Weaving Concurrency in eXecutable Domain-Specific Modeling Languages

Florent Latombe, Xavier Crégut, Benoit Combemale, Julien Deantoni, Marc Pantel

Weaving Concurrency in 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

Benoît Combemale
University of Rennes I, IRISA, Inria, Rennes, France
first.last@irisa.fr

Julien Deantoni
Univ. Nice Sophia Antipolis, CNRS, I3S, Inria, Sophia Antipolis, France
first.last@polytech.unice.fr

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

Abstract

The emergence of modern concurrent systems (e.g., Cyber-Physical Systems or the Internet of Things) and highly-parallel platforms (e.g., many-core, GPGPU pipelines, and distributed platforms) calls for Domain-Specific Modeling Languages (DSMLs) where concurrency is of paramount importance. Such DSMLs are intended to propose constructs with rich concurrency semantics, which allow system designers to precisely define and analyze system behaviors. However, specifying and implementing the execution semantics of such DSMLs can be a difficult, costly and error-prone task. Most of the time the concurrency model remains implicit and ad-hoc, embedded in the underlying execution environment. The lack of an explicit concurrency model prevents: the precise definition, the variation and the complete understanding of the semantics of the DSML, the effective usage of concurrency-aware analysis techniques, and the exploitation of the concurrency model during the system refinement (*e.g.*, during its allocation on a specific platform). In this paper, we introduce a *concurrent executable metamodeling approach*, which supports a modular definition of the execution semantics, including the concurrency model, the semantic rules, and a well-defined and expressive communication protocol between them. Our approach comes with a dedicated meta-language to specify the communication protocol, and with an execution environment to simulate executable models. We illustrate and validate our approach with an implementation of fUML, and discuss the modularity and applicability of our approach.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; F.1.1 [*Computation by Abstract Devices*]: Models of Computation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational Semantics

Keywords Metamodeling, Domain-Specific Languages, Model-Driven Engineering, Operational Semantics, Models of Concurrency

1. Introduction

Modern software-intensive systems are becoming too complex to be addressed as a whole. They are usually split into sub-problems, each belonging to a particular domain (*e.g.*, fault tolerance, security, ...). Domain-Specific Languages (DSLs), in opposition to General-purpose Programming Languages (GPLs) such as C or Java, allow the capitalization of domain knowledge into the languages used to solve these sub-problems. DSLs have proven effective at addressing problems of the domain they have been designed for [26]. Still, domain experts are not necessarily well-versed in using programming languages, in which case Domain-Specific Modeling Languages (DSMLs) are more suitable. DSMLs may be executable, which eases the design, verification and validation of the complex systems being developed [6]. In that case, we call them eXecutable Domain-Specific Modeling Languages (xDSMLs).

Since concurrency is at the heart of modern systems such as the Internet of Things or Cyber-Physical Systems, and mod-

ern platforms are providing more and more actual parallelism (*e.g.*, many-core, GPGPU pipelines, distributed platforms, ...), xDSMLs need rich concurrency constructs with clear semantics, allowing both the design of highly-concurrent systems and their refinement to highly-parallel platforms. Designing and implementing such xDSMLs can be a hard, costly and error-prone task. Moreover, the *concurrency model* used by xDSMLs is usually implicit, which hardens its precise specification, hinders the use of concurrency-aware analyses, and prevents its refinement for the study of semantic variation points or for the adaptation to an execution platform.

The language and modeling communities have already studied the definition of a language through the specification of an Abstract Syntax (AS), of a Concrete Syntax (CS), and of a mapping from the AS to a Semantic Domain [20]. Specifying this mapping has been the subject of an extensive literature. Three main approaches have been identified: operational semantics [38], axiomatic semantics [42] and translational semantics (the latter being named denotational when the translation is a mathematical denotation) [16]. These approaches support the specification of concurrency, however the concurrency model is scattered along the specification of the semantics. It results in a concurrency model that is difficult to understand and seldom suitable to analyses like determinism or deadlock freeness [46, Chapter 14]. Most language constructs have semantics which rely on dynamic data (expressions with variables depend on the values of the variables, conditionals depend on the value of the condition expression, iterations, ...). The specification of constructs which pertain to the concurrency concerns is facilitated by the scattering of the concurrency model along the semantics, as all the information used by the semantics (concurrency and data concerns) are available in the same place.

Unlike the approaches from language theory, works on concurrency theory focus on the concurrency, synchronizations and the, possibly timed, causalities between actions. The actions themselves are opaque and thus details of the data manipulations and sequential control aspects they realize are abstracted away. Such models have proven useful for reasoning about concurrent behaviors. Among the more abstract and seminal works on models of concurrency are *Event Structures* [45] and the *Tagged Signal Model* [25]. In these approaches, the non-relevant parts of a model are abstracted away into events with causalities and synchronization relations between them. In other words, concurrency theory focuses on the concurrent control flow of a model in order to ease reasoning on it.

Previous works [7, 8] have proposed to cross-fertilize the language and concurrency theories to design *Concurrency -aware xDSMLs*. In this approach, the mapping from the Abstract Syntax of a language to its Semantic Domain is separated in two parts: an explicit concurrency model, suitable for analyses; and semantic rules, specifying the data manipulations in an operational manner. But so far, the approach lacks the means to specify the data-dependent parts of the concurrency model; *e.g.*, in *conditional statements* the link between the result of the evaluation of a condition expression (computed in the semantic rules) and which branch is consequently executed (determined in the concurrency model).

This paper proposes a dedicated meta-language to define a sound protocol between the concurrency model and the semantic rules. This protocol allows the specification of the data-dependent parts of the concurrency model, by specifying how to determine if an execution path allowed by the concurrency model is consistent with regards to the runtime state of the model. The concurrency model is kept independent from the data, enabling concurrency-aware analyses to be performed in order to ensure properties of the xDSML. We also depict its implementation and integration in a language workbench, the GEMOC Studio, for validation purposes.

The rest of this paper is structured as follows: in Section 2, we present the GEMOC approach to designing Concurrency -aware xDSML and its application to an example language, fUML. Section 3 presents our contribution which consists in a meta-language to define the Communication Protocol between the concurrency model and the semantic rules of an xDSML. Then in Section 4, we present our implementation of the approach. Section 5 evaluates our contribution with regards to its integration in the design of Concurrency-aware xDSMLs and to its applicability, in particular with regards to the control flow patterns identified in [41]. Finally, Section 6 presents related work and Section 7 concludes and proposes perspectives for future work.

2. The GEMOC Approach to the Design of Concurrency-aware xDSMLs

This section presents our approach to the design of Concurrency-aware xDSMLs, illustrated on an xDSML, fUML. Then, we show the shortcoming of the approach with regards to the specification of data-dependent language constructs.

2.1 Separation of Concerns

Previous work [7] has proposed the following approach to design a Concurrency-aware xDSML.

