Data-dependent Constructs in Concurrency-aware eXecutable Domain-Specific Modeling Languages[⋆]

Florent Latombe[1], Xavier Crégut[1], Benoît Combemale[2], Julien De Antoni[3], and Marc Pantel[1]

[1] University of Toulouse, IRIT, Toulouse, France
first.last@irit.fr
[2] University of Rennes I, IRISA, Inria, Rennes, France
first.last@irisa.fr
[3] Univ. Nice Sophia Antipolis, CNRS, I3S, Inria, Sophia Antipolis, France
first.last@polytech.unice.fr

Abstract. Following Plotkin's Structural Operational Semantics (SOS) principles, execution semantics for Domain-Specific Modeling Languages (DSMLs) can be specified as a set of guarded rewrite rules. The emergence of modern highly-parallel platforms calls for DSMLs where concurrency is a first-order concept (Concurrency-aware DSMLs). In SOS, the concurrency model is encoded either in the rule structure representing the execution context (making it costly to execute), in the internal states of the guards (making it difficult to analyze) or implicitly inherited from the meta-language used (making it implementation-dependent). Some languages, such as Erlang or Akka, make explicit the concurrency model and how it triggers the concurrency agnostic atomic rules. But concurrency often depends on the results of those rules, e.g. in conditional statements, which raises the need for an explicit and unambiguous way to specify the communication between the rewrite rules and the concurrency model. We propose a dedicated meta-language to specify how the model of concurrency of a DSML is conditioned by the results of rewrite rules and illustrate our approach on fUML.

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

1 Introduction

Software-intensive systems evolve in heterogeneous environments (including distributed or parallelized environments) where the implementations of concurrency and communication concerns can vary greatly. Moreover, these systems are usually designed by various stakeholders, whose particular expertise can

[⋆] This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011) be capitalized in eXecutable Domain-Specific Modeling Languages (xDSMLs). Concurrency-aware xDSMLs make explicit the Model of Concurrency and Communication (MoCC), which favors adaptability to various possible execution environments and eases reasoning about the concurrency of an xDSML. This paper focuses on dealing with language constructs whose semantics depend on runtime data from the domain of the xDSML, e.g. in *Conditional Statements* where the branch taken depends on the result of the evaluation of a guard.

Traditionally, the operational semantics of an xDSML can be defined by following Plotkin's Structural Operational Semantics (SOS) approach [32], which consists in defining the behavioral semantics of a language as a set of inference rules specifying the semantics of the concepts from the Abstract Syntax (AS) of the xDSML. Concurrency in SOS is encoded either in the rule structure representing the execution context (making it costly to execute as each step can modify the whole term that must then be wholly analyzed to decide the next step), in the internal states of the guards (for instance when a guard depends on data modified by another SOS rule, making it difficult to analyze as full arithmetic is too expressive) or implicitly inherited from the meta-language used to define the rewrite rules (thus making it implementation-dependent).

Currently, language workbenches [18, 24] used to specify and implement xDSMLs provide meta-languages with an implicit model of concurrency that is imposed to the xDSMLs. Therefore, the first step towards Concurrency-aware xDSMLs is to capture the MoCC using an appropriate meta-language [11]. The SOS is thus split in a concurrent part, the MoCC, and atomic SOS rules. The MoCC is used to define a partial ordering of the atomic SOS rules describing the semantics of the concepts in the AS of the xDSML. But the concurrency of an xDSML often depends on data from the domain, i.e. on the conclusions of previously-executed atomic SOS rules. For instance, a *Conditional Statement* is a language construct which consists in evaluating a condition and in executing one of the branches depending on the result of the evaluation of the condition. The MoCC is not able to define a partial ordering of the SOS rules that depends on such data. Therefore, a communication must be specified between the MoCC and the atomic SOS rules so that the right branch is executed. Our contribution consists in a dedicated meta-language to specify this communication and its integration in the GEMOC Studio[4] for validation purposes.

The rest of this paper is structured as follows: in Sect. 2 we present the xDSML that we will use to illustrate our approach. Section 3 presents background knowledge on the architecture of a Concurrency-aware xDSML and its application to the illustrative xDSML. In Sect. 4 we identify how to specify the communication between the MoCC and the atomic SOS rules. Section 5 details our implementation and its application on the example and discusses the advantages of our approach. Section 6 presents related work. Finally, Sect. 7 concludes and proposes perspectives for future work.

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

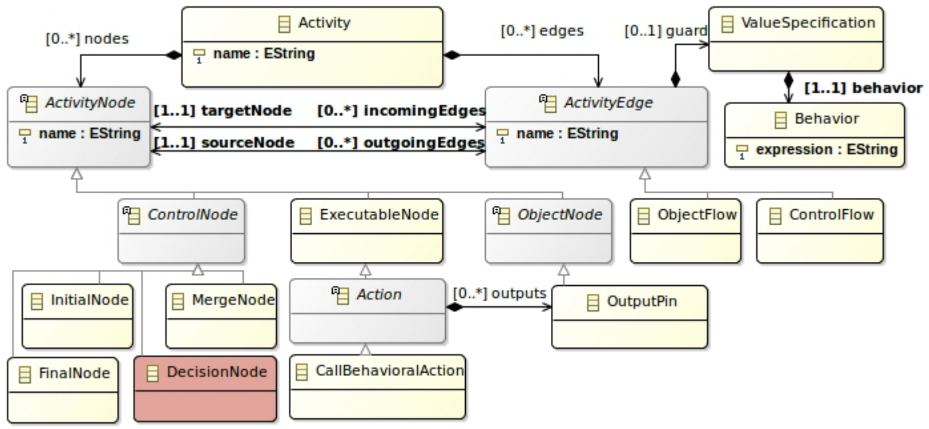


Fig. 1. Excerpt from the Abstract Syntax of fUML

2 Illustrative Example

The Foundational Subset for Executable UML Models (fUML) [29] is an executable subset of UML which specifies the behavioral semantics for Activity Diagrams. An excerpt of its AS is given in Fig. 1. For the sake of simplicity, we restrain ourselves to the classes present in this excerpt and we model a `Behavior` as a String. At runtime, fUML `ActivityNodes` can create, consume, transfer or duplicate tokens (depending on their role) which may be either control tokens (representing the passing of control along a `ControlFlow` edge) or object tokens (representing the passing of data along an `ObjectFlow` edge).

