Tailoring Models of Concurrency to eXecutable Domain-Specific Modeling Languages

Florent Latombe, Xavier Crégut, Marc Pantel

Tailoring Models of Concurrency to eXecutable Domain-Specific Modeling Languages

Florent Latombe
University of Toulouse, IRIT
Toulouse, France
first.last@irit.fr

Xavier Crégut
University of Toulouse, IRIT
Toulouse, France
first.last@irit.fr

Marc Pantel
University of Toulouse, IRIT
Toulouse, France
first.last@irit.fr

ABSTRACT

Models of Concurrency (MoCs) are formalisms used to capture the concurrent aspects of complex systems. They provide powerful abstractions to model and reason about concurrency, that are mapped to the underlying execution platform's facilities. In this paper, we extend the existing GEMOC *concurrent executable metamodeling approach* enabling the specification of *Concurrency-aware eXecutable Domain-Specific Modeling Languages* (xDSMLs) to ease the definition of new MoCs. This approach relies on dedicated meta-languages to separate the data concerns from the concurrency ones in the operational semantics. The latter are captured as CCSL models which define Event Structures, enabling their use for concurrency-aware analyses, refinements and variations. However, this elementary MoC is not the best fit for all concurrency paradigms; and extending the approach with additional ones is complex and costly. We propose a solution to seamlessly define and integrate new MoCs through a recursive definition of concurrency-aware xDSMLs, enabling the use of a previously-defined xDSML as a MoC. This allows a complete tailoring of the MoC to the xDSML, facilitating the debugging of the MoC use by relying on the execution facilities made available for free by the concurrency-aware approach, and paving the way for additional analyses depending on the xDSML used as MoC. We illustrate our approach on fUML and show our implementation in an Eclipse-based language workbench, the GEMOC Studio.

CCS Concepts

•**Theory of computation** → **Concurrency;** *Operational semantics;* •**Software and its engineering** → **Concurrent programming languages; Domain specific languages;**

Keywords

Models of concurrency; domain-specific modeling languages; executable metamodeling; operational semantics

MODELS 2016 October 2–7, 2016, Saint-Malo, France

DOI: 10.475/123_4

1. INTRODUCTION

Modern complex systems (*e.g.*, Cyber-Physical Systems, Internet of Things, etc.) are highly-concurrent and executed on increasingly-concurrent platforms (*e.g.*, many-core CPUs, GPGPU pipelines, etc). The specification of their concurrency concerns is usually done using a Model of Concurrency (MoC), such as Petri nets [30], the Actor model [1], or Event Structures [44]. Some are primarily intended for specifying and reasoning about systems during their design and simulation, while others are used for implementation and testing as the runtime performance of the former are often too low. Concrete implementations relying on lower-level facilities are thus derived from abstract validated design models and tested using scenarios built from the abstract models. In order to harness this sophisticated development scheme involving various languages with significantly different MoCs, we advocate the use of formally-validated language built using an appropriate language workbench such as GEMOC.

In the programming community, MoCs are typically made available through language constructs (*e.g.*, "Process" for Erlang actors, "java.lang.Thread" for Java/JVM, etc.) or frameworks and libraries (*e.g.*, the "threading" and "multiprocessing" modules in CPython, "akka.actor.Actor" in Scala/JVM, "co.paralleluniverse.fibers" in Quasar/JVM, etc.). They are often poorly identified or specified, each implementation bringing its own flavor of details, making the communication between developpers and/or tools difficult (*e.g.*, between two Actors model implementation, such as Akka's [14] and Erlang's [2]). More importantly, the burden of specifying the use of a MoC by a system befalls the system designer, a difficult task which requires additional knowledge about the MoC, its implementation and good practices.

To ease the use of MoCs, a systematic approach can be defined at the language level. This requires additional effort in terms of language design and tool support, but facilitates, for the end user, the specification of highly concurrent systems. For instance, Rust [29] supports a complex type system and memory model which prevents data races, a typical issue in Concurrent Programming. The Functional Programming paradigm also enables the parallel execution of concurrent programs through the use of immutable data structures and side-effect-free functions.

In this paper, we use such an approach for the definition of *eXecutable Domain-Specific Modeling Languages* (xDSMLs). DSMLs capture domain knowledge in their constructs, easing the specification of the solution for problems in a given domain (*i.e.*, usually a specific aspect of a complex system, such as security, hydraulics, energy consumption, etc.). They

are designed to be easy to use by domain experts, who may not be well-versed in programming or in General-purpose Modeling Languages such as the UML [33]. They have proven effective at solving problems of the domain they were designed for [25]. Making them executable, through the definition of an execution semantics, enables the simulation of the systems being designed (*e.g.*, for verification, validation, testing, etc. purposes). The GEMOC approach enables the definition of so-called *Concurrency-aware xDSMLs* [22]. In these languages, the semantics specification relies on CCSL models [26] to define an Event Structure (*i.e.*, MoC used for any model conforming to the xDSML). This is made possible by the domain-specificity of the language, in contrast with General-purpose Programming Languages (GPLs) which must remain generic and thus cannot be too opinionated about the concurrency concerns of the systems being designed. This approach removes from the system designers the burden of having to identify which MoC to use, how to use its implementation, how to debug it, etc.

However, specifying the language-level concurrency concerns is a complex activity. First, the MoC must be adequate for the systems concerned. So far, the GEMOC approach only handles Event Structures, a MoC inspired from concurrency theory expressed using constraints (*e.g.*, CCSL models [26] based on MoCCML [6] constraints). However, not all MoCs are a good fit for all situations. In [41], the authors show that inadequacies of the Actor model lead Scala code bases to mix other MoCs for some particular system aspects. Another difficulty is that with a MoC also comes the associated meta-language to enable its use at the language level. MoCs often do not include such a meta-language, making the integration of new MoCs in the concurrency-aware approach particularly difficult. Our contribution consists in enabling the definition and use of new MoCs in the GEMOC concurrency-aware xDSML approach. This is realized through a *recursive definition* of the approach, in which the MoC used for a new xDSML can be a previously-defined concurrency-aware xDSML. This allows the use of an adequate MoC for any xDSML, without having to "hard-code" a MoC and its associated meta-language into the approach. It also eases the specification of xDSMLs by making possible the independent execution, debugging and animation of the model-level concurrency specification of a system, since it is only a model conforming to a concurrency-aware xDSML, thus benefiting from all the debugging and animation tools available for free for such languages in the GEMOC platform.

The rest of this paper is organized as follows. First, in Section 2, we explain the initial concurrency-aware xDSMLs approach, illustrated on an example xDSML, fUML. In Section 3, we first show that the Event Structures MoC is not a good fit for all xDSMLs, making the specification of the concurrency concerns complex. 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 4. Section 5 focuses on our implementation in the GEMOC language workbench. Finally, we discuss related work in Section 6; then conclude and give perspectives for future work in Section 7.