At the heart of a Concurrency-aware xDSML is its Abstract Syntax, which defines the syntactic concepts of the domain and their relations.

The AS is extended with several elements. First, the *Execution Data* (ED), which are the set of classes, attributes and references representing the runtime state of the model. Then, the semantic rules of its concepts, called *Execution Functions* (EF) in our approach. They specify in an operational manner how the runtime state of the model evolves.

The concurrency model defines the pure concurrent control flow of the language, orchestrating the semantic rules but leaving all the data-dependent aspects of the control flow in the semantic rules. In our approach, it is captured in what is called the *Model of Concurrency and Communication*

(MoCC). The MoCC is an *EventType Structure*, specifying at the language level how, for a model, the *Event Structure* defining its concurrent control flow is obtained. The Event Structure represents all the possible execution paths of the *model* (including all possible interleavings of events occurring concurrently).

Finally, the behavioral semantics is obtained thanks to the *Communication Protocol* which maps some of the EventTypes from the MoCC to Execution Functions. This means that at the model level, when an event occurs, it triggers the execution of the associated Execution Function on an element of the model.

2.2 Illustrative Example: fUML

The *Foundational Subset for Executable UML Models* (fUML) [35] is an executable subset of UML which specifies the behavioral semantics of Activity Diagrams. The semantics is inspired mainly from Petri Nets [33].

Figure 1 shows a simple example fUML activity representing a break where we drink something while talking. In this example, the ForkNode splits the execution flow into two concurrent branches. Therefore, "Talk" and "DrinkSomething" happen concurrently (in sequence or in parallel). The fUML specification only requires that both should have happened before executing the JoinNode is allowed.

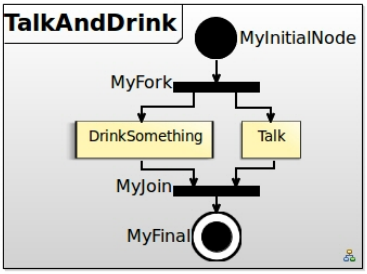

Figure 1. fUML activity modeling a break where we drink while talking.

Figure 2 shows the Execution Data and Functions of fUML as a metamodel to merge with the abstract syntax of fUML. At runtime, tokens are held by ActivityEdges. Nodes can be executed (`ActivityNode.execute()`), realizing various effects depending on their concrete type. For instance, executing an `InitialNode` consists in creating a Token on its outgoing edges. ActivityEdges with a guard have an Execution Function to evaluate their guard.

Figure 3 shows the different concerns, at the model level, of our example: the Event Structure (MoCC at the model level), the Communication Protocol and the Execution Functions (both at the model level). In this representation of event structures, a node is a *configuration*: a set of *event occurrences* which have happened at this point in the execution, in no particular order since it represents a partial ordering. For representation purposes, "..." in a configuration of the event structure represents the collection of event occurrences from

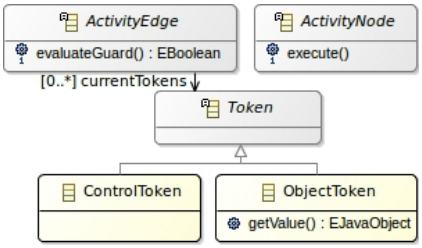

Figure 2. Execution Functions and Data of fUML as a metamodel extending the Abstract Syntax of fUML.

the previous configurations, *e.g.*, {..., e_MyFork} is {e_My InitialNode, e_MyFork}. As for the Communication Protocol, it specifies for instance that the EventType "et_InitialNode" is mapped to `InitialNode::execute()`. At the model level, the event "e_MyInitialNode" (instance of "et_InitialNode") triggers the execution of `MyInitialNode.execute()`.

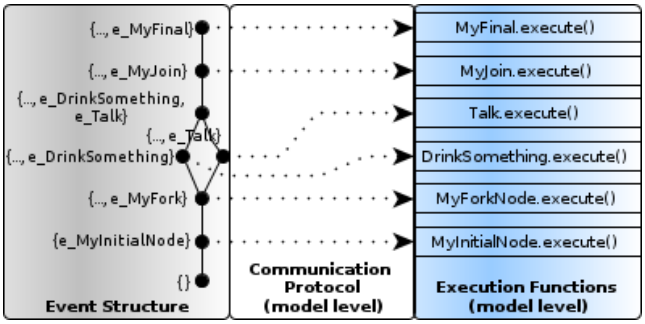

Figure 3. Application of the separation of concerns to the example from Figure 1.

2.3 Shortcoming: Specifying Data-dependencies

Let us consider an fUML construct not used in the example of Figure 1: DecisionNodes. A DecisionNode is a decision point where, depending on the results of the evaluation of guards placed on the outgoing branches, one of the branches will be executed. So far, the approach is not able to specify the semantics of these constructs. The concurrency model is able to represent all the possible future execution paths (*e.g.*, that after a DecisionNode, one and only one of its outgoing branches will be executed). But since it is independent from the data of the domain, it is unable to specify how to choose among the possible paths (*e.g.*, that if a guard returns true, the corresponding branch may be executed).

More generally, many languages have constructs which have concurrency-related semantics depending on data available at runtime. In Finite State Machine (FSM) languages, transitions may have a guard (boolean expression) conditioning whether or not it may be fired. In GPLs, constructs such as the *if-then*, *if-then-else*, *while-loop*, *for-loop*, *switch-case* statements all ultimately rely on a conditional expression evaluated at runtime in order to determine whether the execution should go one way or another. A more thorough identification of data-dependent language constructs is given in Section 5.

This is an important drawback since it prevents systematic extraction of the control flow into a model appropriate for analyses and refinement. It also means that the data-dependent control flow is not fully specified and thus the execution of a model can be inconsistent with the intended semantics of the language. Section 3 proposes a solution to specify the communication between the concurrency model and the semantic rules, allowing in particular the specification of the semantics of data-dependent constructs.

3. Specifying the Semantics of Data-dependent Language Constructs

This section describes our solution to the specification of the semantics of data-dependent language constructs, illustrated on fUML DecisionNodes.

3.1 Example: fUML DecisionNodes

Figure 4 refines the "DrinkSomething" node of the previous fUML example. In this new example, the node "CheckTableForDrinks" randomly returns through its OutputPin either "Coffee", "Tea" or "Neither" to represent which drink we take from the table. The guards outgoing the DecisionNode are such that if "Coffee" is returned, then "DrinkCoffee" is executed, if "Tea" is returned, then "DrinkTea" is executed and if "Neither" is returned, then "DrinkWater" is executed.

