Concurrency-aware Executable Domain-Specific Modeling Languages as Models of Concurrency

Florent Latombe, Xavier Crégut and Marc Pantel
Université de Toulouse, IRIT
Toulouse, France
first.last@enseeiht.fr

***Abstract*—To deal with the increasing complexity of modern highly-concurrent systems, the GEMOC approach for concurrency-aware eXecutable Domain-Specific Modeling Languages (xDSMLs) proposes to make explicit, in the operational semantics model, the concurrency concerns using a Model of Concurrency (MoC). This separation of concerns enables refinements (*e.g.*, for sequential or parallel execution platforms) and analyses (*e.g.*, to assess behavioral properties like deadlock-freeness) of the concurrency concerns. This approach initially provides only one MoC: Event Structures. But this MoC is not the best fit for all concurrency paradigms used in xDSMLs, resulting in complex models which are difficult to maintain or analyze. Moreover, extending the approach with new MoCs is complex: many elements must be integrated, and fit into the APIs used by the implementation. We propose to seamlessly define and integrate new MoCs through a recursive definition of the concurrency-aware xDSML approach, enabling the use of previously-defined xDSMLs as MoCs. This allows xDSMLs to always rely on an adequate MoC which also comes tooled with the generic execution and debugging facilities provided by the concurrency-aware approach. We illustrate this on the definition of fUML in the GEMOC Studio, an Eclipse-based language workbench.**

***Index Terms*—Domain-Specific Modeling Languages; Models of Concurrency; Executable Metamodeling**

I. Introduction

Modern complex systems (*e.g.*, Cyber-Physical Systems, Internet of Things, etc.) are highly-concurrent and executed on increasingly-parallel platforms (*e.g.*, many-core CPUs, GPGPU pipelines, etc.). eXecutable Domain-Specific Modeling Languages (xDSMLs) can be used to specify such systems, but in traditional executable metamodeling techniques, the concurrency concerns are spread throughout the whole semantics, making difficult its identification, refinement or analysis. A novel *concurrent executable metamodeling approach* has been proposed and refined in [1], [2], [3]: the GEMOC concurrency-aware xDSML approach. It proposes to make explicit, in the operational semantics of xDSMLs, the concurrency concerns using a Model of Concurrency (MoC) (*e.g.*, Petri nets [4], the Actor model [5], Event Structures [6], etc.). In this approach, the concurrency concerns are separated from the data concerns. The former are captured in the *Model of Concurrency Mapping (MoCMapping)*, which specifies how a MoC is systematically used for models conforming to the Abstract Syntax (AS) of the language (*i.e.*, it generates the model's concurrency concerns based on a MoC formalism). The latter are captured in the *Semantic Rules*, which extend the AS with the data and operational concerns of the semantics. Both are connected by a third model called the *Communication Protocol.*

The approach from [1], [2], [3] relies on the Event Structures MoC [6], in which partial orders [7] are defined over a set of events. The language-level model, the *MoCMapping*, is built using EventType Structures [8]. However, the use of a particular MoC for an xDSML is an important decision: it defines which formalism will be used to represent the concurrent aspects of the executable models. Depending on the MoC used, different concurrency-aware analyses (usually relying on existing external tools) may be performed to assess behavioral properties of the models. Moreover, not all MoCs are good fits for all xDSMLs. Depending on the xDSML's concurrency paradigm, MoCs may be more or less adequate. Providing only the Event Structures MoC effectively limits the approach, or complicates the modeling of some xDSMLs. Integrating new MoCs alongside Event Structures is complex, since it requires modifying the language workbench in which the concurrency-aware approach is implemented.

In this paper, our contribution consists in a recursive definition of the concurrency-aware xDSML approach, in which a previously-defined concurrency-aware xDSML can be used as the MoC of another xDSML, based on the *composite design pattern*. This greatly reduces the complexity of defining and integrating new MoCs into the approach. Moreover, the MoC can also benefit from any of the generic execution and debugging facilities provided for free by the language workbench implementing the concurrency-aware approach. The contribution to executable metamodeling techniques is thus twofold: we extend the existing concurrency-aware xDSML approach with the possibility to define and integrate new MoCs, effectively allowing xDSMLs to always rely on an adequate MoC; and we bridge the gap between MoCs (usually defined as formalisms dedicated to the modeling of concurrent systems) and software language engineering, effectively providing a common interface to MoCs (*i.e.*, as concurrency-aware xDSMLs).

The rest of this paper is organized as follows. First, in Section II, we illustrate the concurrency-aware xDSML approach on an example language, fUML. In Section III, we first show the importance of the adequacy of a MoC for an xDSML. We explain why directly integrating MoCs into the approach is complex and time-consuming. We then describe our recursive definition of the approach, illustrated by defining a new xDSML that we will use as the MoC of fUML. The upsides and limitations of our contribution are presented in Section IV. Section V focuses on our implementation in the GEMOC Studio's Language Workbench. Finally, we discuss related work in Section VI; then conclude and give perspectives for future work in Section VII.