2. THE CONCURRENCY-AWARE XDSML APPROACH

This section presents the existing concurrency-aware xDSML approach [4, 22]. Its application is illustrated on the *Foundational Subset for Executable UML Models* (fUML) [34]: an executable subset of UML which specifies the behavioral semantics of Activity Diagrams.

2.1 Language Design Overview

The core element in defining a language is its Abstract Syntax (AS), which specifies its concepts and the relations between them. The AS is usually completed with the Static Semantics, *i.e.*, constraints restricting the set of valid models, otherwise hard or impossible to capture in the AS. In Model-Driven Engineering, the AS is typically captured in a *metamodel*, allowing to define *models*, and the static semantics can be expressed using the OMG's Object Constraint Language (OCL).The AS is usually mapped to a Concrete Syntax (CS), textual (*e.g.*, using Xtext [3]) or graphical (*e.g.*, Sirius [11], MetaEdit+ [19]).

Figure 1 shows an example fUML `Activity`. An `Activity` is composed of nodes (`ActivityNode`) of various natures, connected by edges (`ActivityEdge`). In this example, we drink 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.

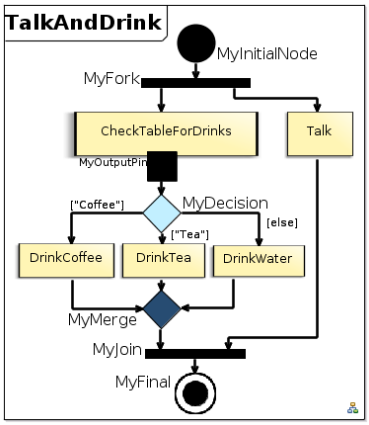

Figure 1: fUML example: Fetching a drink while talking.

We consider that the language's syntax and static semantics are available, and we focus on the definition of its execution semantics. Traditional language design techniques include *axiomatic semantics* [16], *translational semantics* [13] (also called *denotational* when the translation is a mathematical denotation [39]) and *operational semantics* [35]. The concurrency-aware approach is a refinement of the operational semantics approach.

In our example *Activity*, the main point where concurrency matters is in the concurrent branches of the *ForkNode*. This means that 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 of the language [23] or the refinement of the language for a specific execution platform (*e.g.*, sequential, highly-parallel, etc.).

2.2 Separation of Concerns in the Operational Semantics

The concurrency-aware approach relies on a *separation of concerns*, in the operational semantics, of the data and atomic sequential operations, from the concurrency concerns. The former are gathered in the *Semantic Rules*, while the latter are captured in the *Model of Concurrency Mapping*. Both specifications are connected by a *Communication Protocol*.

2.2.1 Semantic Rules

The *Semantic Rules* are composed of two parts. First, the *Execution Data* capture the model runtime state during its execution, *e.g.*, the value of a variable, the current state of an automaton, etc. In fUML, edges carry *Tokens* which may be of two natures (control or data). The second part is the *Execution Functions* which specify how the *Execution Data* evolve at runtime. For instance, a node in fUML can be executed, resulting in changes in the tokens held by its outgoing and incoming edges. Edges with a guard can have their guard evaluated to return a boolean value, etc. When the AS of an xDSML is captured as a metamodel, *Execution Data* and *Functions* can be captured as additional references, attributes and operations weaved onto the metamodel.

2.2.2 Model of Concurrency Mapping

The *Model of Concurrency Mapping* (MoCMapping) specifies how to obtain, for a model conforming to the AS, an instance (called *MoCApplication*) of the MoC used. So far, the approach only includes the *Event Structures* [44] MoC. The MoCMapping is thus called an *EventType Structure* specifying how, for a model, the corresponding MoCApplication (*Event Structure*) representing the pure concurrent control flow of the model is obtained. Figure 2 sums up the relations between these different specifications.

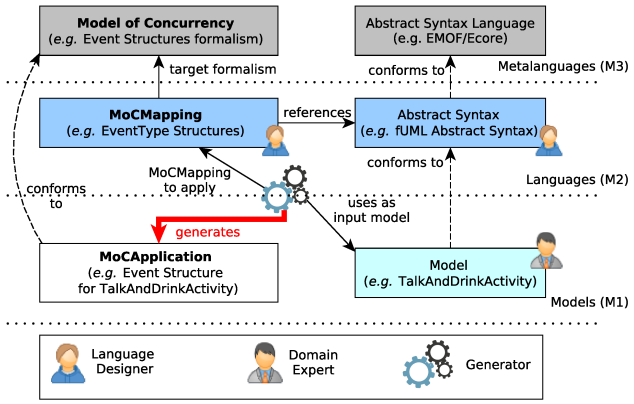

Figure 2: Concepts related to concurrency concerns.

An *Event Structure* is specified through a partial ordering over a set of abstract actions (the events). In other words, it specifies *all the possible execution paths* of a model, independently of its data. Figure 3 shows the metamodel corresponding to *Event Structures* and their execution, while Figure 4 shows the simplified *Event Structure* execution corresponding to our example *Activity*. For representation purposes, we do not show the different orders of execution possible for the guards of edges outgoing a *DecisionNode* (*i.e.*, they can be executed in any order, including in parallel).

In an *Event Structure*, a node is a *Configuration*: an *unordered set of event occurrences* which have happened at this

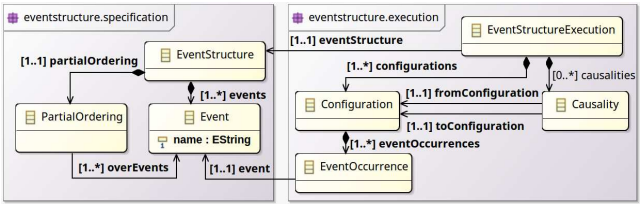

Figure 3: Event Structures and their execution.

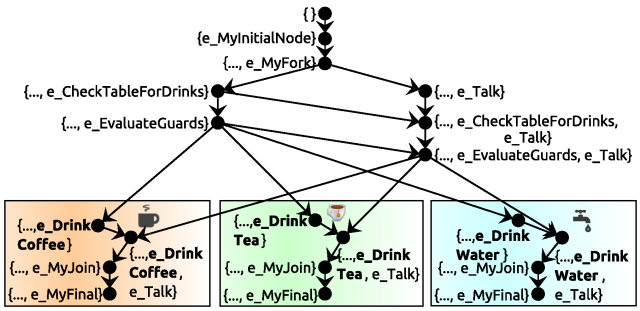

Figure 4: Simplified Event Structure for the example shown on Figure 1.

point. For representation purposes, "..." in a configuration represents the collection of event occurrences from the previous configurations, *e.g.*, {..., e_MyFork} is {e_MyInitialNode, e_MyFork}.