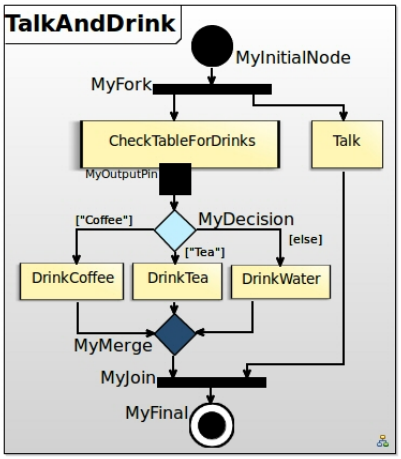

Figure 4. fUML activity modeling a break where we choose a drink on the table while talking.

Note that in fUML, guards on ActivityEdges are restricted exclusively to the outgoing edges of a DecisionNode. Moreover, in fUML a guard is only a value specification (literal or instance) unlike UML where expressions are also allowed. The default guard `[else]` always returns true but its branch is executed only if the other branches cannot be executed. If more than one branch is possible (*e.g.*, if two branches had the same guard), an arbitrary choice is realized by the implementation [34, Page 371]. Meanwhile, the ForkNode splits the execution flow into two concurrent branches. Therefore, "Talk" can be realized in parallel with or interleaved between any of the nodes from the drinking part of the activity.

Figure 5 shows the simplified Event Structure of this new example. Data-dependent causalities are represented in dashed lines. Note that for representation purposes (*i.e.*, to avoid representing all interleavings), we evaluate all the guards concurrently. The specification of fUML only mentions that they should be evaluated in no particular order.

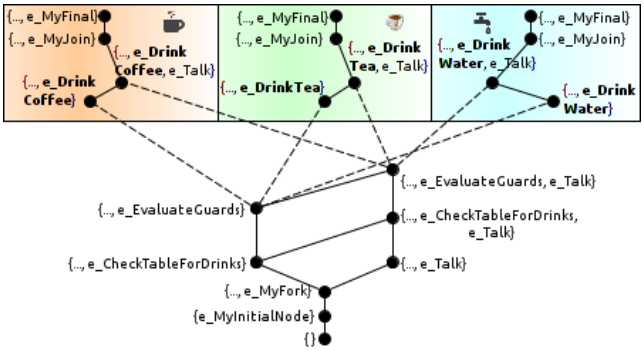

Figure 5. Simplified Event Structure of the TalkAndDrink Activity from Figure 4. Dashed lines represent data-dependent causalities.

If several execution paths are allowed at a point in the Event Structure, it means that there is either *concurrency* or *conflict*. *Concurrency* means that other events are happening concurrently (interleaved or in parallel), in which case the execution paths will eventually merge. This is the case in Figure 5 after the execution of the Fork Node: both branches of the fork are executed concurrently. *Conflict* means that there is a disjunction among the possible execution paths, which ultimately results in different final configurations of the Event Structure. In Figure 5, this is the case in the drinking part of the activity, after evaluating the guards.For a run, only one of the outgoing branches *will* be executed. This means that the three possible execution paths (depending on which drink is available on the table) can never converge.

In case of conflict in the approach we have described so far, an arbitrary choice among the possible execution paths is made by an heuristic of the runtime. This solution is enough for conflicts independent of any data (the language is thus indeterministic). But conflicts can also depend on data available at runtime in the model, as is the case for fUML DecisionNodes. In that case, realizing an arbitrary choice can lead to a meaningless execution of the rest of the model, or to errors (*e.g.*, when trying to execute nodes with no tokens on their incoming edges). Data-dependent conflicts should thus be resolved with respect to the runtime state of the model.

The concurrency model of fUML is able to specify that after executing a DecisionNode, each of its branches may or may not be executed. It is also able to specify that ultimately only one of the branches will actually be executed. Additionally, it is able to specify that if one of the branches that may be executed has the default guard "else", it will only

be executed if it is the only one allowed. But it is not able to specify whether a branch may or may not be executed: this depends on the result of the evaluation of the guard, to which the concurrency model does not have access since it is computed in the semantic rules. Therefore, we need a means to be able to specify that for any branch, if the evaluation of the guard returns "True" then the branch may be executed, and if it returns "False" then it may not be.

3.2 Evolution of the Concurrent Executable Metamodeling Approach

We have identified two natures of Execution Functions. *Modifiers* are functions with side-effects, whose role is to update the runtime state of the model when executed. In previous work [7], Execution Functions were only of this nature. For instance, executing a node in fUML modifies the runtime state of the incoming and outgoing edges. *Queries* are side-effect-free functions whose role is to return runtime information, either about the model itself or computed based on data from the model. For instance, evaluating the guard of an edge in fUML is a Query. Note that this taxonomy is mainly conceptual: an Execution Function with side-effects and returning runtime information can be practical when designing the semantic rules of the xDSML. In that case, special attention must be paid to when and which data this Execution Function modifies and uses to compute its result.

When a Query is executed, it returns a value, hereafter named *Feedback Value*. Our goal is to specify which paths from the Event Structure are inconsistent with regards to the runtime state of the model. This is done thanks to an interpretation of the Feedback Value, specified in a specification named the *Feedback Protocol*. Like the other specifications constituting a Concurrency-aware xDSML, the Feedback Protocol is specified at the language level, but it provides the information for execution at the model level. This means that the Feedback Protocol specifies execution paths in terms of EventTypes (from the MoCC), but that at execution time, its application will range over occurrences of Events from the Event Structure.

In fUML, evaluating the guard of an edge returns a boolean value. If this value is true, then the branch may be executed, otherwise it may not be. The role of the Feedback Protocol is to remove the execution path corresponding to a branch if its guard value was false. Note that the Feedback Protocol not only removes some of the possible execution paths outgoing the current configuration, but also some execution paths outgoing children (or further descendants) configurations of the current configuration. For instance, if "Coffee" has been found on the OutputPin, it means that tea is not available so the Feedback Protocol prunes the execution path leading to "e_DrinkTea". But it must also prune the execution paths leading to "e_DrinkTea", such as the path where after evaluating the guards, the occurrence of "e_Talk" happens. Figure 6 shows the execution paths that must be pruned from the simplified Event Structure of the example model in the case of an execution where coffee is available.

Let us clarify the Feedback Protocol and how it is applied. We consider an Event Structure E. E is defined by $< Evt, \mathbb{C}, \vdash >$, where Evt is a set of Events, $\mathbb{C}$ is an ordered set of consistent configurations and $\vdash$ is the enabling relation [45]. A configuration is a set of events that have occurred by some stage in the process. Also, any event in a configuration should have been enabled by another event in a previous configuration (or by the null set for uncontrolled events like the initial one). We denote $path(c_1, c_2)$ two "causal" configurations, *i.e.*, two configurations such that:

$$\exists e \in c_2, c_1 \vdash e \wedge \nexists c \in \mathbb{C}, c < c_1 \wedge c \vdash e$$

In other words, the configuration c_2 contains at least one event directly enabled by c_1.