II. Concurrency-aware xDSMLs Illustrated

We apply the seminal concurrency-aware xDSML approach [1], [2], [3] to the *Foundational Subset for Executable UML Models* (fUML) [9]: an executable subset of UML Activities.

A. Structural Elements

In Model-Driven Engineering (MDE), a language's Abstract Syntax is usually modeled as a *metamodel* enhanced with static semantics, expressed using the Meta-Object Facilities (MOF) and Object Constraint Language (OCL) from the OMG. Figure 1 shows an example fUML `Activity` composed of nodes (`ActivityNode`) of various natures, connected by edges (`ActivityEdge`). This example models drinking something while talking. The former consists in checking what is available on the table ("CheckTableForDrinks" returns at random "Coffee", "Tea" or "Neither"), and then executing one of the three branches depending on what was found on the table. In this example, concurrency takes place in the branches of

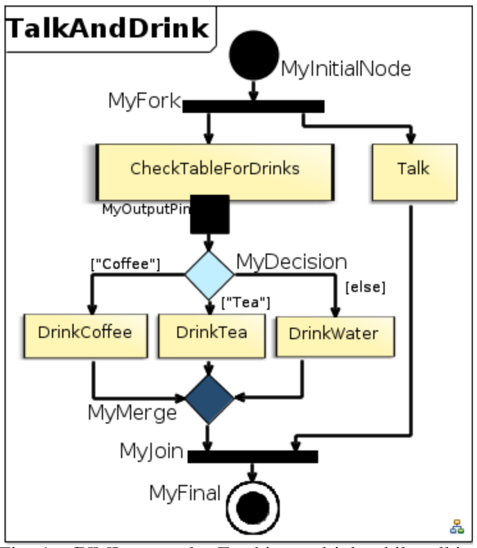

Fig. 1. fUML example: Fetching a drink while talking.

the *ForkNode*: the "Talk" node can be executed simultaneously with, or interleaved with, any of the nodes of the drinking part of the activity. Implementations usually hard code this decision, or rely upon the underlying execution platform. With the concurrency-aware approach, all the valid possibilities are explicitly specified, thus enabling the use of concurrency-aware analyses, allowing the management of semantic variations [2] or the refinement of the language for specific execution platforms (*e.g.*, sequential, highly-parallel, etc.).

B. Separation of Concerns in the Semantics

The concurrency-aware approach introduces a *separation of concerns* in the operational semantics [10] models.

The data and operational concerns are first captured in the **Semantic Rules**. The *Execution Data* define the dynamic data in the language (*i.e.*, current state of an automaton, current tokens in a place, current value of a variable, etc.) and how they evolve during the execution (*i.e.*, through fired transitions, executed instructions, etc.). In fUML, `Tokens` are created and consumed by the nodes. The Execution Data are thus the reference `currentTokens` weaved in the fUML metamodel, from the *ActivityEdge* concept to the *Token* concept. The execution of nodes is performed by an *Execution Function* weaved onto the *ActivityNode* concept, implemented by the `execute()` operation.

The concurrency concerns are then captured in the **Model of Concurrency Mapping**. So far, the approach only supports Event Structures so the MoCMapping is an EventType Structure model. In fUML, the core of the MoCMapping essentially consists in declaring an EventType representing the execution of an *ActivityNode*, and in constraining it such that for every *ActivityEdge*, its source is executed before its target. This language-level model is unfolded for each fUML model to generate the *Model of Concurrency Application* (MoCApplication, in our case, an Event Structure) representing the model's concurrency concerns.

Finally, both concerns are connected by another language-level model called the **Communication Protocol** that maps the *MoCTriggers* (*i.e.*, the abstract actions of the MoCMapping, that is the *EventTypes* of an EventType Structure) and the Execution Functions of the Semantic Rules. It also includes the *Feedback Protocol*, required in fUML for `DecisionNodes`, which was motivated and illustrated in [3].

Figure 2 shows an overview of the approach we just described, as a *Class Diagram*.

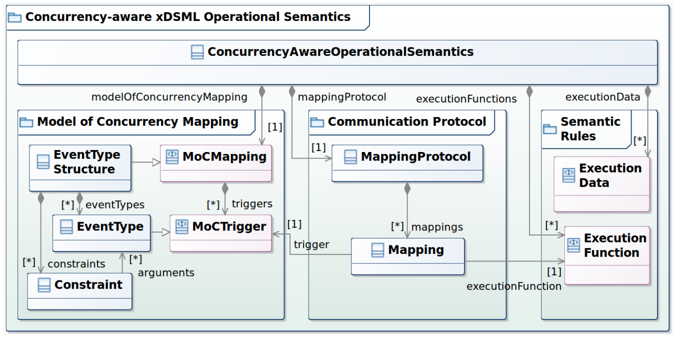

Fig. 2. Class diagram of the concurrency-aware xDSML approach.