Our focus is on `DecisionNode` (in red on Fig 1): it is a kind of `Control Node` for which outgoing `ActivityEdges` must have a guard expression (`Value Specification`). The semantics of `DecisionNode` is given in English on page 371 of the UML Superstructure Specification[5]. Executing a `DecisionNode` consists in evaluating the guards of its outgoing edges in an unspecified order and propagating the execution on one of the edges whose guard is evaluated to true. Note that in fUML, a `DecisionNode` always represents an exclusive choice, so even if several outgoing edges are possible, only one will be executed.

We take the example of an activity that consists in determining what to drink at a coffee break. We want to drink coffee or tea if they are available; else, we want to drink water. Figure 2 gives a fUML model of this activity.

In this example, execution starts at the `InitialNode` on the left. Then the `ExecutableNode` "Check Table for Drinks" returns the list of available drinks from the table through an `OutputPin`. The `DecisionNode` then passes this list to the guard of each outgoing `ActivityEdge`. The guard evaluates to true if the list of drinks contains the specified drink, thus leading to one of the `Executable`

[5] `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`

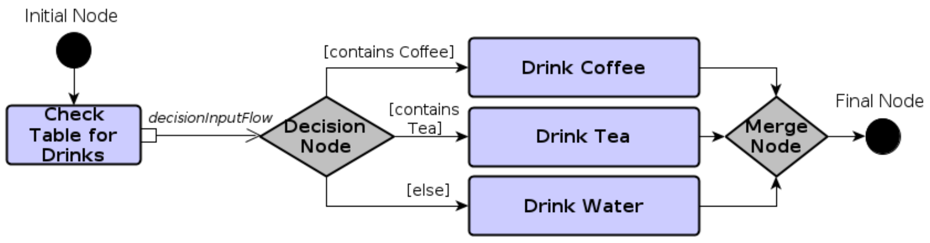


Fig. 2. fUML activity modelling the determination of what to drink based on what is available on the table

`Nodes` "Drink Coffee" if coffee is available, "Drink Tea" if tea is available, "Drink Water" otherwise. The `MergeNode` brings together the three exclusive branches, after which we have finished our drinking activity (hence the `FinalNode`).

3 Background on the Architecture of a Concurrency-aware xDSML and its Runtime

This paper extends previous work done in [8] by Combemale et al.. In this section, we present the elements from [8] upon which we will build our contribution in Sect. 4. We illustrate each concept on fUML.

3.1 Design of a Concurrency-aware xDSML

The architecture of a Concurrency-aware xDSML as illustrated by Fig. 3 comprises 5 Language Aspects. The *Abstract Syntax* captures the syntactic concepts of the domain and their relationships. An excerpt from the AS of fUML was given on Fig. 1. The semantic mapping between the AS and the Semantic Domain of the xDSML is implemented by the 4 other Language Aspects. First, the *Domain-Specific Actions* (DSA) represent the runtime state (*Execution Data*, ED) and the atomic operations (*Execution Functions*, EFs) of the domain. Some of the Execution Functions consist in modifying the ED of the model during the execution, e.g. executing an `ActivityNode` modifies the tokens held by its incoming and/or outgoing `ActivityEdges`. These EFs correspond to the *conclusions* of atomic SOS rules. Other EFs consist in retrieving information from the model, either about the model itself (e.g. the list of tokens held by an `ActivityEdge`), or computed based on data from the model (e.g. the boolean result of the evaluation of the guard of an `ActivityEdge`). These EFs correspond to the *premises* of the atomic SOS rules. In the context of this paper, we will use the data returned by these EFs to condition the rest of the execution, so we denominate them as the *Feedback Values*. The DSA of fUML are shown on Fig. 4 as a metamodel extending the AS: the classes `ControlToken` and `ObjectToken` extend the abstract class `Token`. `ObjectToken` holds some piece of data