This *Event Structure* captures all the possible execution paths for the model: the two concurrent branches which can be executed in parallel or interleaved, and the three different possibilities resulting of the *DecisionNode*.

2.2.3 Communication Protocol

Finally, the main responsibility of the *Communication Protocol* is matching the abstract actions of the *EventType Structure* with the *Execution Functions* of the *Semantic Rules*. These matches are denoted as *Mappings*. This effectively defines how, at runtime, the *Event Structure* is used to orchestrate the calls to the *Execution Functions*, therefore realizing the execution of a model.

The different specifications at the model-level for a simplified version of the example activity (for representation purposes) are shown on Figure 5. The node "DrinkSomething" abstracts the drinking part of the activity of Figure 1.

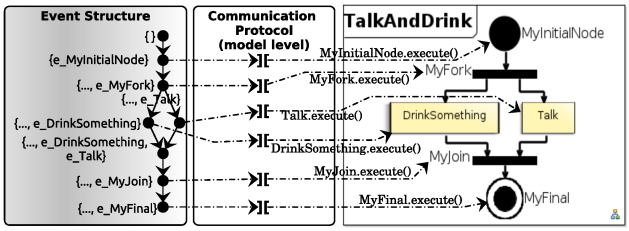

Figure 5: Simplified model-level specifications of the fUML example.

In this figure, the *Event Structure* on the left captures all the possible execution paths of the model. After initializing the activity, the *ForkNode* is executed. After that, in this

simplified view, there are three solutions: drinking something then talking, talking and then drinking something, or talking while drinking something. Ultimately the same configuration is attained. After that, the *JoinNode* and *FinalNode* can be executed. For this simplified activity, this gives us 3 possible scenarios in total. But if we consider the details of the drinking part of Figure 1, we have a total of 64 different possible scenarios, accounting for all the possible interleavings and parallelisms between the talking and drinking part of the activity, as well as the different possible orders of evaluation of the guards outside the *DecisionNode*.

Figure 6 show an overview of the approach we just described, as a *Class Diagram*. On this figure, *EventTypes* are abstracted away as *MoCTriggers* since our approach can be manually adapted to integrate other MoCs.

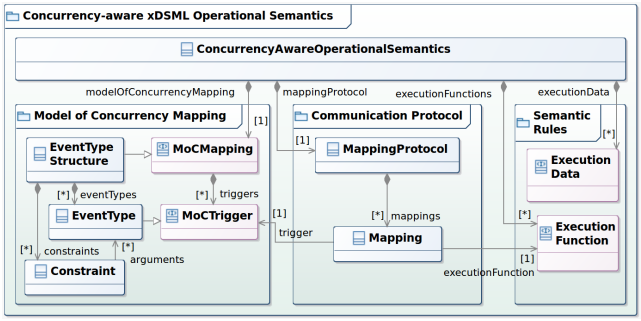

Figure 6: Class diagram of the xDSML approach.

2.3 Runtime of the Approach

Using a model conforming to the AS of the language, the corresponding model-level specifications for each concern are generated. The execution of a model is then as follows. First, the runtime of the *MoCApplication* computes the set of possible execution paths, called *Scheduling Solutions*, based on the partial ordering defined. An arbitrary one is selected, based on an heuristic of the runtime. In particular, it can consist in asking the user to choose (*e.g.*, if both branches should be executed in parallel, or in sequence). Next, based on the *Communication Protocol*, the selected solution is matched against the *Mappings* to find out the set of *Execution Function* calls to realize. These calls are then executed, effectively resulting in changes of the runtime state of the model, realizing one step of execution of the model.

2.4 Upsides

This approach enables the use of concurrency-aware analyses on the *MoCApplication* (*e.g.*, deadlock freeness, starvation, etc.) to guarantee systems safety and behavioral properties [27]. xDSMLs can also be refined to specific execution platforms, for instance to maximize parallel concurrent operation execution instead of interleaving. Reversely, parallel operations can be forbidden in order to cater to a sequential execution platform. It also favors the specification and implementation of *Semantic Variation Points* (SVPs), *i.e.*, parts of the semantics left under-specified to allow further adaptation for specific uses, as illustrated on StateCharts in [23]. Finally, the separation of concerns promotes the reusability and variation of parts of the semantics of a language, since the *Semantic Rules* and *MoCMapping* can be redefined independently.

3. USING A TAILORED MODEL OF CONCURRENCY FOR EVERY XDSML

In this section, we first motivate the use of other MoCs besides Event Structures. We consider their integration into the approach and describe a generic way to define and seamlessly integrate new MoCs thanks to a recursive definition of concurrency-aware xDSMLs. Our contribution is illustrated throughout on the definition of fUML with a new MoC based on the notion of *Thread*.

3.1 Selection and Integration of New MoCs

The selection of the MoC to use for an xDSML is based on several criterias.

The most important one is the adequacy of the MoC to the concurrency paradigm of the systems designed with the xDSML. In "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" [41], 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 [41] 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 criterias can make the use of a MoC complex, which manifests, in our approach by making the specification of the *MoCMapping* more complicated.

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 how remains at the discretion of the MoC used. Some communities have developed tools and methods for such verifications, *e.g.*, Petri nets [30] 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).

Finally, the selection of a MoC is pragmatically based on what is made available by the development environment. The GEMOC concurrency-aware approach, so far, only enables the use of the Event Structures MoC, relying on various constraint-based languages like CCSL and MoCCML to define EventType Structures. Integrating new MoCs, so that different MoCs can be selected for different xDSMLs, based on the criterias we have described above, requires a significant effort. The meta-language corresponding to the MoC must be integrated; as well as the meta-language for the definition of the *MoCMapping* ; accompanied by the generator to unfold the *MoCMapping* for a model conforming to the AS of the xDSML ; and finally the runtime of the MoC in order to be able to execute the resulting *MoCApplication*. All these components must be integrated with the other concerns of the xDSML definition and their respective runtimes. One of the main difficulties resides in designing the meta-language for the *MoCMapping*, as well as its associated tooling. This language-level notion has been contributed by the concurrency-aware approach, motivated by the needs in Domain-Specific Languages design techniques, and is thus not commonly available for MoCs.

In the rest of this section, we will describe 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.

3.2 A Thread-based MoC for fUML

To illustrate our contribution, we propose to reconsider the use of the Event Structures MoC for fUML. There is a conceptual gap between how the semantics is described in the fUML specification (in English, and in plain Java in the reference implementation[1]), and how we are able to capture it using the Event Structures MoC. For instance, specifying the *MoCMapping* pertaining to *DecisionNodes* is complex because of the dependency towards the evaluation of the guards of the outgoing edges. It was precisely the focus of [22] in which we explained in greater details the intricacies of this type of language constructs. Moreover, Event Structures capture explicitly all possible interleavings for a model (*e.g.*, between the drinking and talking branches in our example), leading to an exponential number of configurations and dependencies between configurations for big or highly-concurrent models. This can affect the validation of the semantics, the runtime performance, the performing of analysis and even the representation of the concurrency concerns.