Based on this, we can define an event structure as a triplet $< Evt, \mathbb{C}, \mathbb{P} >$ where $\mathbb{P}$ is the set of paths between the configurations in $\mathbb{C}$. There exists two different kinds of paths in $\mathbb{P}$, *i.e.*, $\mathbb{P} \triangleq \mathbb{P}_I \cup \mathbb{P}_D$. $\mathbb{P}_I$ are the paths independent from the runtime state of the model while $\mathbb{P}_D$ are the data-dependent ones. Let us denote $rts(c)$ the runtime state of the model at the configuration c of the Event Structure. When $path(c_1, c_2) \in \mathbb{P}_D$, and its dependency is towards the runtime state of the model at a specific configuration c, we denote it as $path(c_1, c_2)_{rts(c)}$. This means that depending on the runtime state of the model at a certain point c of the Event Structure (where c precedes c_1 and c_2), going from c_1 to c_2 *may be possible*. Let us denote as $f_{path(c_1,c_2)_{rts(c)}}$ the function that determines if the path from c_1 to c_2 may be taken, depending on an interpretation of $rts(c)$. It returns a boolean value: either the path is allowed or it is not.

For instance on Figure 6, the path from {..., e_Evaluate Guards} to {..., e_DrinkCoffee} depends on the runtime state of the model obtained at configuration {..., e_Evaluate Guards}. The path from {..., e_EvaluateGuards, e_Talk} to {..., e_DrinkTea, e_Talk} also depends on the runtime state of the model obtained at configuration {..., e_EvaluateGuards}. For both of these paths, the runtime state of the model is represented by the respective results of the guards on the

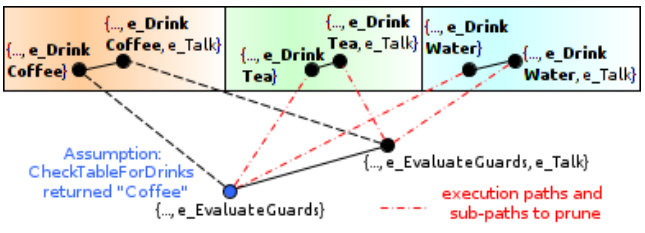

Figure 6. Close-up on the simplified Event Structure of the example model where the Feedback Protocol is applied. Dashed lines represent data-dependent causalities. Red dots-and-dashes lines represent the execution paths that must be pruned as a result of the Feedback Protocol if the "CheckTableForDrinks" node returned "Coffee".

edges “Decision2DrinkCoffee” and “Decision2DrinkTea”. The results are respectively “True” and “False” when the OutputPin of “CheckTableForDrinks” returns “Coffee” (as is assumed in Figure 6). Therefore, the function f will return true for the first path (it is compatible with the runtime state of the model) and false for the second one (it is inconsistent with regards to the runtime state of the model).

The Feedback Protocol must specify (at the language level, *i.e.*, in intention) the set of data-dependent execution paths (*i.e.*, $\mathbb{P}_D$) together with the set of functions f_p (where f_p determines whether a path $p \in \mathbb{P}_D$ may be taken or not). This specification must be independent of any model, but be applicable to any model conforming to the abstract syntax of the language. For a specific model, applying the Feedback Protocol consists in removing the execution paths from $\mathbb{P}_D$ that are inconsistent with the runtime state of the model. It cannot add any paths in $\mathbb{P}$, nor remove any paths from $\mathbb{P}_I$.

3.3 Specifying the Communication Protocol

The Communication Protocol of a Concurrency-aware xDSML is specified at the language level, and is constituted of a *Mapping Protocol* and a *Feedback Protocol*. Figure 7 shows its integration into our approach as a class diagram.

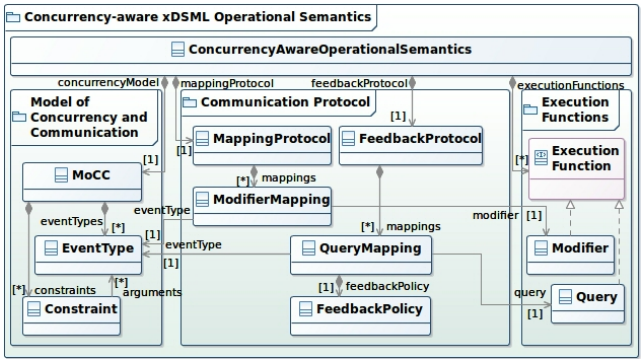

Figure 7. Class Diagram of our approach towards concurrency-aware operational semantics of an xDSML

The Mapping Protocol is constituted of *ModifierMappings* between an EventType and a Modifier. At execution time, when an instance of the EventType (an Event from the Event Structure) occurs, it triggers the execution of the Modifier. This results in an update of the runtime state of the model.

The Feedback Protocol is constituted of *QueryMappings*, each having a *Feedback Policy*. A QueryMapping is a mapping between an EventType from the MoCC and a Query. A Feedback Policy is a function with one parameter (the return type of the associated Query) which returns occurrences of EventTypes. The Feedback Policy is in charge of interpreting the Feedback Value and returning the sets of occurrences of EventTypes inconsistent with the runtime state of the model. By occurrences of EventTypes, we mean specifications designating certain occurrences of the instances of an EventType. For instance in our example model, if evaluating the guards tells us that coffee will be drunk, then the next occurrence of “e_DrinkTea” and the next occurrence of “e_DrinkWater” are inconsistent with regards to the runtime state of the model and should not happen. This means that for an ActivityEdge whose guard evaluation returns false, the Feedback Policy returns the first occurrence of its associated EventType “et_targetOfEdge”. In the case of our example, “e_DrinkCoffee” and “e_DrinkTea” are instances of “et_targetOfEdge”. At runtime, when we progress through the Event Structure and an occurring event is mapped by a QueryMapping, the associated Query is executed and its return value is passed as input to the associated Feedback Policy. The Feedback Policy then returns the set of event occurrences incompatible with the runtime state of the model. When considering how to progress on the Event Structure afterwards, all the execution paths leading to configurations with the occurrences incompatible with the runtime state of the model are removed. This way, only the paths compatible with ($p \in \mathbb{P}_D$ for which f_p returned true) or independent from the runtime state of the model ($p \in \mathbb{P}_I$) are available.

The Feedback Policy must be a function in the mathematical sense: every input (instance of the return type of the Query) must have an output. Otherwise, the selection among the possible execution paths is done by the heuristic of the runtime as it corresponds to a situation of indeterminism. As a result, the execution becomes meaningless and can lead to errors. To ensure that this does not happen, the Feedback Policy should always be able to return a result, whatever the Feedback Value is. In short, this means that the Feedback Policy must define a default result, which is returned if no other result is computed for the Feedback Value.

3.4 Pragmatics of the Approach

Practically, computing the whole Event Structure may be complex or impossible. If the model is very large or highly parallel, then the exponential number of configurations and execution paths (possibly infinite) makes it either too costly to compute or too big to be usable. Let us consider the minimal situation, where we are capable of computing only the children configurations of a configuration.

