Grant ANR-12-INSE-0011

ANR INS GEMOC

D1.3.1 - xDSML/MoCC mapping language, tools and methodology (Report and Software)

Task 1.3.1

Version 2.0

DOCUMENT CONTROL

	–: 2014/05/22	A: 2015/05/26	B:	C:	D:
Written by Signature	Florent Latombe	Florent Latombe			
Approved by Signature					

Revision index	Modifications
–	version 0.1 — Initial version
A	version 1.0 — Added detailed contribution concerning the feedback problem
B	version 2.0 — Added description of the implementation of the meta-language realizing the DSA/MoCC mapping
C	
D	

Authors

Author	Partner	Role
Florent Latombe	IRIT - Université de Toulouse	Lead author
Joël Champeau	ENSTA Bretagne	Contributor
Benoît Combemale	INRIA	Contributor
Xavier Crégut	IRIT - Université de Toulouse	Contributor
Julien DeAntoni	I3S / INRIA AOSTE	Contributor
Marc Pantel	IRIT - Université de Toulouse	Contributor

Contents

1. Introduction

1.1 Purpose

Although this document is entitled ”xDSML and MoCC mapping”, we have since then determined that the MoCC was an entire part of an xDSML. Thus, we feel compelled to specify that this document is about the mapping between an xDSML’s Model of Concurrency and Communication (MoCC) and its Domain-Specific Actions (DSA). We motivate the need for such a mapping and illustrate the different natures of mappings we have identified. Then we describe our solution and its implementation – Gemoc Events Language (GEL) – in the GEMOC Studio.

The link to the sources of the accompanying software of this deliverable are given in Chapter 5.

1.2 Perimeter

In this document we only deal with the architecture of a single xDSML as proposed in Deliverable D1.1.1. More precisely, we only focus on how the mapping between a MoCC and DSA is done. Therefore, this document will not deal with: composition of xDSMLs, design of the Abstract Syntax of an xDSML, design of the Domain-Specific Actions of an xDSML, design of a Model of Concurrency and Communication.

1.3 Definitions, Acronyms and Abbreviations

- **AS**: Abstract Syntax.
- **API**: Application Programming Interface.
- **Behavioral Semantics**: see *Execution semantics*.
- **CCSL**: Clock-Constraint Specification Language.
- **CS**: Concrete Syntax.
- **Domain Engineer**: user of the Modeling Workbench.
- **DSA**: Domain-Specific Action.
- **DSE**: Domain-Specific Event.
- **DSL**: Domain-Specific Language.
- **DSML**: Domain-Specific (Modeling) Language.
- **Dynamic Semantics**: see *Execution semantics*.
- **Eclipse Plugin**: an Eclipse plugin is a Java project with associated metadata that can be bundled and deployed as a contribution to an Eclipse-based IDE.
- **ED**: Execution Data (part of DSA).
- **EF**: Execution Function (part of DSA).
- **Execution Semantics**: Defines when and how elements of a language will produce a model behavior.
- **GEL**: GEMOC Event Language which defines the two way protocol between the MoCC and DSA.

- **GEMOC Studio**: Eclipse-based studio integrating both a language workbench and the corresponding modeling workbenches.
- **GUI**: Graphical User Interface.
- **Language Workbench**: a language workbench offers the facilities for designing and implementing modeling languages.
- **Language Designer**: a language designer is the user of the language workbench.
- **MoCC**: Model of Concurrency and Communication.
- **MoCCML**: MoCC Modeling Language.
- **Model**: model which contributes to the convent of a View.
- **Modeling Workbench**: a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.
- **MSA**: Model-Specific Action.
- **MSE**: Model-Specific Event.
- **Operational semantics**: Constraints on a model that defines its step-by-step execution semantics (see **Execution Semantics**).
- **RTD**: RunTime Data.
- **Static semantics**: Constraints on a model that cannot be expressed in the metamodel. For example, static semantics can be expressed as OCL invariants.
- **TESL**: Tagged Events Specification Language.
- **xDSML**: Executable Domain-Specific Modeling Language.

1.4 Summary

Chapter 1 provides an introduction to this document. It defines the scope and the purpose of the document. Chapter 2 presents previous work related to reifying the concurrency of a language/program in various programming languages communities. Chapter 3 presents the objectives and underlying challenges of mapping the MoCC and the DSA of an xDSML. Chapter 4 describes our approach to realize the mapping between the MoCC and the DSA. Chapter 5 explains our implementation of the meta-language realizing the mapping between DSA and MoCC. Finally, Chapter 6 gives the conclusion of this document.

2. Background

In this chapter, we present the background concerning the mapping between the Model of Concurrency and Communication of an xDSML and its Domain-Specific Actions. We also show that problems similar to the ones we are facing in GEMOC can also be found in General-purpose Programming Languages' communities.

2.1 Separation of concerns in GEMOC

The separation of concerns for the structuration of the semantics of an xDSML in GEMOC originates from [2]. While Harel *et al.* [4] synthesize the construction of a language as the definition of a triple: **Abstract Syntax, Concrete Syntax and Semantic Domain**, Combemale *et al.* [2] focus on the definition of the Abstract Syntax (AS), the Semantic Domain (SD) and the respective mapping between them (M_{as_sd}). Several techniques can be used to define those three elements. In [2], the authors use executable metamodeling techniques, which allow one to associate operational semantics to a metamodel. In this context, they argue that the formal definition of the Semantic Domain must rely on two essential assets: the semantics of Domain-Specific Actions and the scheduling policy that orchestrates these actions. It is currently possible to capture the former in a metamodel with associated operational semantics and the latter in a *Model of Concurrency and Communication (MoCC)*, but the supporting tools and methods are such that it is very difficult to connect both to form a whole semantic domain (see right of Figure 2.1).