We propose to consider a definition of fUML where the MoC used is based on the notion of *Thread*. By "Thread", we designate the conceptual unit for computation, also known as green threads, lightweight threads or user threads, as opposed to operating system's threads (kernel threads). The mapping between the two depends on the implementation of the runtime of the concurrency-aware xDSML approach. This MoC is convenient for fUML because the semantics of fUML is given in English and in plain Java for the reference implementation, and it corresponds to the MoC provided in Java by the threading API.

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 a main thread, the one in charge of the *InitialNode* and the *FinalNode* of the activity. The execution of nodes placed in sequence in an activity can be represented by a sequence of instructions in a thread. For every *ForkNode*, one thread is created by branch. It is in charge of executing the associated branch's content. Executing the corresponding *JoinNode* is possible when all these threads have finished executing their instructions. Threads could also be used for the evaluation of the guards of a *DecisionNode*, since they are to be evaluated in no specific order.

Figure 7 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 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, 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.

This formalism is not only more adequate for the specification of the fUML concerns due to its closeness with the

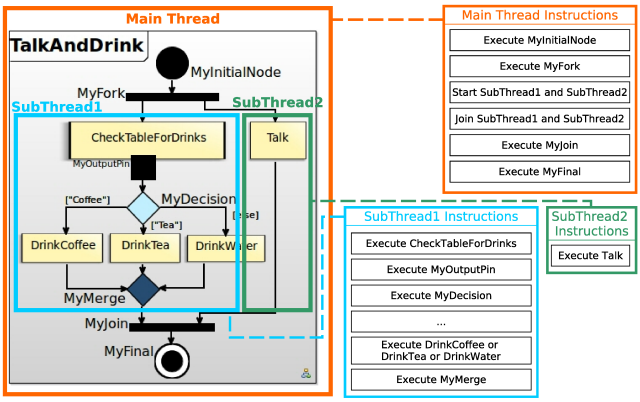

Figure 7: Mapping the fUML example to threads.

specification and reference implementation, it is also more practical to represent graphically, as all the interleavings are not represented explicitly. Instead, they are implicit in the use of concurrent threads for concurrent instructions.

3.3 Introducing a Recursive Definition of Concurrency-aware xDSMLs

More generally, we propose to leverage a previously-defined concurrency-aware xDSML as the MoC for another xDSML. Since such an xDSML could also be used as a MoC for yet another xDSML, this effectively enables a recursive definition of Concurrency-aware xDSMLs, with Event Structures as the base MoC used for the underlying execution. Recursive definitions are not unheard of when defining MoCs. For instance, Java's Futures, Akka's actors [14], etc. are built on top of Java's Threads. The link towards system-level threads is then defined by the JVM implementation used (*e.g.*, most commonly 1:1, as done by Oracle's HotSpot[2]).

The main idea of our approach is as follows: a MoC is a formalism used to specify the concurrency of a system. xDSMLs are language and can thus be used as MoCs when they are adequate. 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 2 is now a kind of model transformation from the xDSML to the MoC. The model resulting from the transformation (the *MoCApplication*) is however *not semantically equivalent* to the original model: it only represents its pure concurrent control flow. The *MoCTriggers* of this *MoCMapping* are the *Mappings* (from the *Communication Protocol*) of the MoC. And since an element in a model may result in several elements in the resulting model (*e.g.*, *ForkNodes* are transformed into several instructions), we also need to be able to exploit the trace of the transformation in order to target the *Mappings* corresponding to the right elements of the *MoCApplication*.

Another way to see our approach is to consider the design of the *EventType Structure* for an xDSML. There are two stages: first, declaring *EventTypes* in the context of concepts from the AS; then defining a symbolic partial ordering over these *EventTypes* (the partial ordering is at the model level). Our

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

[2]http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#Thread%20Management|outline

approach effectively allows us to reuse the partial ordering of an xDSML (implicitly, thanks to the mapping between *EventTypes* and *Mappings*), which is the most difficult part of defining the *EventType Structure* for a language.

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, applied 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 2, or as is being shown in this section. Figure 8 gives an overview of our approach as a class diagram. It relies on two additional specifications presented in the "Concurrency-aware xDSML Recursive Definition" package of the class diagram. These specifications and the runtime are detailed and illustrated in the rest of this section.

3.3.1 Abstract Syntax Transformation

The first one is a **Transformation** from $\mathcal{L}_{Domain}$ to $\mathcal{L}_{MoC}$, denoted as $\mathcal{T}_{Domain \to MoC}$. It specifies how the pure concurrent control flow of $\mathcal{L}_{Domain}$ is 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 some 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. Once we have designed this language using the concurrency-aware approach, we can specify $\mathcal{T}_{fUML \to Threading}$. For our example model of Figure 1, the resulting model corresponds to the right half of Figure 7.

3.3.2 Exploiting the Trace of the AS Transformation

The *Communication Protocol* implements the mapping between the *MoCTriggers* (originally implemented by *EventTypes*) and the *Execution Functions* of $\mathcal{L}_{Domain}$. In this approach, *MoCTriggers* are implemented by the *Mappings* of $\mathcal{L}_{MoC}$. In the case of fUML, the main *Mapping* of interest in the *Communication Protocol* of the *Threading* language pertains to the execution of a *Task*, denoted as `ExecuteTask`.

However, the correspondence between $\mathcal{L}_{Domain}$ and $\mathcal{L}_{MoC}$ is always $1 \to n$ (with $n \geq 0$). When $n = 0$, it means that the element of $\mathcal{M}_{Domain}$ has no direct impact on the control flow (*e.g.*, edges in fUML). We have $n = 1$ when the element of $\mathcal{M}_{Domain}$ is transformed into one element in $\mathcal{M}_{MoC}$, such as fUML nodes generally being transformed into one *Task*. Finally, $n > 1$ when the element of $\mathcal{M}_{Domain}$ is represented using multiple elements in $\mathcal{M}_{MoC}$, such as a *ForkNode* being transformed into several *Tasks* (corresponding to the creating of as many *Threads* as there are branches). In other words, $\mathcal{T}_{Domain \to MoC}$ does not add new information, it merely encodes the control flow associated with the constructs of $\mathcal{L}_{Domain}$, using $\mathcal{L}_{MoC}$. The rest of the specification of $\mathcal{L}_{Domain}$ (*Semantic Rules*) handles the data concerns of the language.