Since the event structure is only partially constructed during a specific execution of the model, we do not have all the paths (and furthermore not all the data-dependent paths). Therefore, applying the Feedback Protocol cannot consist in pruning execution paths in the event structure. Instead, the Feedback Policies only specify the EventTypes which are inconsistent with regards to the runtime state of the model, and at execution time *all the occurrences* of the corresponding instances of the EventType are forbidden. Forbidding an event from occurring results in pruning the corresponding execution path in the implicit event structure. However, it should not prune other occurrences of that same event which depend on another runtime state of the model.

To counteract this issue, we add the following role to the Feedback Policy: its interpretation of the Feedback Value must return the set of EventTypes inconsistent with the run-

time state of the model *and* the set of EventTypes which are data-dependent and consistent with the runtime state of the model. This way, the next occurrences of these consistent EventTypes are considered as the limit after which the occurrences of the inconsistent EventTypes do not represent a data-dependent decision anymore. Thus, after the consistent EventTypes have occurred, forbidding the inconsistent EventTypes ceases. This adds the following constraint: the MoCC should not allow situations where different occurrences of the same event depend on Feedback Policies (possibly several occurrences of the same policy) which can be applied at the same time. When considering two queries, and their Feedback Policies overlap in terms of which events are compatible or incompatible, then the MoCC should not allow these two queries to overlap the application of the Feedback Policy of the other query. This means that the second query should never be executed between an execution of the first query and occurrences of the compatible events of the Feedback Policy of that first query. Otherwise, it is possible that the MoCC falls in a state of deadlock, halting the execution.

4. Implementation

We have implemented our approach in a language workbench, the GEMOC Studio[1]. Our implementation is based on previous work [7]. It is integrated in the Eclipse Modeling Framework (EMF) [14] to benefit from its large ecosystem.

4.1 Existing Elements

To specify the Abstract Syntax, we rely on Ecore, the EMF implementation of EMOF [37]. The associated static semantics are expressed in terms of Object Constraint Language (OCL) invariants [36]. Both EMOF and OCL are standards from the Object Management Group (OMG)[2].

The Semantic Rules are implemented using the Kermeta 3 Action Language (K3AL) [13], built on top of xTend [4]. K3AL allows the definition of aspects for Ecore metaclasses, allowing us to weave the additional classes, attributes, references and operation implementations specifying the Execution Data and the Execution Functions. K3AL, just like xTend, compiles into Java Bytecode and provides an executor based on the Java Reflection API to dynamically execute the Execution Functions.

To specify the MoCC, we use MoCCML [10], a declarative meta-language designed to express constraints between events which can be capitalized into libraries agnostic of any AS. The definition of the EventType Structure is realized using the Event Constraint Language (ECL) [11], an extension of OCL which allows the definition of EventTypes for concepts from the AS. It can also use constraints defined in MoCCML to specify how the Event Structure at the model level must be obtained. The MoCC is compiled to a *Clock Constraint Specification Language (CCSL)* [27] model interpreted by the TimeSquare [12] tool. For the practical reasons mentioned in 3.4, TimeSquare only provides the next set of possible configurations.

At runtime, an Execution Engine written mostly in Java coordinates the K3AL interpreter and the CCSL solver to realize the execution of a model conforming to a Concurrency-aware xDSML. Figure 8 shows the sequence diagram for an execution step. First, the Execution Engine retrieves from the CCSL Solver the next set of possible configurations (scheduling solutions). Its heuristic then selects one solution among the possible ones. A default implementation consists in letting the user realize this selection so as to manage indeterministic situations manually when developing an xDSML. Based on the Mapping Protocol, the set of Execution Functions to execute is deduced from the selected solution. All the execution functions are executed in parallel, resulting in an updated runtime state of the model.

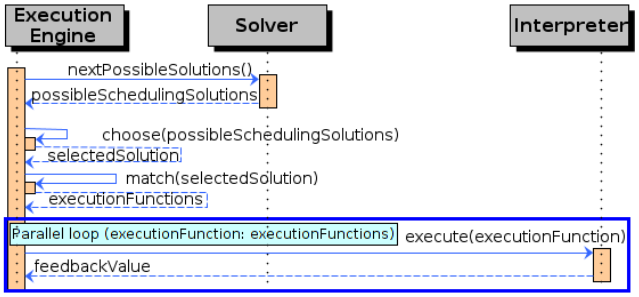

Figure 8. Sequence Diagram representing one step of execution of a model conforming to a Concurrency-aware xDSML.

4.2 Communication Protocol

Our meta-language to specify the Communication Protocol (both the Mapping Protocol and the Feedback Protocol) is called *GEMOC Events Language* (GEL). Figure 9 shows an excerpt from its Abstract Syntax, specified as an Ecore model. It has a textual concrete syntax developed using Xtext [4].

In GEL, *Domain-Specific Events* implement both the ModifierMapping and QueryMapping concepts. If the referenced Execution Function is a Query, then a Feedback Policy may be specified. A Feedback Policy is constituted of at least two rules, including a default one. A Feedback Rule is constituted of a Predicate on the return type of the associated Query, and of an *allowed* EventType (MoccEvent). Since the consequences of all the rules of a policy constitute the set of data-dependent EventTypes, we can specify in the rules either the consistent ones or the inconsistent ones and deduce the others by getting its complement. In GEL, we have chosen to specify in the rules the EventTypes consistent with regards to the runtime state of the model. Listing 1 defines the Feedback Protocol for fUML using GEL. The GEL compiler transforms this specification into a model-level specification using a model conform to the AS of the language. For every instance of the EventType "mocc_evaluateGuard", an instance

[1] `http://gemoc.org/studio/`

[2] `http://omg.org/`

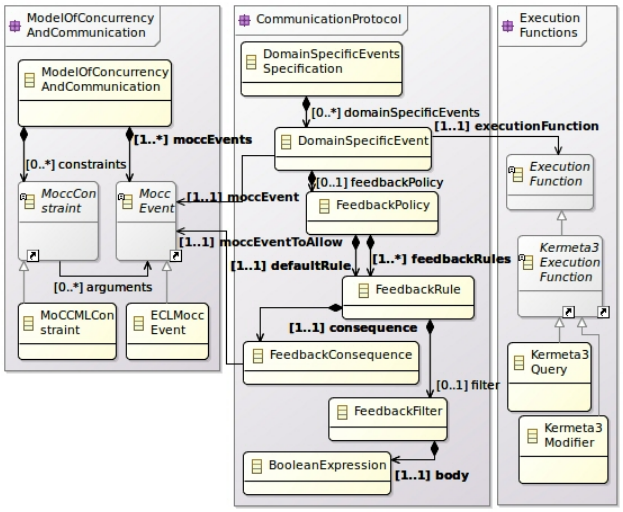

Figure 9. Excerpt from the Abstract Syntax of GEL

of the DSE "EvaluateGuard" is created, and its triggered Execution Function and allowed MoccEvents are adapted to their counterparts at the model level.