III. Defining and Integrating Additional MoCs

This section first motivates the use of other MoCs besides Event Structures and shows why integrating new MoCs directly into the approach is complex and time-consuming. Then, we propose to define and seamlessly integrate new MoCs thanks to a recursive definition of the concurrency-aware xDSML approach. This proposal is illustrated by the use of a new MoC based on the notion of *Threads* instead of Event Structures to model fUML.

A. *Adequacy of Models of Concurrency*

Several criteria can be used to assess the adequacy of a MoC to model an xDSML's concurrency concerns. First, there is the conceptual proximity with the xDSML's concurrency paradigm, or with the systems modeled with the xDSML. In "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" [11], one of the reasons why a Scala code base integrates other MoCs besides the Actor model promoted by Scala, is because of *inadequacies of the actor model*. Using an inadequate MoC increases the chance of creating deadlocks and data races. Another important factor to account for is the language designer's experience with MoCs. This was also one of the reasons in [11] for the mixing of MoCs in Scala code bases. This can lead to an antipattern known as the Golden Hammer: "if all you have is a hammer, everything looks like a nail". These two criteria may complicate the use of a MoC, ultimately making the *MoCMapping* difficult to specify or maintain.

Moreover, one of the main benefits of the concurrency-aware approach lies in being able to formally verify properties of systems based on their *MoCApplication*. Which properties and verification technologies remains at the discretion of the MoC used. The formal method community has developed tools and methods for such verifications, *e.g.*, Petri nets [4] are commonly used for model-checking activities. This may be an important factor in selecting which MoC to use, for instance if particular safety properties must be enforced (*e.g.*, in critical systems).

B. *Integration Cost of Models of Concurrency*

To integrate new MoCs into the concurrency-aware approach, two metalanguages have to be considered. First, the MoC itself, to which the MoCApplication is conforming, with its editor, runtime, and possibly verification tools. Then, the MoCMapping formalism, to which the MoCMapping of an xDSML is conforming, with its editor and generator (used to produce the MoCApplication). Besides the merely technical difficulties of integrating these languages and their associated tools (provided the language workbench used is even open-source, or implemented with the abstractions permitting its extension in the first place), the main issue is that the notion of MoCMapping is the main novel artefact of the concurrency-aware xDSML approach. This means that for existing MoCs, the MoCMapping formalism has to be fully defined and tooled before being integrated into the approach, which is complex and time-consuming.

To overcome these issues, we propose a generic approach to define and seamlessly integrate new MoCs into the approach, enabling the tailoring of the MoCs used in the definition of concurrency-aware xDSMLs.

C. *A Thread-based MoC for fUML*

We illustrate our proposal with the definition and use of another MoC for fUML, whose semantics is given in the standard in both natural language and a reference implementation in Java[1]. Neither description is particularly adapted to the use of Event Structures, so the resulting MoCMapping was complex. Instead, Petri nets [4] can be considered, since it originally inspired UML Activity Diagrams. Another practical alternative is to rely on the notion of *Thread*: a conceptual unit for computation, also known as green threads, or user threads, as opposed to the operating system's threads (or kernel threads). The mapping between these two depends on the implementation of the runtime of the concurrency-aware xDSML approach. This MoC is convenient for fUML because it corresponds to the MoC provided in Java by the threading API, used for its reference implementation.

Let us detail how the *MoCMapping* of fUML can be specified using the notions of threads with a set of instructions, and the possibility for a thread to start another thread or to wait for the end of another thread. An *Activity* is mapped to the main thread in charge of the activity's *InitialNode* and *FinalNode*. The execution of nodes in sequence in an activity can be represented by a sequence of instructions in a thread. For every *ForkNode*, one thread is created per branch. It is in charge of executing the associated branch's nodes. Executing the corresponding *JoinNode* is possible when all these threads have finished executing their instructions.

Figure 3 shows the application of such a mapping on the example fUML *Activity* of Figure 1. The main thread consists in executing the *InitialNode* and the *ForkNode*, which starts the two sub-threads. It then waits for the sub-threads to complete, before allowing the execution of the *JoinNode* and *FinalNode*. In the first sub-thread, corresponding to the drinking part of the activity, the corresponding branch is executed as a sequence of instructions. After the *DecisionNode* and the evaluation of the guards, depending on the results retrieved, only one of the drinking nodes is executed before the *MergeNode* is executed. On the second sub-thread, there is just one node corresponding to talking.