In particular, several concepts of $\mathcal{L}_{Domain}$ can be mapped to a same concept in $\mathcal{L}_{MoC}$, for different purposes. For instance, executing a node is encoded as a *Task* in the *Threading* language, but not all *Tasks* correspond to the execution of a node. In order to be able to identify which mappings of $\mathcal{L}_{MoC}$ correspond to the *MoCTriggers* of $\mathcal{L}_{Domain}$, we thus need an additional specification, which we call the **Projections** of $\mathcal{L}_{Domain}$, designated as $\mathcal{P}_{Domain \to MoC}$. It specifies, for a concept of $\mathcal{L}_{Domain}$, into which concept(s) of $\mathcal{L}_{MoC}$ they are transformed (through $\mathcal{T}_{Domain \to MoC}$) and with which purpose(s), through labels. This allows us to identify, for instance, the *Tasks* corresponding to the creating of the threads resulting from the transformation of a *ForkNode*. For an fUML edge with a guard, there are three *Tasks* corresponding respectively to the evaluation of its guard and to the consequence of the result of its guard (*i.e.*, either the branch is allowed or it is not). This specification is required for the correct generation of the model-level *Communication Protocol* of $\mathcal{M}_{Domain}$. More precisely, $\mathcal{P}_{Domain \to MoC}$ is exploited by the *Communication Protocol* specification. Its model-level counterpart is generated when applying $\mathcal{T}_{Domain \to MoC}$ (*i.e.*, it is part of the trace of that transformation) and exploited when generating the model-level *Communication Protocol*. Otherwise, there would be ambiguities as to which *Mappings* should be used as *MoCTriggers* for the *Mappings* of fUML.

This specification, denoted as $\mathcal{P}_{fUML \to Threading}$ for fUML, as well as its use by the *Communication Protocol* specification of fUML, are both illustrated in Section 5 using our meta-language implementations.

3.3.3 Runtime

Finally, the runtime of the approach must be modified accordingly. The runtime for $\mathcal{M}_{Domain}$ manipulates the runtime for $\mathcal{M}_{MoC}$. It retrieves the occurring $\mathcal{M}_{MoC}$ *Mappings* (thanks to an API of the runtime) and matches them against the *Mappings* used in the *Communication Protocol* of $\mathcal{M}_{Domain}$, thus identifying all the possible solutions for a step of execution. When one of these possible solutions is selected, the runtime of $\mathcal{M}_{MoC}$ is notified so that it performs the selected step of execution. Meanwhile, the corresponding *Execution Function* calls for $\mathcal{M}_{Domain}$ are also executed, thus realizing one step of execution. The main change is that the runtime of xDSMLs must conform to the same interface used previously for the runtime of the *MoCApplication*.

For our example fUML Activity, this means that at runtime, the threading model is (also) executed, but under a finely-grained control operated by the runtime for the fUML Activity. During the execution, when a *Threading* mapping such as "ExecuteTask_MyInitialNode" occurs (as part of a possible *Scheduling Solution*), it is used by the runtime of the fUML model and matched against the *Communication Protocol* for the fUML Activity, thus enabling the fUML mapping "ExecuteActivityNode_MyInitialNode", which, if selected, will trigger the call to "MyInitialNode.execute()". This corresponds to the use of a Threading model to define the pure concurrent control flow of the execution of the example fUML Activity.

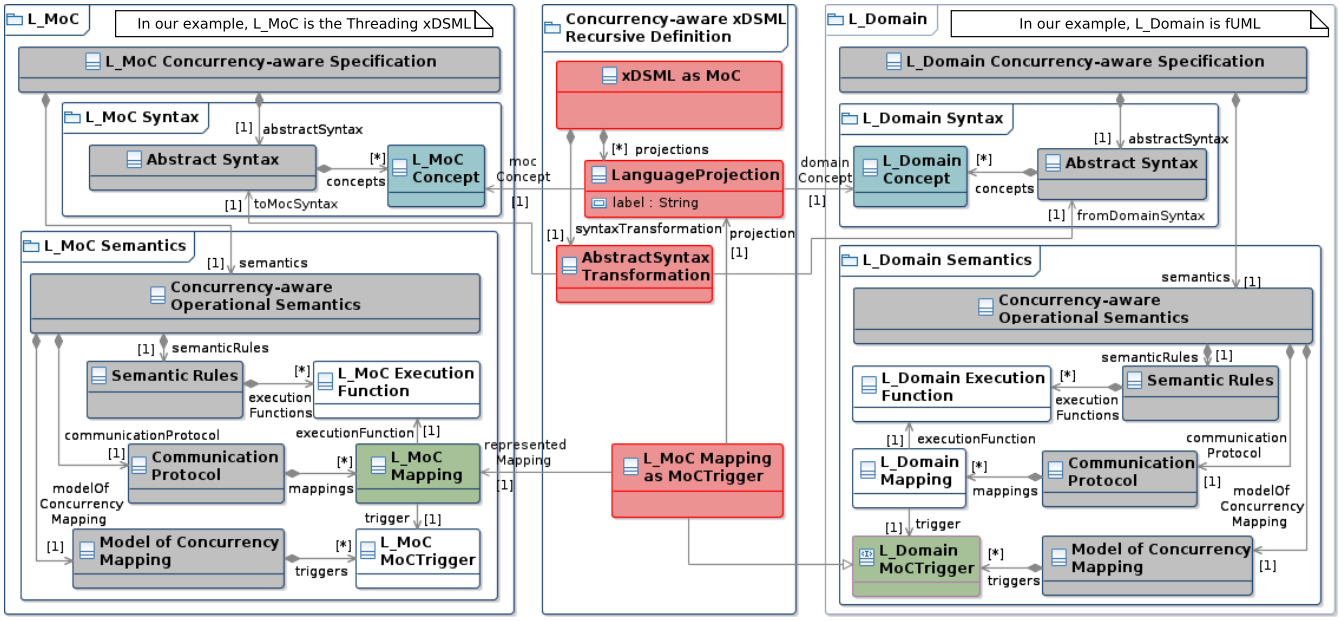

Figure 8: Overview of the recursive definition of concurrency-aware xDSMLs.

4. DISCUSSION

We discuss the advantages and limitations of our approach.

4.1 Modularity

The initial concurrency-aware xDSML approach focused on the separation of concerns in language execution semantics to make explicit the concurrency concern and allow its use for analyses, reuse and variations. Our contribution does not disrupt this modularity, as we have only provided the means to use other MoCs defined as concurrency-aware xDSMLs. The *MoCMapping* remains a data-independent specification making explicit the language concurrency concerns. In fact, our approach favors the reuse of the AS and *Semantic Rules* for languages with different MoCs, for instance to compare two MoCs for a same language in order to determine which is more appropriate. Reversely, concurrency-aware xDSMLs can be used as MoC of any other language.