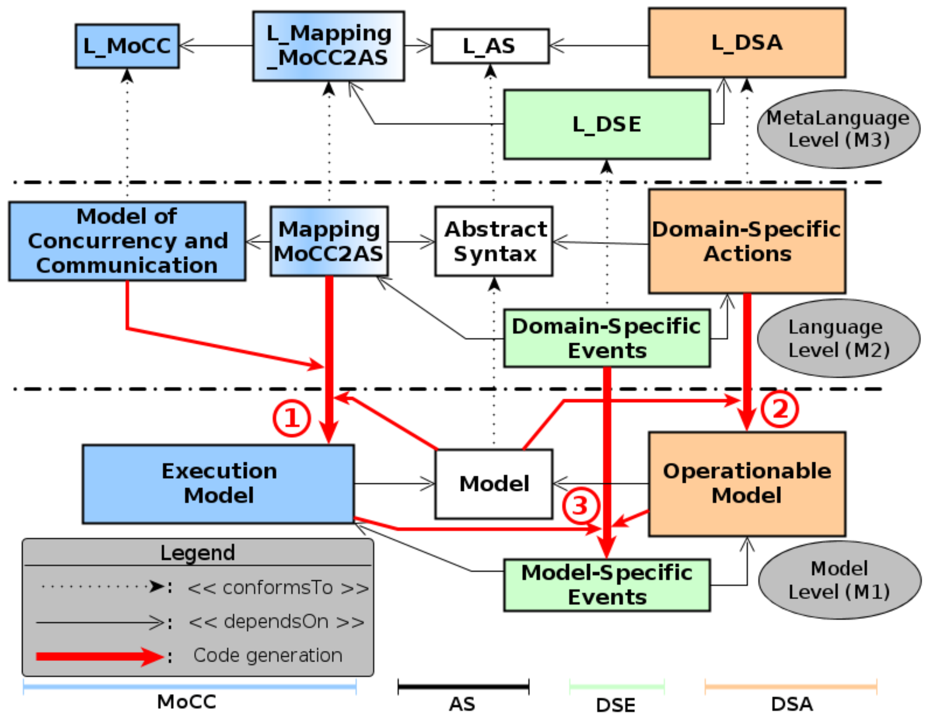


Fig. 3. Architecture of the Language Aspects constituting a Concurrency-Aware xDSML, and their counterparts at the Model Level (M1)

available through the `get()` operation. An `ActivityEdge` holds a list of Tokens and provides the `evaluate()` operation to evaluate its guard. An `ActivityNode` provides the `execute()` operation to implement its execution.

Second, the Model of Concurrency and Communication defines a partial ordering on *MoccEvents* using *MoccConstraints*. MoccEvents are mapped to AS concepts by the MoCC2AS Mapping, which makes the MoCC reusable for different languages. MoccEvents represent *when* something is scheduled to happen on

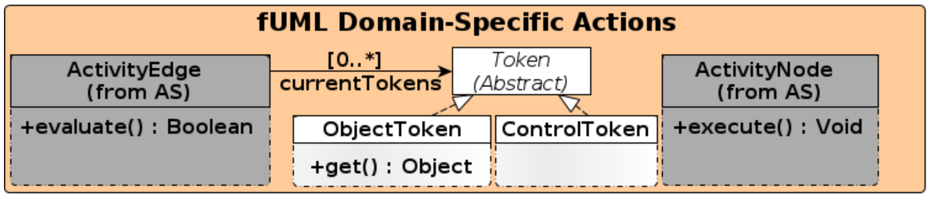


Fig. 4. The Domain-Specific Actions of fUML

the associated AS concept. MoccConstraints are scheduling constraints between these events, defining for instance whether or not two events may happen concurrently. The ordering between MoccEvents is partial, there can be an indefinite number of event occurrences between two other ones. For instance, a *Precedence* constraint between two MoccEvents A and B ensures that the n^{th} occurrence of A happens before the n^{th} occurrence of B, but it does not mean that every occurrence of A is followed immediately by an occurrence of B. For fUML, the MoCC contains 4 MoccEvents: `mocc_executeNode`, mapped to `Activity Node` of the AS to schedule when a node is to be executed; `mocc_evaluateGuard` mapped to `ActivityEdge` to schedule when the guard of an edge must be evaluated; `mocc_doExecuteTarget` and `mocc_doNotExecuteTarget` also mapped to `ActivityEdge` to represent whether or not the `target` of an `ActivityEdge` must be executed after having evaluated its guard. The MoccConstraints consist in the following. The execution of the `source` of any `ActivityEdge` must be done before the execution of its `target`. The evaluation of the guard of an `Activity Edge` must lead to a result which is exclusively `mocc_doExecuteTarget` or `mocc_doNotExecuteTarget`. Executing a `DecisionNode` triggers the concurrent evaluation of all the guards of its outgoing `ActivityEdges`. Only one of the outgoing branches of a `DecisionNode` can be taken therefore there is an exclusion between the `mocc_doExecuteTarget` of the outgoing `ActivityEdges`.

Finally, the Domain-Specific Events (DSE) bridge the gap between the DSA and the MoCC by specifying a 1-to-1 mapping between some of the MoccEvents and the Execution Functions. For fUML, the Domain-Specific Event `Execute ActivityNode` maps the MoccEvent `mocc_evaluateGuard` to the Execution Function `ActivityEdge.evaluate()` and `EvaluateGuard` maps `mocc_executeNode` to `ActivityNode.execute()`.

3.2 Runtime of a Concurrency-aware xDSML

The Language Aspects depend on the AS so their counterparts at the model level (M1) depend on the Model, as shown in the lower part of Fig. 3. First, the *Execution Model* (EM) is the result of the compilation of the MoCC and MoCC2AS using the model (**1** on Fig. 3). It defines MoccEvent instances in the context of the Model's elements (depending on the MoCC2AS Mapping). It also specifies a partial ordering between MoccEvent instances thanks to MoccConstraints. The EM is interpreted by the *Solver*. When asked, the Solver provides the set of the next possible Scheduling Solutions. A Scheduling Solution is composed of occurrences of MoccEvent instances. Second, the *Operationable Model* (OpM) is the result of the compilation of the DSA using the model (**2** on Fig. 3). It defines the data which represent the execution state of the model (Execution Data instances) and the atomic SOS operations available on this model (Execution Functions instances). The OpM is interpreted by the *DSA Executor*, which provides a mechanism to dynamically execute an Execution Function instance of the OpM. Finally, the *MSE Specification*, where MSE stands for *Model Specific Event* (instance of Domain-Specific Event), is the result of the compilation of the DSE with the EM and the OpM (**3** on Fig. 3). It specifies the mapping between