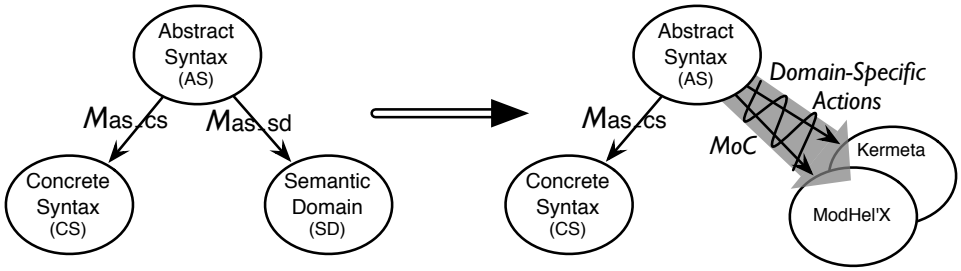

Figure 2.1: A modular approach for implementing the behavioral semantics of a DSL proposed in [2]

The authors propose to model Domain-Specific Actions (DSAs) and MoCCs in a modular and composable manner, resulting in a complete and executable definition of a DSL. The proof of concept relies on two state-of-the-art modeling frameworks developed in both communities: the Kermeta workbench that supports the investigation of innovative concepts for metamodeling, and the ModHel'X environment that supports the definition of MoCCs. Major benefits of this composition should include the ability to reuse a MoCC in different DSLs and the ability to reuse DSAs with different MoCCs to implement semantic variation points of a DSL. Saving the verification effort on MoCCs and Domain-Specific Actions also reduces the risk of errors when defining and validating new DSLs and their variants. This approach and the reuse capacities are illustrated through the actual composition of the standard fUML modeling language with a sequential and then a concurrent version of the discrete event MoCC.

This separation of concerns is at the heart of the design of xDSMLs in the GEMOC methodology. The design of the Domain-Specific Actions of an xDSML is treated more precisely in Deliverable D1.1.1, while the design of the Model of Concurrency and Communication of an xDSML in the GEMOC methodology can be found in Deliverable D2.1.1.

2.2 Other attempts at reifying the concurrency

Although the separation of concerns mentioned above is, in our case, done in a model-driven way and for DSMLs, there are several attempts at reifying the concurrency of programs in other programming languages' communities.

Joe Armstrong, creator of the Erlang[1] programming language, explains in a blog post[2] the difference between the callbacks in Erlang and the callbacks in Javascript. He even goes as far as saying that "every Javascript programmer who has a concurrent problem to solve must invent their own concurrency model. The problem is that they dont know that this is what they are doing.". We feel that in the approach taken in GEMOC for designing xDSMLs, being able to reuse Models of Concurrency is a key feature which will influence a lot how the mapping with our Domain-Specific Actions must be done.

In another community, the Scala community, Actors are the primary concurrency construct[3]. Actors provide an easy-to-use mechanism for developping concurrent applications in Scala, but sometimes the internal behavior of the Actors, described in Scala, can be "clumsy", as qualified in the description of a project accepted at Google's Summer of Code 2014[4] which proposes to specify the actors' internal behaviors using a Scala extension named SubScript instead.

Finally, in [3], the authors propose to capture the (implicit) protocol of a package into a set of rules. This idea is similar to what we want to do by using a MoCC to define the scheduling of DSA calls, and by defining constraints on the DSAs to ensure that they are not being called with a MoCC which would violate some properties (see Deliverable D1.1.1).

[1] http://www.erlang.org/
[2] http://joearms.github.io/2013/04/02/Red-and-Green-Callbacks.html
[3] http://www.scala-lang.org/old/node/242
[4] https://www.google-melange.com/gsoc/project/details/google/gsoc2014/anatoliykmetyuk/5668600916475904

3. Objectives and Underlying Challenges

In this chapter, after some reminders about the architecture of a concurrency-aware xDSML, we present the main objectives of the mapping between MoCC and DSAs of an xDSML and the associated challenges raised by these objectives.

3.1 Reminders

In [1] and Deliverable D1.1.1, the architecture of a Concurrency-Aware xDSML is defined as presented in Figure 3.1. Such an xDSML is defined by its Abstract Syntax (AS), its Domain-Specific Actions (DSA) (Execution Functions, Execution Data also called Execution State), its Model of Concurrency and Communication (MoCC) (events and constraints on these events) and its Domain-Specific Events (DSE). These elements are called the *Language Units* composing an xDSML.

Essentially, the AS specifies the concepts of the domain and their relations, the DSA specify the dynamic informations of the xDSML and how they evolve during the execution, and the MoCC specifies the concurrency aspects of the xDSML (synchronization, causalities, etc...). The MoCC is represented as a partial ordering on an Event Structure [5]. This allows us to abstract away operational operations and sequential control aspects into events (hereafter referenced to as the *MoccEvents*), which favors analyses like deadlock, determinism or freeness [6, Chapter 14]. As such, it defines the set of all possible executions of models conforming to an AS.

The AS, DSAs, DSEs and MoCC are defined at the language (Abstract Syntax) level. At the model level, the Execution Flow Model is composed of MoccEvent instances, Domain-Specific Event instances and Domain-Specific Action instances.

In [1] the execution of a model is realized by the collaboration of entities whose director is the *Execution Engine* (see Figure 3.2). In order to realize an execution step, the execution engine retrieves from the *Solver* of the MoCC which Event(s) can occur concurrently. Because the execution of a model is generally non-deterministic (usually due to concurrency), several sets of Events (*SchedulingSolutions*) can be proposed by the solver. The choice among them is made by an heuristic of the Execution Engine. Then, based on the mapping between the Events and the Execution Functions, the set of Execution Functions to execute is obtained. Finally, the Execution Engine uses the *Interpreter* to execute all these Execution Functions concurrently.