4.2 Concurrency-aware Analyses

Analyses of a system's concurrency aspects is performed based on its *MoCApplication*, depending on the MoC used. For instance, Petri nets [30] are a common formalism to specify the behavior of concurrent systems and verify liveness or safety properties. Other xDSMLs however, may not offer such tooling or well-known properties. The use of *any* concurrency-aware xDSML as MoC gives the language designer the ability to choose the most appropriate MoC depending on its desired properties or tools. Still, since our approach remains rooted in Event Structures, there is ultimately an underlying Event Structure representing a system concurrency concern. In our example, the *MoCApplication* of an fUML activity is a *ThreadSystem* model, whose *MoCApplication* is an Event Structure. By transitivity, we can analyze the concurrency concerns of the fUML activity through this Event Structure [27]. Overall, our approach does not hinder the use of any concurrency-aware analyses that were previously possible, and actually provides an additional hook for analyses by enabling the use of an xDSML as MoC. This MoC may rely on existing tooling or well-known specific properties to help verify additional aspects of the system concurrency.

4.3 Adequate MoCs for xDSMLs

By enabling the use of any concurrency-aware xDSML as a MoC, 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 propose: some formalisms are more adapted for the specification of some concurrency paradigms.

The use of DSLs relies on: 1) being able to identify the DSL to design and 2) having the appropriate tools to specify, implement and use the DSL. This is also the case for the use of an xDSML as MoC: it relies on identifying the most fitted formalism, and on having it specified as a concurrency-aware xDSML. This may require additional work from the language designer, who should also learn the language used as MoC, whereas previously (s)he only needed to master the Event Structures MoC. But in the same way DSLs are worth their costs, so is our contribution. In [41], mixing MoCs in a Scala code base or using an ill-fitted MoC ultimately resulted in complex programs with deadlocks and data races, preventing the use of advanced tools, etc.

By using xDSMLs as MoCs, a practical and adequate formalism can be used for a specific xDSML's concurrency paradigm; and its use is facilitated by the execution, simulation and debugging facilities provided for free for concurrency-aware xDSMLs, which can then be used on the *MoCApplication*, which becomes a model conforming to another concurrency-aware xDSML.

4.4 Unified Structure for MoCs

Another upside of our approach is that it provides a unified

structure to all MoCs. Usually, MoCs are specified informally, sometimes presented as "formalisms" (*e.g.*, Petri nets [30]), available through language constructs (*e.g.*, Erlang actors [2]) or through a framework (*e.g.*, actors in Scala's Akka [14]) or library (*e.g.*, fibers in Ruby [38], threading in Python, etc.). Although some work has been done towards the unification of MoCs [24, 32], they mostly studied a set of MoCs, without considering the possibility to define or use new formalisms as MoCs. With our contribution, any concurrency-aware xDSML can be used as a MoC. It is the use that is made of an xDSML that determines whether it corresponds to a MoC or not. In other words, "MoC" is a *role* played by a language, not its nature. Such MoCs can also be used at the application level, by defining a model conforming to its syntax. But an important point of our recursive approach is that the meta-language for the *MoCMapping* is obtained for free, whereas MoCs typically do not include how they can be used at a language level.

4.5 Comparison with translational semantics

The translational semantics approach consists in defining the execution semantics of a language by translating it into another well-defined language. This is usually done through the specification of a transformation from the source language to a target language. In some particular cases, this technique is also called "Compilation" (*i.e.*, the target language is a lower-level language like machine code). Our contribution bears resemblance with translational semantics in that we do define a transformation from $\mathcal{L}_{Domain}$ to $\mathcal{L}_{MoC}$: $\mathcal{T}_{Domain \to MoC}$. 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 $\mathcal{L}_{MoC}$.

5. IMPLEMENTATION

Our approach has been implemented in a language workbench, the GEMOC Studio[3]. It is based on the Eclipse Modeling Framework (EMF) [10] in order to benefit from its large ecosystem and associated tools.

5.1 Existing Elements

An xDSML *Abstract Syntax* is specified using Ecore, the EMF implementation of EMOF. Its static semantics can be specified using the Object Constraint Language (OCL). EMOF and OCL are both standards from the Object Management Group (OMG)[4].

Semantic Rules are specified using the Kermeta 3 Action Language (K3AL) [9], based on xTend [3]. In K3AL, Ecore metaclasses can be extended with additional attributes, references and operation implementations through aspects. This allows to specify the *Execution Data* and *Execution Functions*. Just like xTend, K3AL compiles into readable Java code, and provides an executor based on the Java Reflection API in order to dynamically invoke the *Execution Functions*.

EventType Structures can be specified using a combination of MoCCML [6], a declarative meta-language designed to create constraints, and of Event Constraint Language (ECL) [7], an extension of OCL allowing the definition of constrained *EventTypes* in the context of concepts from the AS. For a model, the Event Structure is captured as a Clock Constraint Specific Language (CCSL) [26] model, which can be interpreted by the TimeSquare tool [8].

The *Communication Protocol* is specified using a dedicated meta-language called the Gemoc Events Language (GEL) [22] as the *Mappings* between *MoCTriggers* (*e.g.*, ECL *EventTypes*) and *Execution Functions* (*e.g.*, defined in K3AL). It also allows the specification of the *Feedback Protocol* which is required when the semantics of a language construct depends on data available at runtime in the model. This issue and its solution are detailled in [22]. In short, in the case of fUML *DecisionNodes*, it specifies that if the evaluation of a guard returns true, then the corresponding branch may be executed, and that otherwise it may not.

The execution of a model is done by a runtime called the *Execution Engine*. It coordinates the runtime of each meta-language used to specify the concurrency-aware semantics of the xDSML. In our context, the runtime for the *MoC Application* is called the *Solver*, while the runtime for the *Semantic Rules* is called the *Executor*. The runtime for the model-level *Communication Protocol*, which, given a set of occurring *MoCTriggers*, matches which *Execution Function* calls are possible, is called the *Matcher*. Figure 9 shows a simplified sequence diagram of how to realize a step of execution, corresponding to the description we gave in Subsection 2.3. This architecture has a significant execution cost but it eases the modeling and validation of sophisticated xDSMLs. The produced models can then be used as requirements for the implementation of more efficient execution platforms. GEMOC also provides the Behavioral Coordination Language (BCOol) [21] that allows modeling the coordination between various xDSMLs. In the same manner, these models are requirements for more efficient implementations of coordination models.

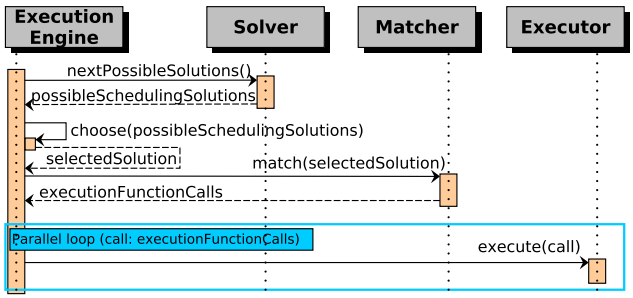

Figure 9: Sequence diagram of a step of execution.