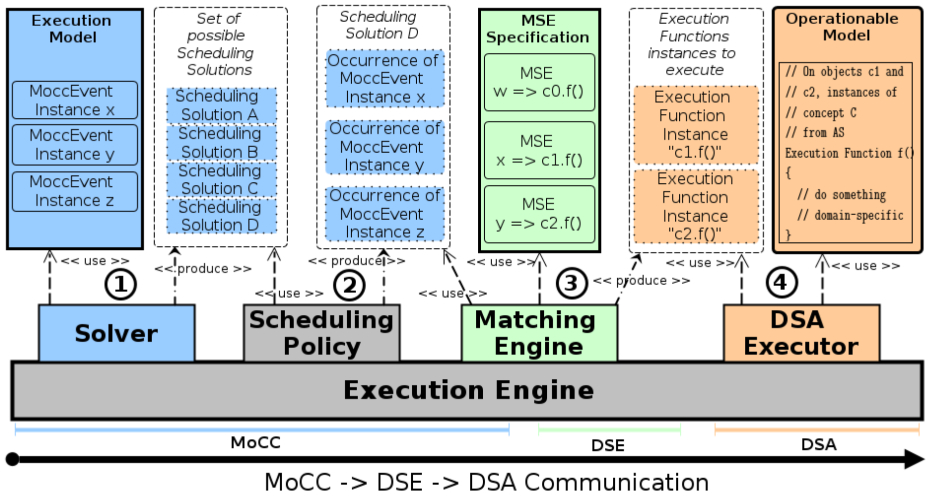


Fig. 5. The four steps to conduct one step of execution of a model conforming to a Concurrency-aware xDSML

the MoccEvent instances from the EM and the Execution Function instances of the OpM. Its interpreter is called the *Matching Engine*.

The *Solver*, *DSA Executor* and *Matching Engine* are coordinated together by the *Execution Engine*. The Execution Engine provides a number of pre-defined *Scheduling Policies* to select one Scheduling Solution among the set of possible ones returned by the Solver. In particular, one of them consists in asking the end-user to choose, allowing for step-by-step execution of the model. Figure 5 shows the 4 steps required to implement one execution step. **1**) the Solver produces a set of possible Scheduling Solutions (representing the allowed futures) based on the constraints in the Execution Model. **2**) the Scheduling Policy selects one Scheduling Solution among the possible ones. **3**) the Matching Engine matches the corresponding Model-Specific Event occurrences for this step using the MSE Specification and returns the Execution Function instances they are mapped to. Since not every MoccEvent is mapped to an Execution Function by a DSE, not every MoccEvent instance is mapped to an Execution Function instance. **4**) the DSA Executor executes the Execution Function instances in the OpM. This results in either some Execution Data being modified (for instance, executing the `InitialNode` creates a `ControlToken` on its outgoing `ActivityEdge`), or some Feedback Value being returned (when evaluating the guard of an `ActivityEdge`).

3.3 Shortcomings of the current approach

When a language construct like `DecisionNode` depends on a Feedback Value such as the result of the evaluation of the guards, there is a need for a DSA→MoCC communication which we call the *Feedback Mechanism*. In [8], the focus was placed on the MoCC→DSE and DSE→DSA mappings. The DSA→MoCC communication was identified but not specified or implemented. Currently, the return values of the executed EF instances are not used, and the Scheduling Policy used by the Execution Engine implements an arbitrary choice between the outgoing branches of the `DecisionNode`. We propose in this paper a dedicated meta-language to specify how to use the Feedback Value so that the Scheduling Policy can only select Scheduling Solutions compatible with the Feedback Value and its integration in the GEMOC Studio for validation purposes.

4 Contribution

Our contribution overview is given in Fig. 6. Its elements are labelled in red with bolder borders. The three steps to implement the Feedback Mechanism are visible on Fig. 6 and described below. **1**) the Feedback Value returned by an Execution Function instance is transmitted to the MSE Occurrence that originally triggered that Execution Function instance. **2**) the *Feedback Interpreter* interprets a new specification called the *Feedback Specification* using the Feedback Value. The Feedback Specification contains the Feedback Policy associated to every Model-Specific Event whose associated Execution Function instance returns a value. The Feedback Policy of an MSE specifies how to transform the Feedback Value returned by its Execution Function instance into a set of *Feedback Constraints*. **3**) the Feedback Constraints are used by a new entity called the *Scheduling Filter* placed between the Solver and the Scheduling Policy. The role of the Scheduling Filter is to remove, among the Scheduling Solutions returned by the Solver, the ones that are not compatible with the Feedback Value. For instance if the guard of an `ActivityEdge` evaluates to false, it will remove the Scheduling Solutions that have occurrences of `mocc_doExecuteTarget`, and therefore the target of that edge cannot be executed. In our approach, the Feedback Specification is specified at the language level, so Domain-Specific Events can have a Feedback Policy only if the Execution Function they trigger returns a value. In the rest of this section, we describe the Abstract Syntax and the semantics of the meta-language used to specify the Feedback Specification.

4.1 Abstract Syntax of the meta-language

The AS of the meta-language used to specify the Feedback Specification is given in Fig. 7. A *Feedback Policy* is composed of *Feedback Rules*. Each Rule is composed of a *Feedback Filter* and a *Feedback Consequence*. A Filter is a predicate on the return type of the Execution Function referenced by the associated Domain-Specific Event. A Consequence is a reference to a MoccEvent. Every Policy must also have a default Rule which has no Filter.