This approach works fine when the MoCC is independent from dynamic data of the model (such as the runtime state of the model). However, when implementing various xDSMLs, we have identified the need for a well-defined communication from the Execution Functions to the MoCC. This need is prevalent in conditionals and conditional-based language constructs where the evaluation of a condition is realized in an Execution Function. Its result must be used by the Execution Engine to choose among the possible Scheduling Solutions proposed by the Solver a solution that is coherent with the result of the guard.

3.2 Main Objectives

We have identified the need for mappings both from the MoCC to the DSA and from the DSA to the MoCC. Below we motivate each mapping individually.

3.2.1 Mapping from MoCC to DSA

The first objective of the mapping at hand is to map MoccEvents to the Execution Functions from the DSA. This allows the triggering of the operational atomic evolution operations (Execution Functions) according to the scheduling defined by an event structure (the MoCC). Challenges concerning this objective include:

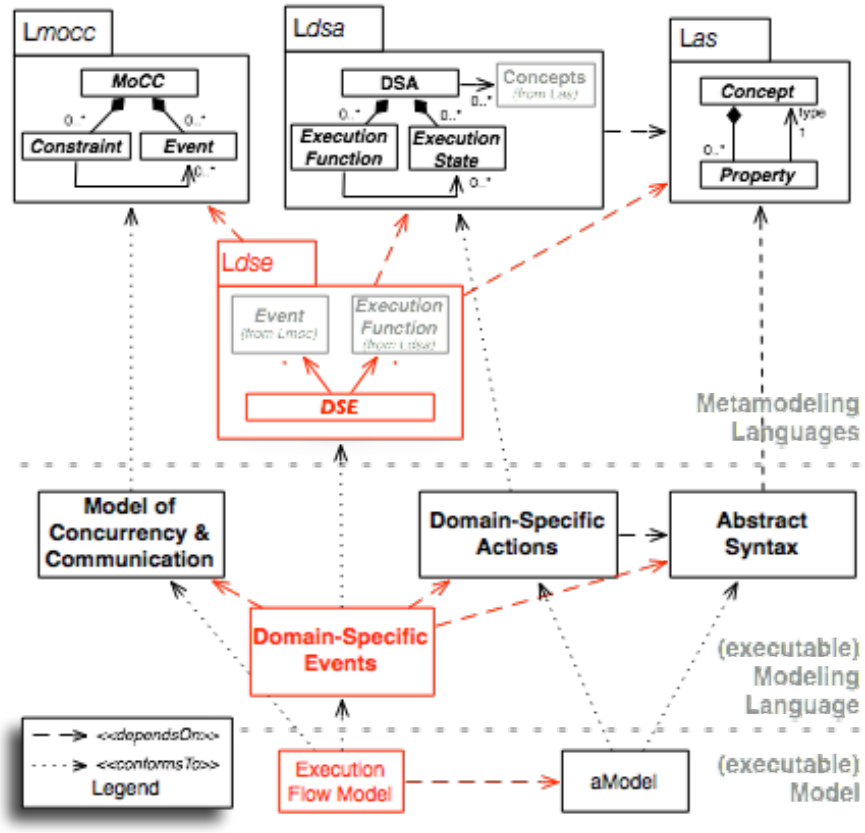

Figure 3.1: Modular Design of a Concurrency-Aware Executable Domain-Specific Modeling Language

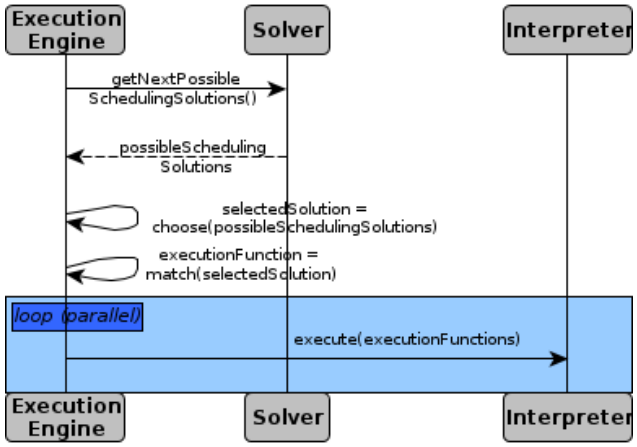

Figure 3.2: Sequence Diagram representing one step of execution of a model conforming to a Concurrency-aware xDSML.

- What is the nature of the mapping? 1-to-1, n-to-1, 1-to-n, n-to-m?

- How is this mapping specified?

3.2.2 Mapping from DSA to MoCC

The MoCC defines all the possible executions through the definition of a partial ordering on an event structure. But sometimes, the MoCC depends on data available only at runtime in the modeL. For instance, in fUML[1], `DecisionNode` is a kind of `ControlNode` whose outgoing `ActivityEdges` may have a guard (`ValueSpecification`). Only one of the outgoing branches can be executed in any case. The MoCC is able to specify that after executing a DecisionNode, either one of its outgoing branches will be executed, but it is not able to specify how the determination of which outgoing branches to execute is done. In that case, we need to be able to specify that if the guard of an edge returns true, then the branch may be executed, and if it returns false, then the branch may not be executed. Afterwards, an arbitrary choice can be made among the possible outgoing branches (in fUML there is no order between the branches unlike in Switch Statements in GPLs).

We call this communication mechanism the Feedback: first, the MoCC must be used to schedule the execution of an Execution Function that will return a value (in the case of fUML, a boolean value result of the guard). Then this data is interpreted to restrict in the MoCC the available choices. This entails the following challenges:

- What is the expressivity required for this mapping? What is its precise semantics?
- How can we specify this mapping?