[1]https://github.com/ModelDriven/fUML-Reference-Implementation

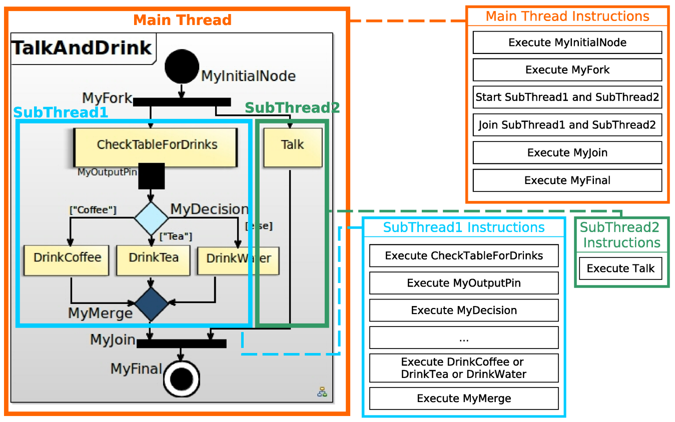

Fig. 3. Mapping the fUML example to threads.

D. Introducing a Recursive Definition of the Concurrency-aware xDSML Approach

More generally, we propose to leverage a previously-defined concurrency-aware xDSML as the MoC for another xDSML. This means that the concurrency concerns of a model are expressed as a model conforming to another xDSML, more appropriate to capture these concerns.

At the language level, this means that the *MoCMapping* as described in Section II is now a model transformation from the xDSML to the xDSML used as MoC. The model resulting from the transformation (the *MoCApplication*) is however *not semantically equivalent* to the original model: it only represents its concurrent aspects. The *MoCTriggers* are the *Mappings* (from the *Communication Protocol*) of the xDSML used as MoC. And since an element in a model may result in several elements in the resulting model (*e.g.*, *ForkNodes* are transformed into several "StartThread" instructions), we also need to be able to exploit the trace of the transformation to disambiguate the Communication Protocol of our xDSML.

More formally, we consider two concurrency-aware xDSMLs, $\mathcal{L}_{Domain}$ and $\mathcal{L}_{MoC}$. We will detail our approach to specify $\mathcal{L}_{Domain}$ using $\mathcal{L}_{MoC}$ as MoC, illustrated in the case where the former is fUML and the latter is an xDSML capturing the notions of threads and their instructions. $\mathcal{L}_{MoC}$ is considered as already defined, which means that it has been specified either as shown in Section II, or as is being shown in this section. Figure 4 gives an overview of our approach as a class diagram. It relies on two additional models presented in the "Concurrency-aware xDSML Recursive Definition" package of the class diagram. The rest of this section focuses on these two models.

1) Abstract Syntax Transformation: The first one is a **Transformation** from $\mathcal{L}_{Domain}$ to $\mathcal{L}_{MoC}$, denoted as $\mathcal{T}_{Domain \rightarrow MoC}$. It specifies how the concurrency concerns of $\mathcal{L}_{Domain}$ are represented using $\mathcal{L}_{MoC}$. It effectively corresponds to the *MoCMapping* of $\mathcal{L}_{Domain}$, as it maps the AS of $\mathcal{L}_{Domain}$ to the structure of the formalism used as MoC. For an input model $\mathcal{M}_{Domain}$, its output is $\mathcal{M}_{MoC}$.

For our example, we must first consider the definition of a concurrency-aware xDSML with the notion of `Threads`. In such an xDSML, we define a `ThreadSystem` as composed of *Threads*, with one of them considered as the main one. Each *Thread* has a number of *Tasks* which can be of different nature (execution, disjunction, conditional, etc.), in particular they can consist in starting or joining other threads. Inside a *Thread*, *Tasks* are executed sequentially. *Threads* are concurrent by nature, so if several are running at the same time, they can execute their instructions in parallel or in any form of interleaving. Joining on another thread waits for the selected thread to have all its instructions executed. *Disjunctions* are tasks for which only one of the two operands (*Tasks*) is executed, while *Conditionals* are executed if all their conditions (other *Tasks*) have been executed previously. The main *Mapping* of interest in the Communication Protocol of this Threading xDSML pertains to the execution of a *Task*, denoted as `ExecuteTask`. Once we have designed this language using the concurrency-aware approach, we can specify $\mathcal{T}_{fUML \rightarrow Threading}$. For our example model of Figure 1, the resulting model corresponds to the right half of Figure 3.

2) Trace of the AS Transformation: When a concept of $\mathcal{L}_{Domain}$ is transformed into several concepts of $\mathcal{L}_{MoC}$, we need a means to specify which one should be used as the trigger for the execution of the $\mathcal{L}_{Domain}$ concept. For instance, a *ForkNode* is transformed into as many *Tasks* as it has branches. We need to be able to define which *Task*'s execution will be mapped to the *ForkNode*'s execution. In order to do so, we thus need an additional language-level model, which consists in an excerpt from the metamodel of the trace of the AS Transformation. This excerpt is called the **Projections** of $\mathcal{L}_{Domain}$, designated as $\mathcal{P}_{Domain \rightarrow MoC}$. It specifies, for a concept of $\mathcal{L}_{Domain}$, into which concept(s) of $\mathcal{L}_{MoC}$ they are transformed (through $\mathcal{T}_{Domain \rightarrow MoC}$) and with which purpose(s), through labels. This allows us to specify, for instance, that the execution of the *ForkNode* is mapped to the execution of the first *Task* representing the execution of one of its branches.