4.2 Semantics of the meta-language

When considering a Feedback Value, a Feedback Policy transmits it to all of its Feedback Rules (except for the default one), which transmit it to their respective Feedback Filter. Each filter is executed against the Feedback Value to determine if the associated Feedback Consequence must be applied. If none of the Feedback Rules is triggered by the Feedback Value, the default Feedback Rule of the Feedback Policy must be triggered. After this first phase, there is at least one Feedback Consequence to apply: either a list of Feedback Consequences whose associated Feedback Filter was validated by the Feedback Value, or the default one (the Feedback Value did not validate any of the Feedback Filters). For instance, evaluating the guard of an `ActivityEdge` returns a boolean value. If this value is 'true', then the consequence is that the corresponding MoccEvent `mocc_doExecuteTarget` may have an occurrence (and implicitly, that

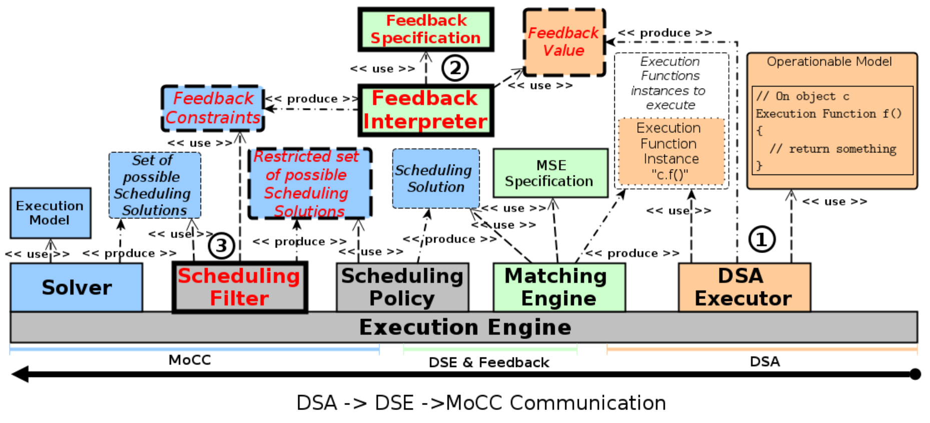


Fig. 6. Global view of the Feedback Mechanism (DSA→DSE→MoCC Communication)

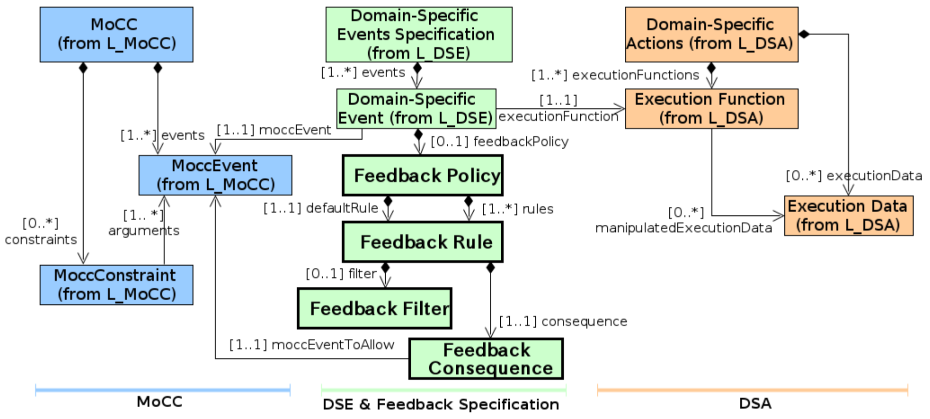


Fig. 7. Excerpt from the MetaModel of the meta-language to specify the Feedback Policies

`mocc_doNotExecuteTarget` may not, since they are in exclusion). This will result in an occurrence of the DSE `ExecuteActivityNode` for the target of this edge, which is what is expected when a guard returns true. If the Feedback Value is 'false', then the default rule is that `mocc_doNotExecuteTarget` may have an occurrence (and implicitly that `mocc_doExecuteTarget may not`) which means that the target of this edge will not be executed. The role of the default Feedback Rule is similar to that of the 'default' case or of the 'else' branch of conditional statements in General-purpose Programming Languages (GPLs). Having a default Feedback Rule is necessary because there may be an indefinite number of Scheduling Solutions between two occurrences of events, therefore relying on an event *not* occurring is not possible. The absence of consequence to a Feedback Value (for instance if all the guards return 'false', none of the branches is executed) must thus be modelled explicitly as a MoccEvent, which is what the Feedback Consequence of the default Feedback Rule represents.

Once the Feedback Consequences to apply have been determined, they are transformed into Feedback Constraints used as input by the Scheduling Filter. They specify which Scheduling Solutions must be removed from the set of possible ones provided by the Solver. The Filter keeps these constraints until one of the Solutions selected by the Scheduling Policy includes an occurrence of at least one of the Feedback Consequences to apply. For instance, the Filter removes all the Solutions with occurrences of `mocc_doNotExecuteTarget` for edges whose guard evaluated to true. Thus, either the next selected Solution has an occurrence of `mocc_doExecuteTarget` for one of these edges, and the Constraints can be removed (the data-dependent part of the conditional statement has been implemented); or not, in which case the Constraints remain in the Filter. This is due to the fact that, in our approach, there may be an indefinite number of steps between a cause (evaluating the guard of an `ActivityEdge`) and its consequence (`mocc_doExecuteTarget`). There may also be several Solutions even after the Filter has been applied. For instance, if both Tea and Coffee are available on the table, it is the Policy that implements the decision to choose between the two. Were `DecisionNode` not an exclusive selection, there would be a third option consisting in drinking both.

These semantics are based on a prerequisite: the MoccEvents referenced by the Feedback Consequences must not have occurrences outside of the context of the Feedback Mechanism. This is enforced by construction via an idiomatic construction in the MoCC Language. These MoccEvents only represent the decision resulting from a Feedback Value (executing one branch or another) so their occurrences cannot be as a consequence from another MoccEvent. As a consequence, these MoccEvents cannot be mapped to Domain-Specific Events. For instance, selecting one of the outgoing branches of a `DecisionNode` has nothing to do with the nodes on this branch. This is why the Domain-Specific Event `ExecuteActivityNode` is mapped to the MoccEvent `mocc_executeNode` and not to `mocc_doExecuteTarget`. Instead, there are causality constraints between `mocc_doExecuteTarget` and `mocc_executeNode` so that every occurrence of the first one causes an occurrence of the second one.

