharts.ecore'

-- Defining the EventTypes in their context
context StateMachine
 def: mocc_initialize : Event = self

context Transition
 def: mocc_fire : Event = self

context StatechartEvent
 def: mocc_occur : Event = self

-- Constraint for UML
context StateMachine
  inv noSimultaneousEvents:
    Relation Exclusion(self.events.mocc_occur)
```

The MoCC is compiled to a *Clock Constraint Specification Language* (CCSL) [19] model interpreted by the TimeSquare [20] tool. Practically, computing the whole Event Structure is not doable because if the model is very large or highly parallel, then the exponential number of configurations (possibly infinite) makes it too costly to compute or too big to be usable. Instead, TimeSquare provides only the next set of possible configurations of the Event Structure. Due to its constraint-based approach, the runtime of CCSL provided by TimeSquare is called a *Solver*.

The Communication Protocol is specified using the *Gemoc Events Language* (GEL) [21]. Listing 3 shows an excerpt of the Communication Protocol of StateCharts. Mappings are implemented by what we call Domain-Specific Events (DSEs) which link an EventType (specified in ECL, see Listing 2), to an Execution Function).

At runtime, an Execution Engine written mostly in Java coordinates the K3AL interpreter, the GEL interpreter and the CCSL solver to execute a model. Figure 7 shows the sequence diagram for an execution step. First, the Execution Engine retrieves from the CCSL Solver the next set of possible configurations (scheduling solutions). Its heuristic

[2]http://gemoc.org/studio/

Listing 3
EXCERPT FROM THE COMMUNICATION PROTOCOL OF STATECHARTS SPECIFIED USING GEL.

```
DSE InitializeStateMachine:
  upon mocc_initialize
  triggers StateMachine.initialize
end