3.3 Secondary Objectives

3.3.1 Reuse of Models of Concurrency and Communication

Reifying the concurrency of the semantics of an xDSML in a MoCC allows:

1. The reuse of existing Models of Concurrency with formally-proven properties for different xDSMLs
2. The variation of different MoCCs for a same domain in order to tackle the semantic variation points of the language.

However, in order to ensure the reachability of this objective, the Model of Concurrency and Communication of an xDSML should strive to be Domain-Agnostic. Since the MoCC is a part of the xDSML, and that the xDSML is by definition Domain-Specific, addressing this challenge imposes conditions on the architecture of the xDSML. One difficulty lies in the reusability of MoCCs for different domains without too much of a technical overhead.

Another consequence of this objective is that **the granularity of a MoCC determines its potential of reusability**. Indeed, the lower the granularity of a MoCC, the fewer xDSMLs can make use of this MoCC, thus limiting its reuse. On the contrary, the higher the granularity of a MoCC, the more xDSMLs can make use of this MoCC, for instance by aggregating elements of the MoCC, possibly even over time, in order to connect very finely-grained MoCCs with higher-level (more abstract) DSAs. This is very connected to the notion of interface and to the notion of API design: the more information is exposed by an API, the more it can be exploited by other programs or libraries. Similarly, the higher the granularity of a MoCC (by exposing in more details some of the causalities between events, for instance), the more it can be used for various applications, at the cost of complicating the mappings between MoCC and DSA.

3.3.2 Reuse of Domain-Specific Actions

Another objective of this mapping is the dual of the previous objective. This objective is the possibility to try out different versions of semantic variation points of a language, or in another words the reuse of Domain-Specific Actions with another MoCC. For example, in [1], a timed finite state machine is executed with either a ”rendez-vous” semantics or a ”send-receive” semantics.

[1] fUML is a foundational subset for executable UML models. See http://www.omg.org/spec/FUML/1.1/

This imposes that **the granularity of the Domain-Specific Actions of an xDSML determines the possibilities of semantic variation points**. Indeed, if the Execution Functions are very low-level (in the sense that they do one thing only, they are elementary), then we are able to aggregate them together in the mapping between MoCC and DSAs in order to create higher-level DSAs. Thus, depending on the possibilities of aggregation for the DSAs, the lower the level of abstraction of the DSAs is, the more semantic variation points there are (within the limit of the availability of MoCCs). On this matter, the notion of constraints placed on the DSAs of a language developed in Deliverable D1.1.1 intervenes and a notion of compatibility between DSAs and MoCCs can be explicited.

3.3.3 Behavioral Interface for heterogeneous models composition

The role of the Domain-Specific Events is twofold: to serve as interface between the MoCC and the DSAs of a language, and to serve as interface between an xDSML and its environment (user or other xDSML). The reason why these two interfaces are merged is because, first, we have identified that in many cases the intersection between these two interfaces is not empty (for example, the firing of a Finite State Machine `Transition` can be triggered due to a temporal guard or by a user through a GUI) ; second, the environment can also be seen as a MoCC: indeed, if the MoCC of the language leaves out choices for the environment (for example, which `Transition` to fire) but still manages some internal mechanics (evaluating the guards of `Transition`s), then the environment can be seen as ad-hoc constraints added on these choices. This is especially visible for models of reactive systems: if there is no environment, then the execution does not happen (waiting for some outsider's input) ; but if we consider the model and its environment as a model on its own, then its execution will happen in its entirety.

There must be a mechanism to specify whether or not a DSE is part of the behavioral interface of an xDSML.

This double nature must appear in the specification of the Domain-Specific Events. For example, DSEs used to bridge MoCC with DSAs may not be relevant for the behavioral interface of the xDSML, and thus should not appear to the environment. On the contrary, all the DSEs available in the behavioral interface should be triggerable by the internal mechanics of the xDSML. Concretely, a given DSE is either used purely for internal mechanics and thus does not pertain to the behavioral interface, or is relevant to the behavioral interface and as such should always be triggerable by the MoCC used by the xDSML: as we have explained earlier, in our approach the environment which would have triggered this DSE could be modelled as part of a MoCC used by the xDSML.

4. Description of Gemoc Events Language (GEL)

In this chapter, we describe our approach towards the mapping between MoCC and DSA. We denominate as *Communication Protocol* this mapping. The next chapter describes our implementation of this protocol in the GEMOC Studio.

4.1 Overview

The overview of our approach is given in Figure 4.1 as a class diagram. It is described in the following sections.