At runtime, when an occurrence of the Event "mocc_evaluateGuard_Decision2DrinkCoffee" happens, the associated Execution Function `Decision2DrinkCoffee.evaluateGuard()` is executed, and its result is stored in the variable "result". If "result" is true, the first rule is applied (event "mocc_mayExecuteTarget_Decision2DrinkCoffee" is allowed), otherwise the default rule is applied (event "mocc_mayNotExecuteTarget_Decision2DrinkCoffee" is allowed).

More generally, when a query returns a value, we determine the set of rules to apply. The value is passed through all the rules of the associated policy. For each non-default rule, if the feedback value validates the predicate then the rule must be applied, else it must not be applied. If none of the rules is to be applied, then the default rule is applied. Based on this set, we obtain the MoccEvent instances consistent with the runtime state of the model (consequences of all the rules to apply). They represent the data-dependent execution paths. The set of incompatible MoccEvent instances is deduced from the rest of the specification: it is the set of the consequences of all the rules which are not applied, minus the consequence of the default rule if it is already in the compatible MoccEvent instances. All upcoming Scheduling Solutions containing occurrences of incompatible MoccEvents are removed until the allowed MoccEvent instances have all occurred in the selected solutions. This can be seen as dynamic constraints being added to the MoCC and removed when they are not relevant anymore (the compatible MoccEvents have occurred).

Listing 1. The `EvaluateGuard` Domain-Specific Event (QuerryMapping) and its Feedback Policy defined in GEL.

```
1 DSE EvaluateGuard:
2   upon mocc_evaluateGuard
3    triggers ActivityEdge.evaluateGuard returning result
4   feedback:
5     [result] => allow ActivityEdge.mocc_mayExecuteTarget
6     default => allow ActivityEdge.mocc_mayNotExecuteTarget
7   end
8 end
```

Figure 10 shows the changes of the runtime as a modification of the sequence diagram shown previously on Figure 8. After retrieving the possible solutions from the Solver, a new entity, the Protocol Engine, filters out the solutions containing occurrences of incompatible events. Based on the Communication Protocol, the set of Execution Functions to execute is deduced and they are all executed. If a result is returned by an Execution Function, and there is an associated Feedback Policy, then it is interpreted by the Protocol Engine to create what we call dynamic constraints. They correspond to the specification of which MoccEvents are incompatible or compatible with the runtime state of the model, so as to determine which events are to be forbidden for the next steps of execution, and until when.

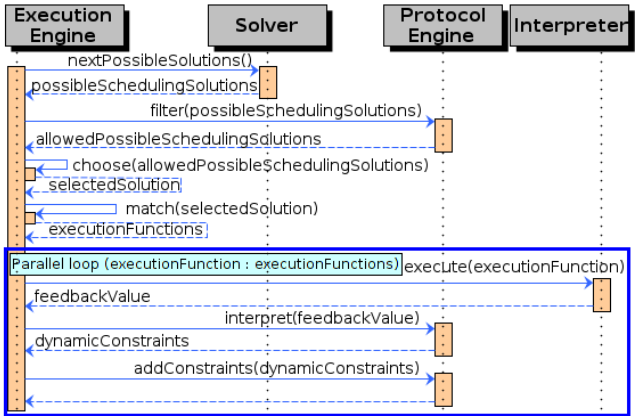

Figure 10. Updated Sequence Diagram representing one step of execution with the Feedback Mechanism included.

4.3 Execution of the Example Model

A video showing the execution of the example model is available at `http://gemoc.org/sle15/`. It also provides the GEMOC Studio and the workspaces containing the source for our implementation of fUML and the example model. We have used Groovy as an action language to define the body of the fUML Actions. Figure 11 shows a close-up on the DecisionNode part of the Event Structure. Based on the value returned by each guard, an event corresponding to each guard occurs (the "May..." if true was returned, "MayNot..." if false was returned). Since the branch for drinking water has the default guard "else", it is only executed if none of the other two branches can be executed. If both tea and coffee are available (although in our model this will never happen), then an arbitrary choice is made by the runtime.

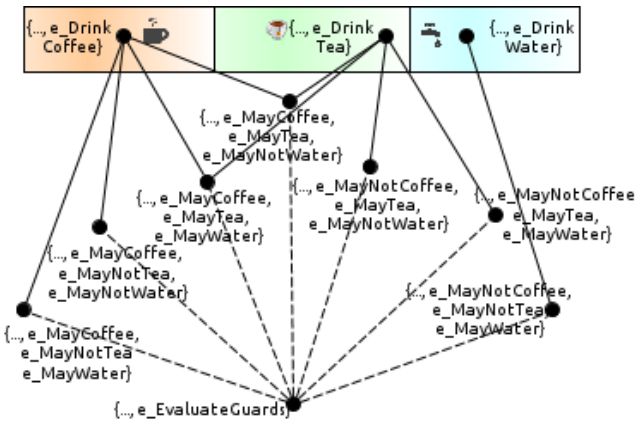

Figure 11. Close-up on the DecisionNode part of the simplified Event Structure of the example model

5. Evaluation

We evaluate the integration of our contribution into the design of Concurrency-aware xDSMLs with respect to the original objectives of the approach. We also identify its applicability. Then we discuss some limitations of our approach. Finally we compare our approach to the Structural Operational Semantics approach [38] which originally inspired this work.

5.1 Modularity

We have set up the Feedback Protocol alongside the Communication Protocol in order to retain the modularity of the separation of concerns of our approach. Another solution would have been to augment the concurrency model with the specification of which conflicts are data-dependent and how to solve them. However this would have tied the concurrency model directly to the data of the domain, which would have compromised the possibility to realize concurrency-aware analyses. By keeping this modularity, the initial advantages of making explicit the concurrency remain. The concurrency concerns of the xDSML are captured in an explicit model (in our case, the MoCC), independent from the data of the domain. This enables the use of concurrency-aware analyses, such as determinism or deadlock freeness. The concurrency model can also be refined to fit the language to a specific platform (depending on the concurrency facilities available).

The concurrency model and the semantic rules are independent. Thus, both can be specified, implemented and tested independently. Not only does this ease the design of concurrency-aware xDSMLs, but it also facilitates the implementation of Semantic Variation Points (SVPs). For instance, a SVP can consist in changing an Execution Function implementation to create or consume more tokens. This also means we can change the action language used for fUML without having to modify the MoCC. SVP implementations can also consist in modifying the concurrency model (*e.g.*, forcing branches of a ForkNode to be executed in sequence). SVP implementations can also be concretized in the Communication Protocol (by modifying the mapping between EventTypes and Execution Functions), including in the Feedback Protocol. In the case of DecisionNodes, we could choose to interpret a "False" result as allowing a branch to be executed and a "True" result as preventing a branch from being executed. Although it denatures the intended semantics of fUML, these variations can be used to realize alternative versions of fUML.