5.2 Projections, Transformation and Communication Protocol

We have designed a dedicated meta-language for the specification of $\mathcal{P}_{Domain \to MoC}$. Listing 1 shows the projection for fUML, $\mathcal{P}_{fUML \to Threading}$. fUML *ActivityNodes* are transformed into *Tasks*. When a *Task* is executable, then the

[3]http://www.gemoc.org/studio
[4]http://www.omg.org

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.

Listing 1: fUML projections on Threading.

```
Projections:
 LanguageProjection Projection_Execution:
  fuml.ActivityNode projected onto threaded.Task
 end
 LanguageProjection Projection_Evaluation:
  fuml.ActivityEdge projected onto threaded.Task
 end
 LanguageProjection Projection_MayExecute:
  fuml.ActivityEdge projected onto threaded.Task
 end
 LanguageProjection Projection_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. The OMG has standardized M2M transformations in Queries, Views, Transformations (QVT). (see for exemple the Atlas Transformation Language (ATL) [17]). But M2M transformations can also be specified using general-purpose programming languages such as Java with EMF's APIs. However, the transformation must not only transform a $\mathcal{M}_{Domain}$ into a corresponding $\mathcal{M}_{MoC}$, but it must also generate the model-level specification of $\mathcal{P}_{Domain \to MoC}$, denoted as $\mathcal{P}_{DomainModel \to MoCModel}$. This specification is the trace that relates elements of $\mathcal{M}_{Domain}$ with elements of $\mathcal{M}_{MoC}$. It is used for the generation of the model-level *Communication Protocol*.

Finally, our *Communication Protocol* meta-language, GEL, has been extended. *MoCTriggers* can now consist of a *Mapping* (from $\mathcal{L}_{MoC}$) and of a reference to one of the projections from $\mathcal{P}_{Domain \to MoC}$. Listing 2 shows the *Communication Protocol* for our implementation of fUML.

Listing 2: Communication Protocol for fUML.

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

DSE EvaluateGuard:
 upon event ExecuteTask
       with Projection_Evaluation
 triggers ActivityEdge.evaluateGuard
          returning result
 feedback: // Presented in more details in [22].
  [result] => allow event ExecuteTask
                 with Projection_MayExecute
  default => allow event ExecuteTask
                with Projection_MayNotExecute
 end
end
```

Mappings are implemented by what we call *Domain-Specific Events* (DSE). The DSE *ExecuteActivityNode* occurs whenever the corresponding *ExecuteTask* DSE occurs. At the model level, this means that *ExecuteActivityNode_MyInitial Node* occurs whenever *ExecuteTask_Execute_MyInitialNode* occurs, because *Execute_MyInitialNode* is the Task corresponding to *MyInitialNode* through the projection *Projection_Execution*.

5.3 Changes to the Runtime

When $\mathcal{L}_{Domain}$ uses $\mathcal{L}_{MoC}$ as its MoC, the *MoCApplications* are models conforming to $\mathcal{L}_{MoC}$. Therefore, the *Solver* for $\mathcal{L}_{Domain}$ is the *Execution Engine* of $\mathcal{L}_{MoC}$. In our example, the *Solver* used by the *Execution Engine* of fUML is the *Execution Engine* of our *Threading* language used as MoC. In our implementation, the *Execution Engine* also implements the *Solver* API.

5.4 Execution of the Example Activity

The full execution of the example fUML Activity is available as a video at http://gemoc.org/models16/. The video shows the different parts in the concurrency-aware specification of fUML using *Threading* as a MoC. We then show the graphically animated execution of the fUML example *Activity* in our language workbench. The sources are also made available as an archive file.

6. RELATED WORK

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

First, programming language theory, which is used for defining (executable) DSMLs. Language editors, also called Language Workbenches [12, 20], such as JetBrain's MPS [43], Metacase's MetaEdit+ [19] or Microsoft's DSL Tools [5], traditionally focus on the syntactic issues of languages (*i.e.*, the definition of the abstract and concrete syntaxes). Executability of DSMLs has been studied through "meta-programming" approaches such as Rascal [42] or Spoofax [18] (and the older Centaur and ASF+SDF toolsets) ; or "executable metamodeling" approaches, such as xMOF [28] or the K Framework [37]. In both cases, the concurrency concerns are either embedded in the meta-languages provided by the approach used, or are implicitly provided by the underlying execution platform (*e.g.*, if the semantics of a DSML is given by translation to a GPL like Java). In the approach we use, the concurrency concerns are made explicit in a dedicated specification *at the language level*. However, it raises the issue of which formalism to use for the concurrency concerns, and how to integrate it at the specficiation and runtime levels. This paper proposed a generic solution to this issue by considering concurrency-aware xDSMLs as MoCs.

Second, concurrency theory, which has studied Models of Concurrency for a long time [31] both on the formal and tooling sides, among which Petri nets [30], the Actor Model [1] or Event Structures [44]. These formalisms have proven to be useful for the specification of systems concurrency concerns as they specify how the different threads of execution collaborate to complete the task they are given: different MoCs typically split a task in different manners and the communication and collaboration between threads is done in different ways (*i.e.*, in terms of data-sharing, scheduling, cooperation, etc.). Unifications of MoCs have been proposed through the use of the "tagged signal" model [24] or of category theory [32], but in our approach we focus on the design and use of new MoCs, and unify their definition by specifying them as languages, enabling the use of traditional language tooling: editors, debuggers, animators, etc. Moreover, these approaches do not bring the MoCs to the language level, so understanding and using a MoC remains challenging for the system designers. For instance, the "tagged signal" presented in [24] has lead to the development of Ptolemy[5] [36],

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

a framework supporting the actor-oriented design of systems. It relies on the notion of director, corresponding to what we have called MoC. But Ptolemy focuses on the design of systems, using a finite set of available directors, and adding new ones requires modifying its source code. The ModHel'X framework [15] improved on it by relying on a component infrastructure to combine MoCs. In our approach, the "director" used is specified at the language level, guaranteeing its use at the system level, and our contribution allows the specification of new ones directly integrated into our framework, thanks to our recursive approach.