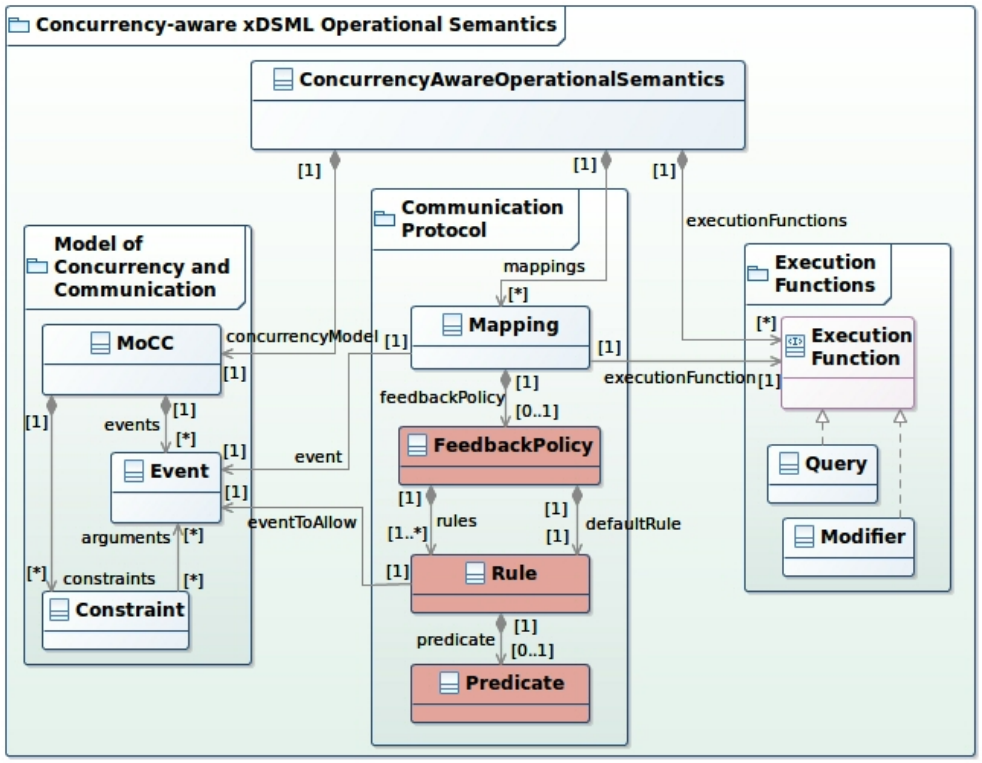

Figure 4.1: Class Diagram of our approach towards concurrency-aware operational semantics of an xDSML.

4.2 Mappings

A *Mapping* (Domain-Specific Event) maps an Event from the MoCC to an Execution Function. We have identified two natures of Execution Functions: *Modifiers* and *Queries*. When they are executed, Modifiers update the runtime state of the model. For instance, executing an InitialNode modifies the tokens held by its outgoing ActivityEdges. Queries do not modify the runtime state of the model, but instead provide runtime information, either about the model itself (e.g. the list of tokens held by an ActivityEdge during the execution), or computed based on data from the model (e.g. the boolean result of the evaluation of a guard). In the case of fUML, `ActivityNode.execute()` is a Modifier and `ActivityEdge.evaluateGuard()`

is a Query which returns a Boolean value. In the context of our contribution, since the MoCC depends on runtime information from the model, the Feedback Values will be provided by the Queries. Therefore the first step to realize the Feedback Protocol is the execution of a Query.

4.3 Feedback Protocol

4.3.1 Main Idea.

The main idea is to interpret the data returned by an Execution Function, hereafter called *Feedback Value*. In the case of fUML, the evaluation of a guard returns a boolean value. The interpretation of the Feedback Value must result in the selection of a Scheduling Solution coherent with the runtime state of the model. For instance if the guard of a branch evaluates to false, it may not be executed. But if it evaluates to true, then it may be executed.

Initially, the MoCC specifies that the guard of a branch must be evaluated, and that either the branch may be executed or it may not be. The role of the Feedback Protocol is to further restrain this partial ordering during runtime so that only the solution coherent with the feedback value remains possible. The Feedback Protocol cannot allow scenarios that were not originally allowed by the MoCC: it only realizes a further restriction of the partial ordering defined by the MoCC.

The Feedback Value is interpreted by a new entity of the runtime, the *Protocol Engine*, according to the Feedback Protocol specification, to create constraints. These constraints are then dynamically added to the MoCC, thus further restraining the partial ordering defined by the MoCC. Afterwards, these constraints are removed.

4.3.2 Concepts of the Feedback Protocol.

The concepts realizing the Feedback Protocol consist in the classes in red on Figure 4.1. A *Mapping* can have a *FeedbackPolicy* if the Execution Function it triggers is a Query. A Feedback Policy is made up of at least two *Rules*, including a default rule. A Rule is made of a *predicate* on the type returned by the Query, and of an Event from the MoCC; except for the default Rule which does not have a predicate.

At runtime, the Feedback Value returned by the Query is passed to all the non-default rules of the Feedback Policy. Either it validates the predicate of a rule, in which case this means that the rule must be applied ; or it does not, in which case it means that the rule must not be applied. If none of the rules are to be applied, then the default rule will be applied. Applying a rule consists in having the specified Event from the MoCC occur so as to model the decision taken based on the Feedback Value.

Applying the Rules whose predicate was validated by the Feedback Value consists in allowing the Events they reference to occur (hereafter *allowed Events*, in opposition with the *disallowed Events*). But this must be done with respect to the partial ordering defined by the MoCC. To do that, we consider the disallowed Events. We forbid these Events from occurring (by filtering out the solutions proposed by the Solver which contain occurrences of these Events). This is done until the allowed Events actually occur in the solution executed by the Execution Engine. This way, there are two possible scenarios when considering the solution chosen by the heuristic of the Execution Engine:

- Either the allowed Events are occurring, in which case forbidding the disallowed Events stops because we have reached our goal;
- Or the allowed Events are not occurring. Forbidding the disallowed Events continues in the next execution steps until the allowed Events are occurring.

The ordering of the Events remains partial: concurrent Events (such as the Event for the “Talk” action) may happen between the evaluation of a guard and its consequence. This is because by filtering out the solution containing occurrences of the disallowed Events, we only remove the solutions not in coherence with the runtime state of the model. Thus, interleaving the “talking” action, initially allowed by the MoCC, is still possible since it is not incoherent with regards to the runtime state of the model.