This model is then used by the Communication Protocol of $\mathcal{L}_{Domain}$, in order to generate its model-level counterpart without ambiguities. This model, denoted as $\mathcal{P}_{fUML \rightarrow Threading}$ for fUML, as well as its use by the fUML *Communication Protocol* model, are both illustrated in Section V using our metalanguage implementations in the GEMOC Studio.

IV. Discussion

We discuss our contribution's benefits and drawbacks.

A. Modularity

The modularity of the initial approach is not disrupted. The data and concurrency concerns are still separated. In fact, it even favors the reuse of the AS and Semantic Rules of an xDSML by using different MoCs, for instance to compare two MoCs for the same language. Reversely, concurrency-aware xDSMLs can be reused as MoCs for other xDSMLs.

B. Concurrency-aware Analyses

Depending on the available tools, or expected behavioral properties, the language designer may choose to use one or another MoC for their xDSML. For instance, Petri nets [4] are commonly used to assess liveness or safety properties. Our approach also remains ultimately rooted in Event Structures. By transitivity, analyzing the concurrency concerns of a model can also be done through its underlying Event Structure [12]. Our contribution has thus provided an additional hook for potential analyses of executable models.

C. Facilitated MoCMapping Modeling

By facilitating the integration of new MoCs defined as xDSMLs, we allow concurrency experts to provide a large MoC library. Thus, we allow language designers to use the most appropriate MoC for the xDSML being modeled. This is similar to how DSLs are used for the dedicated abstractions they

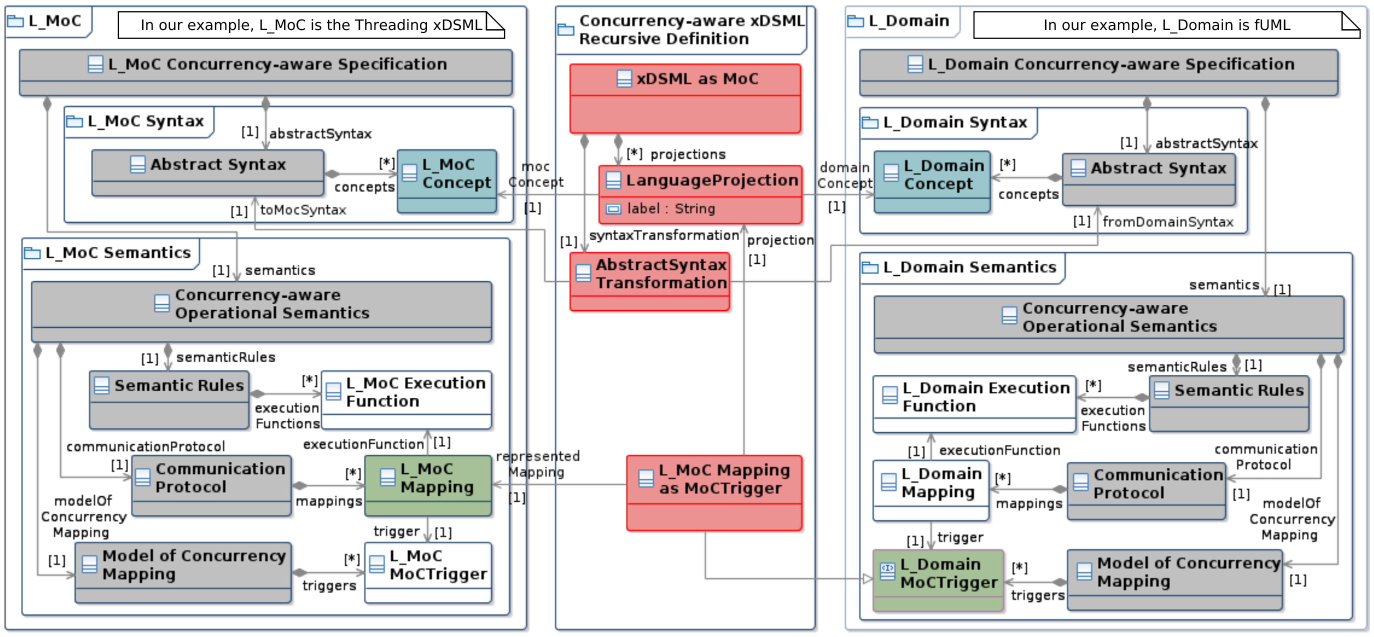

Fig. 4. Overview of the recursive definition of the concurrency-aware xDSML approach.

propose: some formalisms are more adapted for the modeling of some concurrency paradigms. This adequacy translates into an overall simpler or more maintainable MoCMapping model. Moreover, instead of having to learn and master an additional metalanguage for the MoCMapping (*e.g.*, EventType Structures), the language designer now only has to learn the Projections metalanguage (which is small and straightforward), while using a model transformation for the MoCMapping. Execution and debugging facilities of the concurrency-aware approach are also made available for free for the xDSML used as MoC.