DSE FireTransition:
  upon mocc_fire
  triggers Transition.fire
end
```

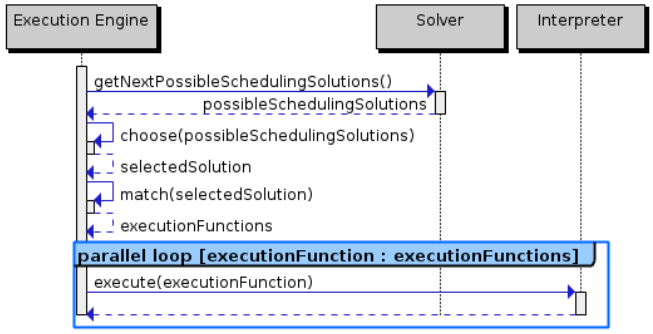

Fig. 7. Sequence Diagram representing one step of execution of a model conforming to a Concurrency-aware xDSML.

then selects one solution among the possible ones. A default implementation consists in letting the user do the selection to manage indeterministic situations manually when developing an xDSML. Based on the Communication Protocol, the set of EFs to execute is deduced from the selected solution. All the EFs are executed in parallel, resulting in an updated runtime state of the model.

Figure 8 gives the architecture of our implementation of StateCharts.

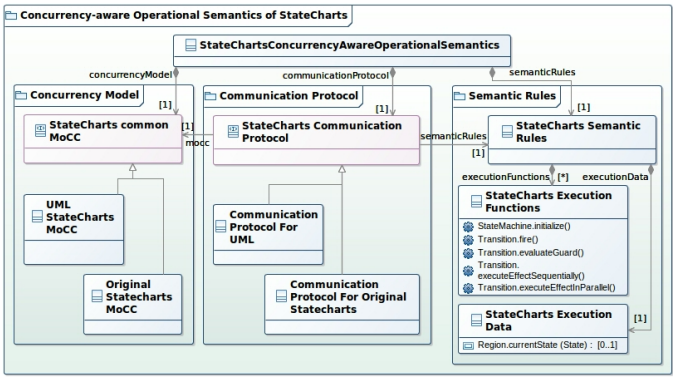

Fig. 8. Architecture of the different specifications constituting the concurrency-aware operational semantics of the different versions of StateCharts.

The modularization of our approach effectively allows the reuse of part of the operational semantics (in our case, of the ED and EFs and of parts of the Communication Protocol and of the MoCC) for the various dialects of StateCharts. To implement our second SVP, "Communication Protocol for UML" maps "Transition.executeEffectSequentially()" while for original StateCharts, "Transition.executeEffectInParallel()" is mapped. In "UML StateCharts MoCC", a constraint is added to exclude simultaneous occurrences of *Events* (as shown in Listing 2). Priority is given to the inner *Transition* in case of conflicting *Transitions* (third SVP) while the "Original StateCharts MoCC" gives priority to the outer *Transition*.

V. RELATED WORK

DMSL editors, also called Language Workbenches [22], such as Metacase's MetaEdit+ [23] or JetBrain's MPS [24] usually do not provide explicit means to handle the SVPs of the DSMLs. Existing executable metamodeling approaches such as xMOF [25] or Kermeta [26] usually only provide semantic variability through operations redefinitions and strategy design pattern uses. SVPs are a recurrent issue for executable metamodeling, in particular for executable UML. In [27], the authors draw inspiration from the notion of Genericity available in object-oriented GPLs such as Java or C++ to introduce a notion of template parameters at the metamodel level which can be bound either at the model level or at the metamodel level. This approach is exclusively at a structural level, with the assumption that behaviors are encapsulated in operations and triggered by operation calls. In our approach, we only consider the behavioral aspects and how the SVPs are implemented, assuming that the structural issues have been resolved beforehand. In [28] is introduced the notion of *inner semantics* of a language (semantics where the SVPs are left open) which corresponds, for us, to using the common MoCC of the language without extending it with SVP implementations. Our approach is closer to what was done for the SVPs of UML State Machines in [29], where the SVPs are reified in their own models to avoid tying the semantic choices to the implementing tool. The UML model to be executed is then transformed into a target UML model reflecting the semantic variations selected. In our approach, the SVPs are always specified in their own models at the language level since our Execution Engine is generic and able to execute any model conforming to a language described using our language workbench. Our approach also bears resemblance to [30], in which the authors have developed a framework to analyze the different variants of StateCharts thanks to pluggable semantics implemented in Java which parameterize their execution engine. In our case, the different semantic variants are models belonging to the language definition which are interpreted (after compilation for a specific model) by the generic execution engine making our approach generic and not tied to a family of languages.

VI. CONCLUSION AND PERSPECTIVES

Semantic Variation Points (SVPs) are usually poorly identified in language specifications, and their implementations are often hardcoded into the implementing tools, hindering the end-users from choosing their own variation of the language, particularly one that may be a better fit for a particular

class of problems. The GEMOC executable metamodeling approach separates the semantic mapping of xDSMLs to make explicit the Concurrency Model, completed by Semantic Rules describing how the runtime state of the model evolves, and by a Communication Protocol to connect both. Its event structure-based approach to the concurrency model eases the specification of nondeterministic situations, corresponding to potential SVPs. Implementing a SVP is then done by extending the concurrency model with additional constraints, resolving partially or totally nondeterminisms that were left in the language's Concurrency Model. The SVP implementations are thus weaved into the language definition, allowing the execution tool to remain independent from any arbitrary choice with regards to the SVPs. Future work will be concerned with adding the possibility