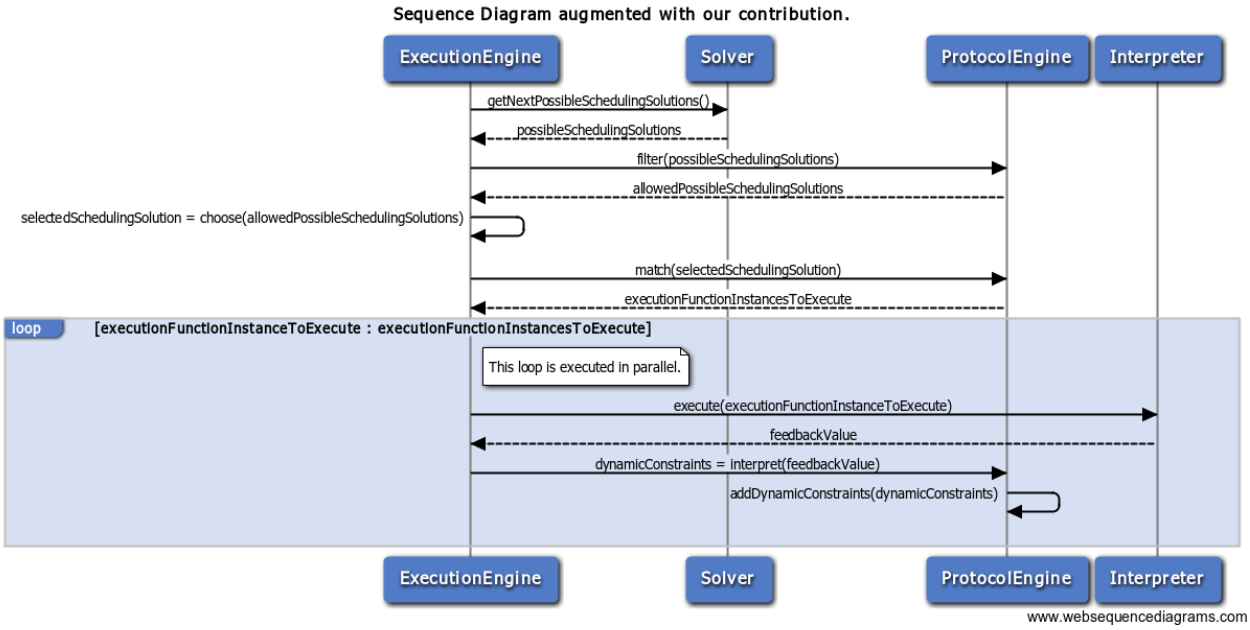

Figure 4.2: Updated Sequence Diagram representing one step of execution with the Feedback Mechanism included.

The default rule of a Feedback Policy acts as a mandatory "else" branch in the conditional statement of a General-purpose Programming Language (GPL) or as the "default" case in the 'switch-case' constructs available in many GPLs. Having a default rule per Feedback Policy is necessary because there may be an indefinite number of Events between two occurrences of an Event, 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 may be executed) must thus be modeled explicitly as an Event in the MoCC, which is what the consequence of the default rule represents. Otherwise, it is not possible to make the difference between the situation where all the guards returned `false` and none of the branch can be executed, and the situation where some branches will ultimately be executed but some concurrent nodes have been interleaved between the evaluation of the guards and the execution of the branches.

4.3.3 Changes in the runtime.

Figure 4.2 shows the changes of the runtime as a modification of the sequence diagram shown in Section 3.

The *Protocol Engine* is in charge of realizing the communication in both directions: forward, when there are occurrences of Events from the MoCC which are mapped to Execution Functions, in which case the Execution Functions must be executed; and backward, when a Query has been executed and there is an associated Feedback Policy. By adding dynamic constraints to the MoCC through the Feedback Protocol, a Feedback Value has an influence on future execution steps, as its name suggests.

The Protocol Engine is used three times by the Execution Engine (cf. Figure 4.2).

1. First, to take into account feedback from previously-executed steps: after retrieving from the Solver which Events may happen concurrently, the Protocol Engine removes from the set of possible solutions the ones not conforming to the dynamic constraints. The heuristic of the Execution Engine then chooses one of the remaining solutions. If the solution chosen has occurrences of the allowed Events then the corresponding dynamic constraints are removed.

2. Second, the Protocol Engine realizes the forward communication by matching the Events in the selected solution to the Execution Functions to execute.

3. Third, when a Query is executed and it has an associated Feedback Policy, the Protocol Engine interprets the Feedback Value it returns. This results in adding the dynamic constraints which will be applied in future execution steps to restrain the set of possible Scheduling Solutions (cf. step 1).

5. Implementation of GEL

The definition given in the previous chapter has been implemented in the GEMOC Studio. Our implementation of GEL allows the definition of Domain-Specific Events, realizing the mapping between a MoccEvent and an Execution Function. It also allows the definition of a Feedback Policy when the Execution Function is a Query (returns a result) which realizes the communication from an Execution Function to the MoCC in order to solve the feedback problem described in Section 3.

All the sources are available on the Git repository of the GEMOC project (branch `feature/gel`, folder `org/gemoc/GEL`) and are integrated in the GEMOC Studio starting from version 1.1.

5.1 Syntax

Domain-Specific Events and Feedback Policies depend on both the MoCC and the Execution Functions. Moreover a Feedback Policy is defined in the context of a Domain-Specific Event, therefore we decided to use the same meta-language (GEL) to specify both. It allows the declaration of Domain-Specific Events realizing the mapping between an Event of the MoCC defined in ECL and an Execution Function whose signature given in the AS of the xDSML is implemented using the Kermeta 3 Action Language (K3AL).

An excerpt from the Abstract Syntax of GEL is shown on Figure 5.1.

The GEL Concrete Syntax, defined using xText, is illustrated on Figure 5.2 using fUML as an example xDSML. The import mechanism is not illustrated on this figure but is used in order to import both the MoCC and the Execution Functions (through the `import` keyword preceding a URI). The mapping to the Abstract Syntax of the example shown on Figure 5.2 is as follows:

- Lines 1 to 4, between keywords `DSE` and `end`, is the definition of a Domain-Specific Event with a name;
- Line 2, after the keyword `upon` is a reference to an Event from the MoCC;
- Line 3, after the keyword `triggers` is a reference to an Execution Function of the domain.