D. *Unified Interface for MoCs*

MoCs are usually defined in many different manners, but usually quite “informally”, either as “formalisms”, or through language or framework constructs (*e.g.*, Erlang actors [13], Scala’s Akka actors [14]). Our contribution allows any concurrency-aware xDSML to be used as MoC. MoCs are thus all modelled as xDSMLs in a uniform way and it is the use made of an xDSML that determines whether it corresponds to a MoC or not. In other words, “MoC” is a *role* played by an xDSML, not its nature.

E. *Comparison with translational semantics*

Our contribution bears resemblance with translational semantics (*i.e.*, a language’s execution semantics is defined by a translation to another well-defined language). Indeed, we propose to define a transformation from $\mathcal{L}_{Domain}$ to $\mathcal{L}_{MoC}$: $\mathcal{T}_{Domain \rightarrow MoC}$. However, the *purpose* of this transformation is very different from that of translational semantics. In our approach, the source model ($\mathcal{M}_{Domain}$, conforming to $\mathcal{L}_{Domain}$) and the target model ($\mathcal{M}_{MoC}$, conforming to $\mathcal{L}_{MoC}$) are *not semantically equivalent*. $\mathcal{M}_{MoC}$ is only a representation of the concurrency concerns of $\mathcal{M}_{Domain}$, using $\mathcal{L}_{MoC}$ as a formalism; whereas in translational semantics, the *intention* of the transformation is to produce a semantically equivalent model. The data treatments done in the *Semantic Rules* of $\mathcal{L}_{Domain}$ are never translated in terms of concepts of $\mathcal{L}_{MoC}$, and only the concurrency concerns of $\mathcal{L}_{Domain}$ are transformed into $\mathcal{L}_{MoC}$. This is quite similar to the abstraction phase done during the modeling of a system in order to assess the correctness of the concurrency concerns using model checkers.

V. Implementation

Our proposal has been implemented in the GEMOC Studio[2], an EMF-based language workbench. The full execution of the example fUML Activity is available as a video at http://gemoc.org/exe16/. An archive file also provides the GEMOC Studio with our implementation of fUML based on Threads, as well as the source files for these xDSMLs.

A. *Existing Elements*

The Abstract Syntax is captured as an Ecore metamodel. The Semantic Rules can be weaved into the AS by defining aspects using the Kermeta 3 Action Language (K3AL) [15]. EventType Structures are specified using a combination of MoCCML [16] and ECL [8]. The Communication Protocol is specified using a dedicated metalanguage called the GEMOC Events Language (GEL) [3].

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

B. Projections, Transformation and Communication Protocol

To specify $\mathcal{P}_{Domain \to MoC}$, we have designed a small dedicated metalanguage. Listing 1 shows the projections for fUML, $\mathcal{P}_{fUML \to Threading}$. fUML *ActivityNodes* are transformed into *Tasks*. When a *Task* is executable, then the corresponding *ActivityNode* (if there is one) is also executable. In the same manner, some *Tasks* correspond to the evaluation of the guard of an *ActivityEdge*, while some others represent the fact that a branch may or may not be executed, based on the result of its guard (more details in [3]).

Listing 1. fUML projections on Threading.

```
Projections :
 LanguageProjection Proj_Execution:
  fuml.ActivityNode projected onto threaded.Task end
 LanguageProjection Proj_Evaluation:
  fuml.ActivityEdge projected onto threaded.Task end
 LanguageProjection Proj_MayExecute:
  fuml.ActivityEdge projected onto threaded.Task end
 LanguageProjection Proj_MayNotExecute:
  fuml.ActivityEdge projected onto threaded.Task end
end
```

$\mathcal{T}_{Domain \to MoC}$ can be specified using any Model to Model (M2M) transformation language [17] and any general-purpose programming languages relying on an appropriate model management library such as Java with EMF's APIs. However, the transformation must not only transform $\mathcal{M}_{Domain}$ into a corresponding $\mathcal{M}_{MoC}$, but it must also generate the model-level models of $\mathcal{P}_{Domain \to MoC}$, denoted as $\mathcal{P}_{DomainModel \to MoCModel}$, and used during the generation of the model-level Communication Protocol. Indeed, we have extended GEL with the capacity to reference projections from $\mathcal{P}_{Domain \to MoC}$ when defining the *Mappings*. Listing 2 shows the *Communication Protocol* for our implementation of fUML.