5 Implementation and Evaluation

Our contribution has been implemented in the GEMOC Studio, based on previous work done in [9, 8]. It is integrated in the Eclipse Modeling Framework (EMF) [15] to benefit from its large ecosystem.

5.1 Implementation

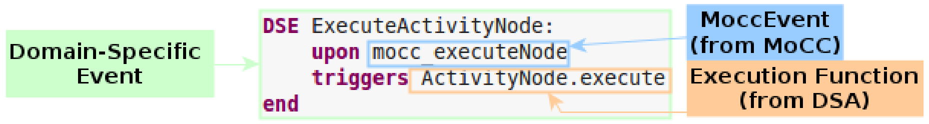
Abstract Syntax. We rely on Ecore, a meta-language at the heart of EMF [15] implementing EMOF [30], the OMG[6] standard meta-meta-model for Model-Driven Engineering, for specifying the Abstract Syntax of an xDSML. The associated Static Semantics are usually expressed in terms of OCL invariants [31]. Figure 1 shows the Ecore MetaModel of our fUML implementation.

Domain-Specific Actions. We rely on the Kermeta 3 Action Language (K3AL) [14], built on top of xTend [6] to implement the Execution Data as a set of classes, attributes and references aspectized onto a metaclass of the AS and the Execution Functions as operation implementations aspectized onto a metaclass of the AS. An excerpt of the Domain-Specific Actions for fUML was given on Fig. 4 as a MetaModel extending the AS of fUML (the implementation of the Execution Functions were not shown in this figure). K3AL, just like xTend, compiles into Java Bytecode and provides a DSA Executor which uses the Java Reflection API in order to dynamically execute Execution Functions.

Model of Concurrency and Communication and MoCC2AS Mapping. To specify the MoCC, we use MoCCML [11], a declarative meta-language designed to express constraints between events which can be capitalized into libraries agnostic of any AS. The mapping between the MoCC and the AS is defined with the Event Constraint Language (ECL) [12], an extension of OCL which allows the definition of events for concepts from the AS (MoccEvents), and the instantiation of MoCCML Constraints over these events (MoccConstraints). ECL is compiled to a *Clock Constraint Specification Language (CCSL)* [25] model interpreted by the TimeSquare [13] tool.

Domain-Specific Events and Feedback Specification. As the meta-language from Sect. 4 is at the same level as the DSE, the same language is used to specify the DSE and the Feedback Specification. Our implementation is called the *Gemoc Events Language* (GEL). It allows the declaration of Domain-Specific Events implementing the mapping between a MoccEvent defined in ECL and an Execution Function whose signature given in the AS of the xDSML is implemented in K3AL. Figure 8 gives an example of the GEL Concrete Syntax (defined using xText [6]) and the links between the Concrete and Abstract Syntaxes shown on Fig. 7. The Domain-Specific Event `ExecuteActivityNode` has occur-

[6] http://omg.org/



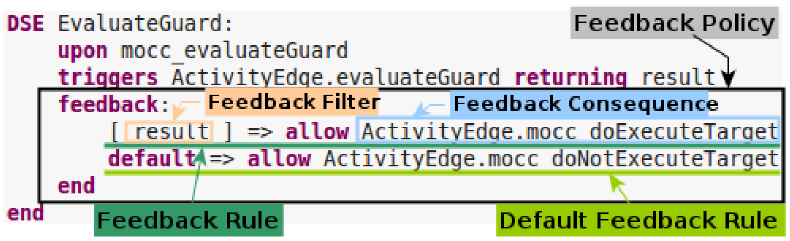
```
DSE ExecuteActivityNode:
    upon mocc_executeNode
    triggers ActivityNode.execute
end
```

Fig. 8. The ExecuteActivityNode Domain-Specific Event defined using GEL

rences whenever the MoccEvent `mocc_executeNode` has an occurrence in the selected Scheduling Solution, and triggers the Execution Function `execute()` from `ActivityNode` in the DSA. GEL also allows the specification of the Feedback Policy of a Domain-Specific Event if the associated Execution Function returns a result. Feedback Filters are specified using an expression language derived from OCL [31]. For instance, the Domain-Specific Event `EvaluateGuard` shown in Fig. 9 has a Feedback Policy. It has occurrences whenever `mocc_evaluateGuard` has an occurrence in the selected Scheduling Solution, and triggers the Execution Function `evaluateGuard()` of the class `ActivityEdge` when occurring. Its boolean result is then stored into the local variable `result` and depending on its value, either `mocc_doExecuteTarget` or `mocc_doNotExecuteTarget` is allowed to occur. Using a Model conforming to the AS of the xDSML, the EM and the OpM, GEL can be compiled into its Model Level counterpart, which is visible on Fig. 6 (the MSE Specification and the Feedback Specification).