When the heuristic of the Execution Engine selects a solution with an occurrence of the Event `mocc_executeNode`, it creates an occurrence of the DSE `ExecuteActivityNode`, which results in the execution of the Execution Function `ActivityNode.execute()`.

GEL allows the specification of a Feedback Policy for a Domain-Specific Event. Its associated Execution Function should however be a Query. In this context, the value returned by the Query is called the *Feedback Value*. Figure 5.3 shows the Feedback Policy for the evaluation of a guard in fUML (either the guard returns true and the branch may be executed or it returns false and it may not):

- Lines 1 to 8, definition of the `EvaluateGuard` Domain-Specific Event and its Feedback Policy;
- Line 2, reference to an Event from the MoCC
- Line 3, reference to an Execution Function, and after the keyword `returning` is the definition of the local variable (here, `result`) into which its result will be stored
- Line 4 to 7, between keywords `feedback` and `end`, is the definition of the Feedback Policy
- Line 5, between square brackets is the predicate of the rule (in this example the Execution Function returns a Boolean value). Predicates are specified using an expression language derived from OCL. After the keyword `=> allow` is the consequence of the rule (`mocc_mayExecuteTarget` is allowed)
- Line 6, is the default rule of the policy: it has no predicate (replaced by the keyword `default`) and allows the Event `mocc_mayNotExecuteTarget` to occur

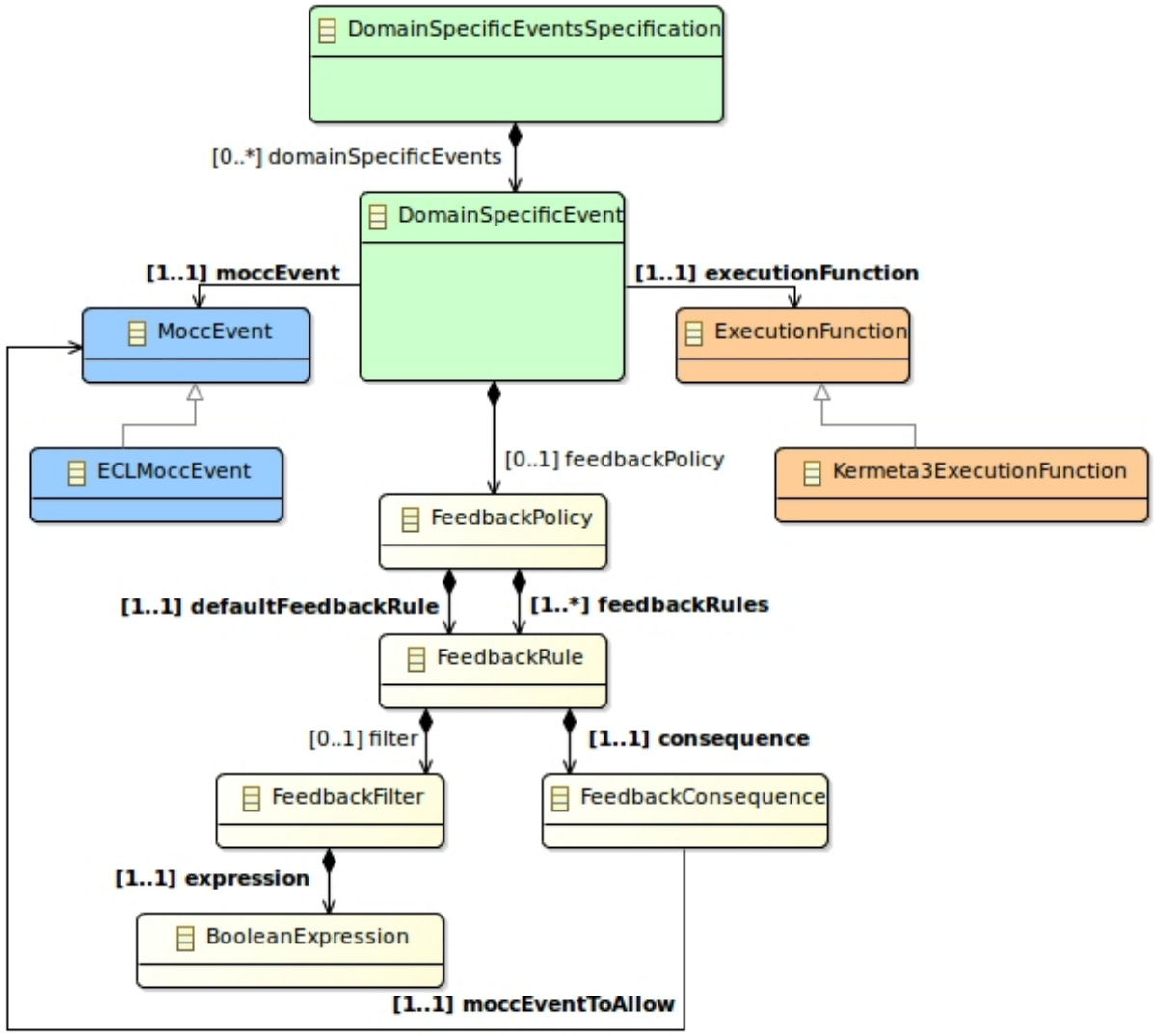

Figure 5.1: Excerpt from the Abstract Syntax of GEL

```
1 DSE ExecuteActivityNode:
2     upon mocc_executeNode
3     triggers ActivityNode.execute
4 end
```

Figure 5.2: The `ExecuteActivityNode` Domain-Specific Event defined using 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 5.3: The Domain-Specific Event `EvaluateGuard` and its Feedback Policy defined using GEL

When the Domain-Specific Event `ExecuteActivityNode` occurs, its result is stored in the local variable `result` and passed to the only non-default rule. If the predicate is validated (the result of the Execution Function was true) then the `mocc_mayExecuteTarget` will be allowed to occur (as explained below). If not, its default rule is applied (`mocc_mayNotExecuteTarget` is allowed to occur).