Listing 2. Communication Protocol for fUML.

```
DSE ExecuteActivityNode:
upon event ExecuteTask with Proj_Execution
triggers ActivityNode.execute blocking end

DSE EvaluateGuard:
upon event ExecuteTask with Proj_Evaluation
triggers ActivityEdge.evaluateGuard returning bool
 feedback: // Presented in more details in [3].
   [bool] => allow event ExecuteTask with Proj_MayExecute
  default => allow event ExecuteTask with Proj_MayNotExecute
 end
end
```

VI. Related Work

The concurrency-aware xDSML approach we have extended brings together two fields of research.

First, language Workbenches [18], such as MPS [19] or MetaEdit+ [20] traditionally focus on the language syntaxes. Executability of DSMLs is provided using "meta-programming" like Rascal [21] or Spoofax [22]; or "executable metamodeling" like xMOF [23] or the K Framework [24]. In both cases, the concurrency concerns are either embedded in the metalanguages provided by the approach used, or implicitly inherited from the underlying execution platform. In concurrency-aware xDSMLs, they are made explicit *at the language level* through the use of a MoC.

Then, concurrency theory has studied Models of Concurrency like Petri nets [4], the Actor model [5] or Event Structures [6] for a long time [25]. Different MoCs typically split a task in different manners and the communication and collaboration between the computing entities (*e.g.*, threads, actors, etc.) is done in different ways (*i.e.*, in terms of data-sharing, scheduling, cooperation, etc.). A unification of MoCs has been proposed through the use of the "tagged signal" model [26] or of category theory [27], but they do not focus on defining and integrating new MoCs. For instance in Ptolemy[3] [28], adding new MoCs (called "directors") requires complex changes in its source code.

In "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" [11], the authors analyze Scala code-bases and interview their developers to determine how often and why the Actor MoC was not the only one used. Reasons were categorized into three groups: a) inadequacies of the actor library, b) inadequacies of the MoC, and c) inadequate developer experience. Concurrency-aware xDSMLs move a) and c) away from the system designer to the language designer, greatly reducing the number of users concerned with the correct uses of MoCs. Thanks to our contribution, b) is removed as any MoC can be used for any xDSMLs, as long as they have been modeled as concurrency-aware xDSMLs. This enables the language designer to use the best MoC for the concurrency concerns of each xDSML.

Finally, we would like to point out the similarity of our proposal with the design pattern identified by Bran Selic as the "Recursive Control" pattern in [29]. In our case, $\mathcal{L}_{MoC}$ and its runtime is the internal control for $\mathcal{L}_{Domain}$ and its runtime. Since $\mathcal{L}_{MoC}$ itself can be specified using another xDSML, this effectively corresponds to an application of the "Recursive Control" pattern.

VII. Conclusion and Perspectives

The concurrency-aware xDSML approach advocates making explicit, in the operational semantics, the concurrency concerns using a Model of Concurrency. However, only one MoC, namely Event Structures, was provided initially. We have shown that not all MoCs are good fits for all concurrency paradigms, and that manually integrating new MoCs into the approach is complex and time-consuming as it requires two metalanguages (for the language and model levels) as well as their tools.

To ease this activity, we have proposed an extension which enables the use of previously-defined concurrency-aware xDSMLs as MoCs. This contribution relies on two main changes in the language models: an *abstract syntax transformation* to specify how the xDSML's concurrency concerns are encoded in the xDSML used as MoC; and

[3]http://ptolemy.eecs.berkeley.edu/

the *Projections*, a part of the transformation metamodel, that accounts for cases where an xDSML concept is transformed into several MoC concepts. The former can be specified using any classical model transformation languages, while we have devised a dedicated metalanguage for the latter. Our contribution has been implemented in the GEMOC Studio, an Eclipse-based language workbench, and illustrated on fUML defined using an xDSML capturing the notions of threads with instructions. We have made available a video showing the execution of an example fUML Activity, and the sources for the two xDSMLs. By defining MoCs as concurrency-aware xDSMLs, we give them a systematic structure, enabling their use at the language-level for the modeling of other concurrency-aware xDSMLs. In particular, it enables the use of the MoC that is the best fit for the concurrency paradigm of the language being developed. It also eases the development of an xDSML, since the model-level application of the MoC is simply a model conforming to an xDSML, that can be executed, debugged and animated like a regular model. Moreover, different formal behavioral properties can be assessed on executable models depending on the MoC used by the language.