5.2 Evaluation

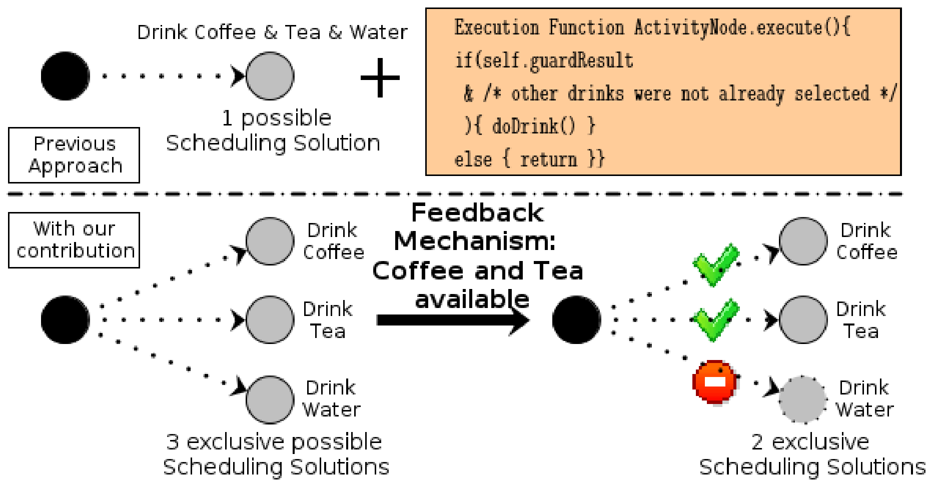
In our approach, the MoCC specifies that all the outgoing branches of a `Decision Node` *can* be executed after the `DecisionNode`, and that only one of them will actually happen (in exclusion with the other ones). Thanks to our contribution, the Scheduling Filter (using the Feedback Constraints) removes the possibility to execute any branch whose guard evaluated to false. This is illustrated in the lower half of Fig. 10. Later on, the Scheduling Policy is able to select which one of the legitimately possible branches is executed. Note that the relations



```
DSE EvaluateGuard:
    upon mocc_evaluateGuard
    triggers ActivityEdge.evaluateGuard returning result
    feedback:
        [ result ] => allow ActivityEdge.mocc_doExecuteTarget
        default => allow ActivityEdge.mocc_doNotExecuteTarget
    end
end
```

Fig. 9. The Domain-Specific Event EvaluateGuard and its Feedback Policy defined using GEL

between the different branches depend only on the MoCC (exclusion in the case of `DecisionNode`). We could implement conditional statements with more complex branching strategies. For instance, the difference between `Decision Node` and the *Switch Statement* of a GPL is that there is no order of evaluation of the guards, unlike in most GPLs where the first possible case is executed; and there is no possible fallthrough, unlike in many GPLs (C, Java, . . . as opposed to Pascal or Ruby). Such variations could be implemented by specifying the right constraints (for example, a precedence relation between the evaluations of the guards) between the MoccEvents. Without our contribution, the only way to implement conditional statements would be to specify in the MoCC that all outgoing branches must be executed, e.g. that the three `Executable Nodes` "Drink Coffee", "Drink Tea" and "Drink Water" must be executed. The exclusive selection would be implemented by storing the Feedback Value into the Execution Data and adding a check at the beginning of the Execution Function implementation to ensure that the guard has evaluated to true and that no other choice has been taken yet (should several guards evaluate to true, only one of the outgoing branches can be taken). This is illustrated in the upper-half of Fig. 10. In that case, the MoCC would not be an accurate model of the concurrency because the causality between the evaluation of a guard and its consequences are not modelled: all the possible consequences must be executed, and the analysis of the result of the guard is done in the Execution Function implementations. The exclusion between the outgoing branches of the `Decision Node` would also not be visible in the MoCC, but instead would be encoded in the EF implementations. Moreover, the MoCC would not be able to know which of the branches was really executed, so the next Scheduling Solutions it would


Drink Coffee & Tea & Water
1 possible Scheduling Solution
+
Previous Approach
With our contribution
Drink Coffee
Drink Tea
Drink Water
3 exclusive possible Scheduling Solutions
Feedback Mechanism: Coffee and Tea available
Drink Coffee
Drink Tea
Drink Water
2 exclusive Scheduling Solutions

```
Execution Function ActivityNode.execute(){
if(self.guardResult
 & /* other drinks were not already selected */
 ){ doDrink() }
else { return }}
```

Fig. 10. Implementing conditionals without (upper half), respectively with (lower half), our contribution

provide would consist in proceeding with the execution on all the branches, so all the other EFs would need to start by checking if they can really be executed. Ultimately, the MoCC would end up spread all over the EF implementations. This breaks the purpose of the MoCC as a first-order concept for the concurrency of an xDSML. Our approach maintains the strict separation of concerns between the MoCC and the DSA of an xDSML.

6 Related Work

Theoretical computer science has studied and formalized models of concurrency for a long time now, establishing well-known concepts such as Petri Nets [28, 22]; Process Algebras [21, 27, 26]; Agha's Actor Model [1] and its implementations in Erlang [4] or in Scala [19]. Some tools allow the design of multi-paradigm systems based on models of concurrency, such as Ptolemy [7, 17], ModHel'X [20], or Metropolis [5], but they lack the support for DSL design. DSL editors, usually called Language Workbenches, [18] such as Metacase's MetaEdit+ [36], JetBrain's MPS [37], Microsoft's DSL Tools [10], Spoofax [23], Rascal [34], Whole Platform [33], Itemis' xText/xBase [16] 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. Our approach aims at providing a Language Workbench with the tools to define the explicit model of concurrency used by the created DSL. Implementing conditional statements when the control flow is loosely coupled to the data has been a problem for other communities. Event-driven programming such as promoted by node.js [35] 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" [3], where the control flow is spread all over the code. This is one of the many reasons for the development of languages or libraries based on the Actor Model [1], such as Erlang [4] or Scala's Akka actors [19]. In our approach, since the model of concurrency is concretized in the MoCC, analyzing and reasoning about it is not an issue, at the cost of having to specify the communication between the MoCC and the DSA.

7 Conclusion and Perspectives

Most languages rely heavily on the use of data-dependent constructs such as conditionals or conditional-based constructs, therefore being able to implement them in Concurrency-aware xDSMLs is an important feature. Implementing these constructs was intricate previously [8] and led to an inaccurate Model of Concurrency and Communication and complex Domain-Specific Actions. Using our approach, it is possible to implement such language constructs while keeping the concurrency concern entirely in the MoCC and the Execution Functions and Execution Data simpler. In fUML, `DecisionNode` represents an unordered exclusive selection, but it is possible to implement more advanced data-dependent constructs such as a Switch Statement with fallthrough or a Switch Statement with parallel branches (equivalent to a Fork with guards on the outgoing branches). Our approach, validated in the GEMOC language workbench, can be implemented in other existing language workbenches, as long as they provide the adequate meta-languages for the Language Aspects presented in Sect. 3. In this paper we have considered the Execution Functions as atomic synchronous operations which modify the Execution Data or return data from the model. But we have identified the need for composite Execution Functions, for instance in order to be able to implement xDSMLs whose semantics are defined recursively; and the need for asynchronous Execution Functions, for example in the case where the expression of a Conditional Statement takes a long time to compute. These issues raise a lot of concerns with regards to the MoCC and the handling of data (arguments and exceptions) between the Execution Functions. Future works target solutions to these issues.