5.2 Semantics

Applying the FeedbackRules whose predicate was validated by the Feedback Value consists in allowing the MoccEvents they reference to occur (hereafter *allowed MoccEvents*, in opposition with the *disallowed MoccEvents*). But this must be done with respect to the partial ordering defined by the MoCC. To do that, we consider the disallowed MoccEvents. We forbid these MoccEvents from occurring (by filtering out the solutions proposed by the Solver which contain occurrences of these MoccEvents). This is done until the allowed MoccEvents actually occur in the solution executed by the Execution Engine. This way, there are two possible scenarios when considering the solution chosen by the heuristic of the Execution Engine:

- Either the allowed MoccEvents are occurring, in which case forbidding the disallowed MoccEvents stops because we have reached our goal;
- Or the allowed MoccEvents are not occurring. Forbidding the disallowed MoccEvents continues in the next execution steps until the allowed MoccEvents are occurring.

The ordering of the MoccEvents remains partial: concurrent MoccEvents may happen between the evaluation of a guard and its consequence. This is because by filtering out the solution containing occurrences of the disallowed MoccEvents, we only remove the solutions not in coherence with the runtime state of the model. Thus, interleaving the execution of two nodes, initially allowed by the MoCC, is still possible since it is not incoherent with regards to the runtime state of the model.

Another way to realize this would have consisted in filtering out the solutions not containing occurrences of the allowed MoccEvents: this way, the only solutions among which the heuristic makes its choice have occurrences of the allowed MoccEvents. But this would have had the following drawbacks:

- It would have created a locally total ordering of the MoccEvents, which goes against the MoCC being a partial ordering of its MoccEvents. This would have removed interleaving possibilities initially allowed by the MoCC. For instance in fUML, the execution of a node in concurrence with a DecisionNode means that the node's execution can be interleaved between the evaluation of a guard and the occurrence of its consequence MoccEvents. This would not have been possible anymore with the approach described above: the Scheduling Solution containing solely an occurrence of the MoccEvents corresponding to the execution of the concurrent node would have been filtered out for not containing an occurrence of the allowed MoccEvents;
- It would have not taken into account possible other constraints on the MoccEvents. For instance in our fUML example, suppose we had added a constraint to ensure that the concurrent node is executed strictly before any of the outgoing branches of a DecisionNode is executed. Suppose we had just evaluated the guards and had not executed the concurrent node yet. To satisfy this new constraint,

the Solver would have proposed only one solution, which would have been to execute the concurrent node. This would have created a deadlock since this solution would have been filtered out for not containing occurrences of the allowed MoccEvents.

In our solution, this deadlock issue can happen only if the allowed MoccEvents conflict with the constraints of the MoCC; but this would be a design problem of either the MoCC or the Feedback Policy. The Feedback Policy is only a means to remove possible solutions proposed by the Solver; it cannot allow new solutions that were not originally allowed by the MoCC. Therefore if the allowed MoccEvents were originally not possible according to the MoCC, a deadlock will appear at runtime.

The default rule of a Feedback Policy acts as a mandatory "else" branch in the conditional statement of a General-purpose Programming Language (GPL) or as the "default" case in the 'switch-case' constructs available in many GPLs. Having a default rule per Feedback Policy is necessary because there may be an indefinite number of MoccEvents between two occurrences of a MoccEvent, therefore relying on a MoccEvent *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 may be executed) must thus be modeled explicitly as a MoccEvents in the MoCC, which is what the consequence of the default rule represents. Otherwise, it is not possible to make the difference between the situation where all the guards returned `false` and none of the branch can be executed, and the situation where some branches will ultimately be executed but some concurrent nodes have been interleaved between the evaluation of the guards and the execution of the branches.

5.3 Runtime

The runtime of GEL is written mostly in Java as a set of Eclipse plug-ins integrated in the GEMOC Execution Engine (see Deliverable D4.2.1.).

6. Conclusion

The separation of concerns in GEMOC between the MoCC of a language and its DSA raises many challenges in the realization of the mapping between both concerns, especially while keeping in mind the reuse of existing language units such as MoCC and DSAs. Based on a preliminary approach integrating existing tools of all the partners of the GEMOC project, we have identified many of these challenges. We have described the meta-language GEL and its implementation in the GEMOC Studio that allows the specification of the communication between MoCC and DSA, both in the “forward” direction (from MoCC to DSA, an occurrence of a MoccEvent triggers the execution of an Execution Function) and in the “backward” direction (from DSA to MoCC when the latter depends on dynamic data available at runtime in the model). GEL implements the core functionalities of the mapping between MoCC and DSA, and we may implement additional features such as parameters for Domain-Specific Events. In that case, this document will be updated to reflect the additions realized.

7. References

[1] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Fr'ed'eric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In Richard F. Paige Martin Erwig and Eric van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science, Indianapolis, 'Etats-Unis, 2013. Springer-Verlag.

[2] Benoit Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoit Baudry. Bridging the Chasm between Executable Metamodeling and Models of Computation. In *5th International Conference on Software Language Engineering (SLE 2012)*, volume 7745 of *Lecture Notes in Computer Science*, pages 184–203, Dresden, Allemagne, February 2013. Springer-Verlag.

[3] Shahram Esmaeilsabzali, Rupak Majumdar, Thomas Wies, and Damien Zufferey. Dynamic package interfaces - extended version. *CoRR*, abs/1311.4934, 2013.

[4] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[5] Glynn Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, LNCS. Springer, 1987.

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