5.2 Applicability of our Approach

Although illustrated on fUML, our solution is entirely at the meta-language level and is thus not specific to a particular language. This genericity means that our solution can be applied to any xDSML built using our concurrency-aware approach. But identifying the situations in which a language construct will need a Feedback Protocol can be complex. Below, we give specific examples of xDSML constructs which need a Feedback Protocol to be specified correctly using our approach, and also generalize to control flow patterns in order to give a broader view of the applicability of our contribution.

Language constructs with data-dependent control flow are key to the semantics of many xDSMLs. For instance, in some variations of *Finite State Machine* (FSM) languages, guards of transitions may be boolean expressions written using an embedded expression language. In that case, the concurrency model is unable to specify how the evaluation of a guard relates to its transition being allowed to fire or not. As in fUML, a Feedback Protocol is needed to specify that if a guard evaluates to true, then the associated transition may be fired, else it may not be. Another example of this is the *Simulink* [28] language. In Simulink a model is usually a dataflow of blocks where the output of a block is used as input for the next block. However, the output of a block can also be connected to a special port of the next block, the *EnablePort*[3]. In that case, the output of the first block is compared to 0. If it is greater, then the block with the EnablePort is executed, else it is not. In that case, the Feedback Protocol is required to specify that if the output value of the first block is greater than zero, then the second block will be executed, else it will not. Note that this requires being able to manage the same data type as returned by a Simulink block, and then to compare it to 0. This assumes a certain expressive power from the meta-language used to specify the Feedback Policy as discussed later in the limitations of the approach.

More generally, any language construct whose semantics rely on conditionals where the condition expression can be based on runtime data from the model will raise the same problem as fUML DecisionNodes: the concurrency model is able to enumerate all the possible scenarios, but it does not have the expressive power to specify how to choose among them. We rely on the classification of control flow constructs in workflow systems provided in [41] to identify more formally the applicability of our approach. In this study, the

[3] http://fr.mathworks.com/help/simulink/slref/enable.html

authors have identified 43 patterns describing the control flow perspective of workflow systems. They give a formal description of their semantics in the Coloured Petri-Net formalism. These patterns are usually handled by a language construct (or a combination of constructs) in formalisms such as BPMN, UML Activity Diagrams, BPEL, ... Among these patterns, 9 have semantics which, described using our approach, could cause data-dependent conflicts in the concurrency model and thus require a Feedback Protocol. Patterns depending on the evaluation of a condition expression (*e.g.*, *Exclusive Choice*, akin to fUML DecisionNode; *Multi-Choice*, akin to a UML ForkNode with guards on outgoing branches; ...) are typically concerned. Patterns based on iterations (*e.g.*, *Arbitrary Cycles*, corresponding to loops based on *goto* statements; *Structured Loop*, corresponding to repetitions based on dedicated language constructs such as *while...do* or *repeat...until*) also rely on the evaluation of a condition expression. As stated by the authors, "*Although initially focused on workflow systems, it soon became clear that the patterns were applicable in a much broader sense*" and "*Amongst some vendors, the extent of patterns support soon became a basis for product differentiation and promotion.*" [41]. Considering the number of these patterns that could cause data-dependent conflicts, specifying the Feedback Protocol is thus a mandatory step when specifying a concurrency-aware xDSML with constructs defining complex control flow structures.

5.3 Limitations

The Feedback Protocol is very dependent on the expressive power of the meta-language used to specify the Feedback Policies. We explain the minimal requirement and how it impacts the rest of a concurrency-aware xDSML specification. We also present the consequences of the Feedback Protocol on the design of the concurrency model.

5.3.1 Expressivity Requirements of the Meta-language for the Feedback Policy

At the very least, the meta-language used to specify the Feedback Policies should be able to handle boolean results. An Event Structure representing the concurrent control flow of an executable model can be transformed into another Event Structure, equivalent with regards to the execution semantics (*e.g.*, the same Modifiers are executed in the same order) but with at most 2 possible executions paths in its conflicts. Figure 12 shows this equivalence. On the left, the conflict between the three execution paths can be resolved using a query associated to "a". It is equivalent to the structure on the right: only one of b, c, d will occur ultimately. But in the latter structure, conflicts are between only two execution paths, for which a boolean query provides enough information. The difference between these two structures is the intermediary event "not_b". But since this new event is mapped to a query, which is side-effect-free, it does not interfere with the semantics of the model being executed. Although the two structures are not equivalent per se, they are equivalent with regards to the execution semantics of a model.

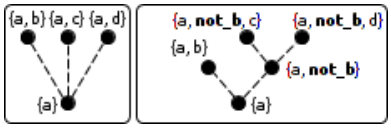

Figure 12. Equivalent Event Structures with regards to the execution semantics of a model if "not_b" is not mapped to a Modifier.

Even though a meta-language for the Feedback Policy able to handle boolean inputs is enough (so long as the queries are designed to return the right boolean values), queries are conceptually allowed to return any type of value. Implementors of our approach may want to either provide strong expressive power for the meta-language of the Feedback Policy, so that any return type can be handled, or restrict the return types allowed for Queries depending on the expressive power of the meta-language used for the Feedback Policy.

5.3.2 Consequences of the Feedback Protocol on the Design of the Concurrency model

There may be an indefinite number of event occurrences between the execution of a query and its consequence. Therefore, describing a partial ordering based on the *absence* of an event occurrence is not possible. It is possible that the event occurrence is simply "delayed" due to concurrent parts of the model execution being interleaved. This is why in fUML, if the guard of the branch for drinking coffee returns false, we cannot simply restrict the event "e_MayCoffee" to *not* occur. Instead, we must model explicitly that we may *not* drink coffee by having the additional event "e_MayNotCoffee" occur. More generally, we take the following systematic approach when designing the part of the concurrency model that will interact with the Feedback Protocol: each possible decision should have its own EventType in the concurrency model. The rest of the concurrency model is then connected to these events through the partial ordering. Moreover, these additional EventTypes should be mapped to the same context (concept from the AS) as the EventType which originally triggered the QueryMapping. Having the same context for a QueryMapping and its FeedbackPolicy guarantees that at the model level, each FeedbackPolicy's consequence events (the compatible and the incompatible ones) are distinct for each instance of the context's instance. For example in our fUML model, each branch has its own event "e_Drink...". Thanks to this guarantee, this limits the possible conflicts between overlapping queries and their consequences. The only possible conflicts remaining are between a query's multiple occurrences, which should not be allowed to happen as explained in subsection 3.4.