to hinder parts of the concurrency model of the language from being extended by dialects. So far in our approach, every point of nondeterminism left in the concurrency model is considered as a potential SVP. But sometimes, the nondeterminisms left in the concurrency model are integrally part of the semantics of all variants allowed and should not be extended and resolved by dialects. We also plan to consider the case of SVPs which spread through the three units constituting the semantics (Concurrency Model, Semantic Rules or Communication Protocol) because so far, we have only considered SVPs which are contained in only one of these units. At last, we target the integration of these proposals with SVPs at the syntax level, both abstract and concrete; and to experiment the use of variability management techniques to make the management of language variants more explicit.

Acknowledgment

This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011).

References

[1] T. Kosar, N. Oliveira, M. Mernik, V. J. M. Pereira, M. Črepinšek, C. D. Da, and R. P. Henriques, "Comparing general-purpose and domain-specific languages: An empirical study," *ComSIS*, 2010.

[2] OMG, "fUML specification v1.1," 2013.

[3] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, 1987.

[4] M. L. Crane and J. Dingel, "UML vs. Classical vs. Rhapsody statecharts: not all models are created equal," *SoSyM*, 2007.

[5] G. Winskel, *Event structures*. Springer, 1987.

[6] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *TOSEM*, vol. 5, no. 4, pp. 293–333, 1996.

[7] OMG, "UML superstructure specification v2.4.1," 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

[8] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of "semantics"?" *Computer*, 2004.

[9] B. Combemale, C. Hardebolle, C. Jacquet, F. Boulanger, and B. Baudry, "Bridging the Chasm between Executable Metamodeling and Models of Computation," in *SLE2012*, ser. LNCS. Springer, Sep. 2012.

[10] B. Combemale, J. De Antoni, M. Vara Larsen, F. Mallet, O. Barais, B. Baudry, and R. France, "Reifying Concurrency for Executable Metamodeling," in *SLE 2013*, ser. LNCS. Springer-Verlag.

[11] G. D. Plotkin, "The origins of structural operational semantics," *The Journal of Logic and Algebraic Programming*, vol. 60, pp. 3–15, 2004.

[12] Eclipse Foundation, "EMF homepage," 2015. [Online]. Available: http://www.eclipse.org/modeling/emf/

[13] OMG, "MOF specification v2.5," 2015. [Online]. Available: http://www.omg.org/spec/MOF/

[14] ——, "OCL specification v2.4," 2014. [Online]. Available: http://www.omg.org/spec/OCL/

[15] DIVERSE-team, "Github for K3AL," 2015. [Online]. Available: http://github.com/diverse-project/k3/

[16] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[17] J. De Antoni, P. Issa Diallo, C. Teodorov, J. Champeau, and B. Combemale, "Towards a Meta-Language for the Concurrency Concern in DSLs," in *DATE'15*, Mar. 2015.

[18] J. De Antoni and F. Mallet, "ECL: the event constraint language, an extension of OCL with events," Inria, Tech. Rep., 2012.

[19] F. Mallet, "Clock constraint specification language: specifying clock constraints with uml/marte," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 309–314, 2008.

[20] J. De Antoni and F. Mallet, "Timesquare: Treat your models with logical time," in *Objects, Models, Components, Patterns*. Springer, 2012.

[21] F. Latombe, X. Cregut, B. Combemale, J. Deantoni, and M. Pantel, "Weaving Concurrency in eXecutable Domain-Specific Modeling Languages," in *SLE 2015*. ACM, Oct. 2015. [Online]. Available: https://hal.inria.fr/hal-01185911

[22] M. Fowler, "Language workbenches: The killer-app for domain specific languages," 2005. [Online]. Available: http://martinfowler.com/articles/languageWorkbench.html

[23] J.-P. Tolvanen and S. Kelly, "MetaEdit+: defining and using integrated domain-specific modeling languages," in *OOPSLA*, 2009.

[24] M. Voelter and V. Pech, "Language modularity with the MPS language workbench," in *ICSE*. IEEE, 2012.

[25] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, "xMOF: Executable DSMLs based on fUML," in *Software Language Engineering*. Springer, 2013, pp. 56–75.

[26] J.-M. Jézéquel, O. Barais, and F. Fleurey, "Model driven language engineering with kermeta," in *Generative and Transformational Techniques in Software Engineering III*. Springer, 2011, pp. 201–221.

[27] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard, "Templatable metamodels for semantic variation points," in *Model Driven Architecture-Foundations and Applications*. Springer, 2007.

[28] H. Grönniger and B. Rumpe, "Modeling language variability," in *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*. Springer, 2011.

[29] F. Chauvel and J.-M. Jézéquel, "Code generation from UML models with semantic variation points," in *Model Driven Engineering Languages and Systems*. Springer, 2005, pp. 54–68.

[30] D. Balasubramanian, C. S. Păsăreanu, M. W. Whalen, G. Karsai, and M. Lowry, "Polyglot: modeling and analysis for multiple statechart formalisms," in *ISSTA*. ACM, 2011.