References

[1] Agha, G.A.: Actors: A model of concurrent computation in distributed systems. Tech. rep., DTIC Document (1985)

[2] Appel, A.W.: Compiling with continuations. Cambridge University Press (2007)

[3] Armstrong, J.: Red and green callbacks (2013), `http://joearms.github.io/2013/04/02/Red-and-Green-Callbacks.html`

[4] Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in ERLANG. Citeseer (1993)

[5] Balarin, F., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A., Sgroi, M., Watanabe, Y.: Modeling and designing heterogeneous systems. In: Concurrency and Hardware Design, pp. 228–273. Springer (2002)

[6] Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd (2013)

[7] Buck, J.T., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: A framework for simulating and prototyping heterogeneous systems (1994)

[8] Combemale, B., De Antoni, J., Vara Larsen, M., Mallet, F., Barais, O., Baudry, B., France, R.: Reifying Concurrency for Executable Metamodeling. In: Martin Erwig, R.F.P., van Wyk, E. (eds.) 6th International Conference on Software Language Engineering (SLE 2013). Lecture Notes in Computer Science, Springer-Verlag, Indianapolis, 'Etats-Unis (2013), `http://hal.inria.fr/hal-00850770`

[9] Combemale, B., Hardebolle, C., Jacquet, C., Boulanger, F., Baudry, B.: Bridging the Chasm between Executable Metamodeling and Models of Computation. In: SLE2012 - 5th International Conference on Software Language Engineering. LNCS, Springer (Sep 2012), `http://hal.inria.fr/hal-00725643`

[10] Cook, S., Jones, G., Kent, S., Wills, A.C.: Domain-specific development with visual studio dsl tools. Pearson Education (2007)

[11] De Antoni, J., Issa Diallo, P., Teodorov, C., Champeau, J., Combemale, B.: Towards a Meta-Language for the Concurrency Concern in DSLs. In: Design, Automation and Test in Europe Conference and Exhibition (DATE'15). Grenoble, France (Mar 2015), `https://hal.inria.fr/hal-01087442`

[12] De Antoni, J., Mallet, F.: Ecl: the event constraint language, an extension of ocl with events. Tech. rep., Inria (2012)

[13] De Antoni, J., Mallet, F.: Timesquare: Treat your models with logical time. In: Objects, Models, Components, Patterns, pp. 34–41. Springer (2012)

[14] DIVERSE-team: Github for k3al (2015), `http://github.com/diverse-project/k3/`

[15] Eclipse-Foundation: Eclipse modeling framework homepage (2015), `http://www.eclipse.org/modeling/emf/`

[16] Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for java. In: ACM SIGPLAN Notices. vol. 48, pp. 112–121. ACM (2012)

[17] Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity-the ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (2003)

[18] Fowler, M.: Language workbenches: The killer-app for domain specific languages (2005), `http://martinfowler.com/articles/languageWorkbench.html`

[19] Gupta, M.: Akka essentials. Packt Publishing Ltd (2012)

[20] Hardebolle, C., Boulanger, F.: Modhelx: A component-oriented approach to multi-formalism modeling. In: Models in Software Engineering, pp. 247–258. Springer (2008)

[21] Hoare, C.A.R., et al.: Communicating sequential processes, vol. 178. Prentice-hall Englewood Cliffs (1985)

[22] Jensen, K.: Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 1. Springer Science & Business Media (1997)

[23] Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and ides. In: ACM Sigplan Notices. vol. 45, pp. 444–463. ACM (2010)

[24] LanguageWorkbenchesChallenge: Language workbenches challenge comparing tools of the trade (2014), `http://www.languageworkbenches.net/`

[25] Mallet, F.: Clock constraint specification language: specifying clock constraints with uml/marte. Innovations in Systems and Software Engineering 4(3), 309–314 (2008), `http://dx.doi.org/10.1007/s11334-008-0055-2`

[26] Milner, R.: Communicating and mobile systems: the pi calculus. Cambridge university press (1999)

[27] Milner, R., Milner, R., Milner, R., Milner, R.: A calculus of communicating systems, vol. 92. springer-Verlag Berlin (1980)

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

[29] OMG: fuml specification (2013), `http://www.omg.org/spec/FUML/`

[30] OMG: Mof specification (2014), `http://www.omg.org/spec/MOF/`

[31] OMG: Ocl specification (2014), `http://www.omg.org/spec/OCL/`

[32] Plotkin, G.D.: A structural approach to operational semantics (1981)

[33] Solmi, R.: Whole platform (2005)

[34] van der Storm, T.: The Rascal language workbench. CWI. Software Engineering [SEN] (2011)

[35] Tilkov, S., Vinoski, S.: Node. js: Using javascript to build high-performance network programs. IEEE Internet Computing 14(6), 0080–83 (2010)

[36] Tolvanen, J.P., Kelly, S.: Metaedit+: Defining and using integrated domain-specific modeling languages. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. pp. 819–820. OOPSLA '09, ACM, New York, NY, USA (2009), `http://doi.acm.org/10.1145/1639950.1640031`

[37] Voelter, M., Pech, V.: Language modularity with the mps language workbench. In: Software Engineering (ICSE), 2012 34th International Conference on. pp. 1449–1450. IEEE (2012)