Our contribution allows the specification of data-dependent constructs whose semantics interact with the control flow of the language. However, since the concurrency model specifies in intention all the possible execution paths, this means

that all possible interactions must have been accounted for when designing the concurrency model. Language constructs such as *Continuations* are powerful, enabling complex customizations of the control-flow, but also dependent on data available in the semantic rules. Continuations can be realized in our approach, provided the concurrency model has been designed in a flexible manner and that the Communication Protocol and Semantic Rules have been specified accordingly. For the same reasons, some control flow patterns identified in [41] are limited. The concerned patterns are those pertaining to the creation of additional instances. They can be dealt with, if the concurrency model and semantic rules have been designed in such a way that the resulting Event Structure has pre-established events for potential new instances created.

5.4 Parallel with Structural Operational Semantics

In the Structural Operational Semantics (SOS) approach [32, 38], the operational semantics of a language is specified as a set of inference rules defining transitions of the system state. The premise of a rule defines whether or not a transition may be executed. The concurrency model is thus spread throughout both the premises of the rules and the algorithm used to schedule in which order the allowed rules are executed. The conclusions of the SOS rules correspond to the Modifiers we have defined in section 3, since they correspond to how the runtime state of the model being executed evolves.

6. Related Work

Concurrency models have been the subject of theoretical computer science for a long time, establishing well-known concepts such as Petri Nets [33]; Process Algebras [21, 30, 31]; the Actor Model [1]; Event Structures [45]; Tagged Signals [25]. Some tools allow the design of multi-paradigm *systems* based on models of concurrency, such as Ptolemy [39] or ModHel'X [19], but the concept of concurrency model has not been reified to the language level.

Some frameworks, libraries or languages promote a form of separation between the control flow and the operational aspects of an application. Event-driven programming such as promoted by node.js usually uses a form of Callback, inspired from the Continuation-passing style of programming [2]. This can lead to a complicated programming style commonly known as "Callback hell", where the control flow is spread all over the code, and the code itself is hard to read, debug and refactor. This is one of the many reasons for the development of languages or libraries based on the Actor Model [1], such as Erlang [3] or Scala's Akka actors [18] ; or based on Communicating Sequential Processes (CSP) [21] such as Occam [22], Go [17] or Clojure [5, Chapter 6]. In these approaches, the data-dependent parts of the control flow are specified in an ad-hoc manner at the model level, whereas in our approach the specification of the Feedback Protocol is done at the language level. Moreover, these languages or frameworks merely provide facilities or native constructs to use a specific model of concurrency. There is no guarantee that a program is indeed using the advocated model of concurrency. xDSMLs are usually smaller languages with the corresponding domain knowledge being captured in the language constructs themselves, including the concurrency concerns. Therefore, the model of concurrency of the program (model) is guaranteed by the definition of the concurrency of the *language*.

DSL editors, usually called Language Workbenches [15], such as Metacase's MetaEdit+ [24], JetBrain's MPS [44], Microsoft's DSL Tools [9], Spoofax [23] or Rascal [43], and executable metamodeling approaches such as the K Framework [40] or xMOF [29] do not provide an explicit model of the concurrency of the semantics of the DSLs created. Instead, the implicit model of concurrency is usually the one used by the hosting platform or language. This means that running concurrency-aware analyses on the language or refining it for a specific platform is complex. In our approach the concurrency model is explicit and specified in an appropriate meta-language, making it more suitable for analyses and refinement.

7. Conclusion and Perspectives

We have proposed a concurrent executable metamodeling approach which allows the specification of Concurrency-aware xDSMLs. Previously, this approach did not support the specification of data-dependent language constructs. This issue hindered many languages from being specific correctly, as illustrated by the proportion of control flow patterns whose semantics relies on runtime data, identified in [41]. Thanks to our contribution, which extends the Communication Protocol between the concurrency model and the semantic rules with a Feedback Protocol, the approach now support the specification of these constructs. We have also described how the Feedback Protocol is applied at the model level so as to respect the concurrency model of the language. Our approach has been illustrated on fUML, a language in which branches outgoing a DecisionNode are allowed depending on the evaluation of their guard, and implemented in a language workbench, the GEMOC Studio. Our approach allows the use of concurrency-aware analyses on the concurrency model of a language, and eases the comprehension and variability of a language's semantics. In particular, by varying the semantic rules and/or the concurrency model, semantic variation points can be realized. The concurrency model can also be refined to fit a specific execution platform (*e.g.*, single core or multi-core, distributed, ...).

In this paper, we have considered the semantic rules as opaque and blocking, *i.e.*, they cannot communicate with the concurrency model before they are completed. In order to create composite semantic rules (to reuse existing semantic rules, *e.g.*, evaluating a binary expression can be done by evaluating both operands first), we need to establish a communication between an executing semantic rule and the con-

currency model (which specifies whether operands should be evaluated concurrently or in sequence). Future work should provide an adequate paradigm for this communication and its specification. Additional work should also focus on the scalability of our approach to larger languages and models, and investigate the possible relations between logical time (in the concurrency model) and domain time (in the AS) or physical time (in the runtime of the semantic rules).

Acknowledgments

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

References

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

[2] A. W. Appel. *Compiling with continuations*. 2007.

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

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

[5] P. Butcher. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. 2014.

[6] B. Combemale, X. Crégut, and M. Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *APSEC*. IEEE, Dec. 2012.

[7] B. Combemale, J. De Antoni, M. Vara Larsen, F. , O. Barais, B. Baudry, and R. France. Reifying Concurrency for Executable Metamodeling. In *SLE*, 2013.

[8] B. Combemale, C. Hardebolle, C. Jacquet, F. Boulanger, and B. Baudry. Bridging the Chasm between Executable Metamodeling and Models of Computation. In *SLE*, 2012.

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

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

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

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

[13] DIVERSE-team. Github for k3al, 2015.

[14] Eclipse Foundation. EMF homepage, 2015.

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

[16] L.-a. Fredlund, B. Jonsson, and J. Parrow. An implementation of a translational semantics for an imperative language. In *CONCUR*, LNCS, pages 246–262. Springer, 1990.

[17] Google. Golang, 2015.

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

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

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

[21] C. A. R. Hoare et al. *Communicating sequential processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.

[22] G. Jones. *Programming in OCCAM*. 1986.

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

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

[25] E. A. Lee and A. Sangiovanni-Vincentelli. Comparing models of computation. In *ICCAD*. IEEE, 1997.

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

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

[28] MathWorks. Simulink, 2015.

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

[30] R. Milner. *A calculus of communicating systems*. springer-Verlag Berlin, 1980.

[31] R. Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[32] P. D. Mosses. *Foundations of modular SOS*. Springer, 1999.

[33] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

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

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

[36] OMG. OCL specification v2.4, 2014.

[37] OMG. MOF specification v2.5, 2015.

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

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

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

[41] N. Russell, A. H. Ter Hofstede, and N. Mulyar. Workflow control-flow patterns: A revised view. 2006.

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

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

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

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

[46] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.