Although any concurrency-aware xDSML can be used as a MoC, the concurrency theory community has studied in details a large number of MoCs such as the Actor Model [5] or Petri nets [4]. We plan to provide reference implementations for these ones in a MoC standard library, including Event Structures to bootstrap our approach. Afterwards, existing tools around these formalisms, such as model-checking tools, can be integrated seamlessly in our approach. Still, the approach remains rooted in the seminal MoC (*i.e.*, Event Structures), so higher-order transformations could be used to verify domain properties using the underlying MoC, while translating their results back to the domain [30]. Finally, even though we have considered concurrency-aware xDSMLs as language models for the implementation of more efficient tools, we plan to study how code generation or scheduler synthesis could be used to generate more efficient implementations of concurrency-aware xDSMLs.

Acknowledgments

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

References

[1] B. Combemale, J. Deantoni, M. Vara Larsen, F. Mallet, O. Barais, B. Baudry, and R. France, “Reifying Concurrency for Executable Metamodeling,” in *SLE’13*.

[2] F. Latombe, X. Crégut, J. Deantoni, M. Pantel, and B. Combemale, “Coping with Semantic Variation Points in Domain-Specific Modeling Languages,” in *EXE 2015*. Ottawa, Canada: CEUR, 2015.

[3] F. Latombe, X. Crégut, B. Combemale, J. Deantoni, and M. Pantel, “Weaving Concurrency in eXecutable Domain-Specific Modeling Languages,” in *8th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*, 2015. [Online]. Available: https://hal.inria.fr/hal-01185911

[4] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, no. 4, 1989.

[5] G. A. Agha, “Actors: A model of concurrent computation in distributed systems.” DTIC Document, Tech. Rep., 1985.

[6] G. Winskel, “Event structures,” in *Petri Nets: Applications and Relationships to Other Models of Concurrency*, ser. LNCS, 1987.

[7] V. Pratt, “Modeling concurrency with partial orders,” *International Journal of Parallel Programming*, vol. 15, no. 1, pp. 33–71, 1986.

[8] J. Deantoni and F. Mallet, “ECL: the event constraint language, an extension of OCL with events,” Inria, Tech. Rep., 2012.

[9] OMG, “fUML specification v1.1,” 2013. [Online]. Available: http://www.omg.org/spec/FUML/

[10] G. D. Plotkin, “The origins of Structural Operational Semantics,” *The Journal of Logic and Algebraic Programming*, 2004.

[11] S. Tasharofi, P. Dinges, and R. E. Johnson, “Why do scala developers mix the actor model with other concurrency models?” in *ECOOP 2013*. Springer, 2013.

[12] F. Mallet and R. De Simone, “Correctness Issues on MARTE/CCSL constraints,” *Science of Computer Programming*, vol. 106, pp. 78–92, Aug. 2015. [Online]. Available: https://hal.inria.fr/hal-01257978

[13] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent programming in ERLANG*. Citeseer, 1993.

[14] M. Gupta, *Akka essentials*. Packt Publishing Ltd, 2012.

[15] DIVERSE-team, “Github for k3al,” 2016. [Online]. Available: http://github.com/diverse-project/k3/

[16] J. Deantoni, P. Issa Diallo, C. Teodorov, J. Champeau, and B. Combemale, “Towards a Meta-Language for the Concurrency Concern in DSLs,” in *DATE*, 2015.

[17] OMG, “QVT specification v1.2,” 2015. [Online]. Available: http://www.omg.org/spec/QVT/

[18] Language Workbenches Challenge, “Comparing tools of the trade,” 2014. [Online]. Available: http://www.languageworkbenches.net/

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

[20] S. Kelly, K. Lyytinen, M. Rossi, and J. P. Tolvanen, “MetaEdit+ at the age of 20,” in *CAiSE*. Springer, 2013.

[21] T. Van Der Storm, “The Rascal Language Workbench,” 2011.

[22] L. C. Kats and E. Visser, “The spoofax language workbench: rules for declarative specification of languages and ides,” in *ACM Sigplan Notices*, 2010.

[23] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, “xMOF: Executable DSMLs based on fUML,” in *SLE*, 2013.

[24] G. Rosu and T. F. Serbanuta, “K overview and simple case study,” in *Proceedings of International K Workshop (K’11)*, 2014.

[25] M. Nielsen, “Models for concurrency,” in *Mathematical Foundations of Computer Science*, 1991.

[26] E. Lee, A. Sangiovanni-Vincentelli *et al.*, “A framework for comparing models of computation,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1998.

[27] M. Nielsen, V. Sassone, and G. Winskel, *Relationships between models of concurrency*. Springer, 1994.

[28] C. Ptolemaeus, *System Design, Modeling, and Simulation: Using Ptolemy II*. Ptolemy. org Berkeley, CA, USA, 2014.

[29] B. Selic, “An architectural pattern for real-time control software,” in *Workshop on Frameworks and Architectures, PLoP Conference*, 1996.

[30] F. Zalila, X. Crégut, and M. Pantel, “A transformation-driven approach to automate feedback verification results,” in *Model and Data Engineering*. Springer, 2013, pp. 266–277.