Initially, the concurrency-aware xDSML approach was only based on Event Structures, and as such meta-languages used to specify the concurrency concerns at the language level were also based on Event Structures. In this paper, we have provided the means to exploit any previously-defined concurrency-aware xDSML as MoC. In particular, this eases the identification of MoCs, since they now have a more formal structure. The poor identification of MoCs, and the unability to select a specific MoC for the description of a particular system are among the issues plaguing the use of MoCs in the programming community. For instance, in "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" [41], the authors analyze several Scala code-bases and interview their developers in order to determine how often and why the Actor model of concurrency was not the only one used. Reasons were categorized into three groups: inadequacies of the actor library (i), inadequacies of the actor model (ii), and inadequate developer experience (iii). By using the concurrency-aware xDSML approach, (i) and (iii) are moved away from the system designer to the language designer, thereby greatly reducing the number of users concerned with how a MoC should be used correctly. Morever, thanks to the contribution described in this paper, (ii) is solved since any MoC can be specified, used or re-used for different xDSMLs, as long as they have been specified as concurrency-aware xDSMLs using our approach. This enables the language designer to use an adequate formalism for the concurrency concerns of an xDSML. MoCs can then be combined thanks to operators defined using BCOol [21].

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 [40]. 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.

7. CONCLUSION AND PERSPECTIVES

We have extended a concurrent executable metamodeling approach supporting the specification of concurrency-aware xDSMLs. These xDSMLs integrate an explicit and systematic use of Models of Concurrency. However, this approach initially only supports the Event Structures MoC. We advocated that not all MoCs are a good fit for all concurrency paradigms, and that manually integrating new MoCs into the approach is difficult as it requires two meta-languages (for the language and model levels) as well as their toolings.

To ease this activity, we have proposed in this paper to extend the approach with the means to consider previously-defined concurrency-aware xDSMLs as MoCs. This effectively enables a recursive definition of concurrency-aware xDSMLs, using Event Structures as the base MoC. This contribution relies on two additional parts in the language specification: a *syntax transformation*, encoding the concurrency concerns of an xDSML using another xDSML (the MoC); and parts of this transformation trace, which we have called the *Projections*, in order to account for cases where a concept of the xDSML is transformed into several concepts of the MoC. The former can be specified using common model transformations, while for the latter, we have devised a dedicated meta-language. The approach has been implemented in an Eclipse-based language workbench, the GEMOC Studio, including the modification of the existing meta-languages to support the recursive definition of xDSMLs we have described. We have illustrated our contribution by specifying fUML using an xDSML as MoC. This xDSML captures the notions of threads with instructions, and is conceptually closer to the fUML specification given in English and plain Java, making it more adequate. We have made available a video showing the execution of an example fUML Activity.

Models of Concurrency are typically described as formalisms, or languages, but rarely implemented as such. By defining MoCs as concurrency-aware xDSMLs, we have given them a systematic structure, enabling their use at the language-level for the specification of other concurrency-aware xDSMLs. In particular, it enables the use of a MoC that is a good 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 validated on executable models depending on their MoC.

Although all concurrency-aware xDSMLs can technically be used as MoCs, the concurrency theory community has already studied in details a number of well-known MoCs such as the Actor Model or Petri nets. In future work, we plan to provide reference implementations of these best-known MoCs, to provide a standard library of MoCs, upon which existing languages and models could be reproduced using our approach. Afterwards, existing tooling around these formalisms, such as model-checking tools, could be integrated seamlessly in our approach. Still, concurrency-aware xDSMLs remain rooted in the Event Structures MoC, so we first plan to include analysis tools based on the Event Structures formalism. This requires the possibility to translate domain properties into Event Structure properties (possibly through several layers, if the MoC's MoC is another xDSML itself), as well as translating their results back to the domain. Higher-order Transformations could be used for such back-and-forth translations [45]. These analyses could however be performed on any model conforming to any concurrency-aware xDSML. Finally, future work should include precise analysis of the performance and costs associated with our approach, as relying on such a recursive definition of xDSMLs can be expensive for very large languages or models. This is not a strong issue for the use of language models as references or requirements for more efficient tools. However, we plan to study how code generation or scheduler synthesis from language models could help in the development of these tools.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

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

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

[4] B. Combemale, J. Deantoni, M. Vara Larsen, F. , O. Barais, B. Baudry, and R. France. Reifying Concurrency for Executable Metamodeling. In *SLE'13*.

[5] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-specific development with visual studio dsl tools*. Pearson Education, 2007.

[6] 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.

[7] J. Deantoni and F. Mallet. ECL: the event constraint language, an extension of OCL with events. Technical report, Inria, 2012.

[8] J. Deantoni and F. Mallet. Timesquare: Treat your models with logical time. In *Objects, Models, Components, Patterns*, pages 34–41. Springer, 2012.

[9] DIVERSE-team. Github for k3al, 2016.

[10] Eclipse Foundation. EMF homepage, 2016.

[11] Eclipse Foundation. Sirius homepage, 2016.

[12] M. Fowler. Language workbenches: The killer-app for domain specific languages, 2005.

[13] L.-a. Fredlund, B. Jonsson, and J. Parrow. An implementation of a translational semantics for an imperative language. In *CONCUR*. Springer, 1990.

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

[15] C. Hardebolle and F. Boulanger. ModHelâĂŹX: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering*, pages 247–258. Springer, 2008.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[17] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006.

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

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

[20] Language Workbenches Challenge. Comparing tools of the trade, 2014.

[21] M. E. V. Larsen, J. DeAntoni, B. Combemale, and F. Mallet. A behavioral coordination operator language (bcool). In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada*, 2015.

[22] 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.

[23] F. Latombe, X. Crégut, J. Deantoni, M. Pantel, and B. Combemale. Coping with Semantic Variation Points in Domain-Specific Modeling Languages. In *1st International Workshop on Executable Modeling (EXE 2015)*, Ottawa, Canada, 2015. CEUR.

[24] 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.

[25] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *4 th Workshop on Domain-Specific Modeling*, 2004.

[26] F. Mallet. Clock constraint specification language: specifying clock constraints with UML/MARTE. *Innovations in Systems and Software Engineering*, 2008.

[27] F. Mallet and R. De Simone. Correctness Issues on MARTE/CCSL constraints. *Science of Computer Programming*, 106:78–92, Aug. 2015.

[28] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. xMOF: Executable DSMLs based on fUML. In *SLE*. 2013.

[29] Mozilla Research. The Rust Programming Language.

[30] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.

[31] M. Nielsen. Models for concurrency. In *Mathematical Foundations of Computer Science*. 1991.

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

[33] OMG. UML superstructure specification v2.4.1, 2011.

[34] OMG. fUML specification v1.1, 2013.

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

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

[37] G. Rosu and T. F. Serbanuta. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, 2014.

[38] Ruby Community. Ruby documentation about Fibers.

[39] D. S. Scott and C. Strachey. *Toward a mathematical semantics for computer languages*. 1971.

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

[41] 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.

[42] T. Van Der Storm. The Rascal Language Workbench, 2011.

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

[44] G. Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, LNCS. Springer, 1987.

[45] F. Zalila, X. Crégut, and M. Pantel. A transformation-driven approach to automate feedback verification results. In *Model and Data Engineering*, pages 266–277. Springer, 2013.