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 0.1

DOCUMENT CONTROL

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

Revision index	Modifications
–	version 0.1 – Initial version
A	
B	
C	
D	

Authors

Author	Partner	Role
Florent Latombe	IRIT - Université de Toulouse	Lead author
Xavier Crégut	IRIT - Université de Toulouse	Contributor
Marc Pantel	IRIT - Université de Toulouse	Contributor
Benoît Combemale	INRIA	Contributor
Julien DeAntoni	I3S / INRIA AOSTE	Contributor
Joël Champeau	ENSTA Bretagne	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 (DSAs). In particular, it will explain the motivations for such a mapping and the first (and current) approach used for this mapping in the context of GEMOC. Then, we will expose the limitations of the aforementioned approach and propose an improved approach that we call Gemoc Events Language (GEL). The link to the sources of the accompanying software of this deliverable are given in section 4.

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 Model of Concurrency and Communication and Domain-Specific Actions 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. The purpose of the mapping at hand is to improve the reusability of existing MoCCs and of existing DSAs ; but this raises a lot of challenges in the expressivity that this mapping must allow.

1.3 Definitions, Acronyms and Abbreviations

- **AS**: Abstract Syntax.
- **API**: Application Programming Interface.
- **Behavioral Semantics:** see *Execution semantics*.
- **CCSL**: Clock-Constraint Specification Language.
- **Domain Engineer**: user of the Modeling Workbench.
- **DSA**: Domain-Specific Action.
- **DSE**: Domain-Specific Event.
- **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.
- **Execution Semantics:** Defines when and how elements of a language will produce a model behavior.
- **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.
- **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.
- **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 DSAs of an xDSML. Chapter 4 show the current approach used for mapping the MoCC of an xDSML with its DSAs and the main weaknesses and limiting points of this approach. Chapter 5 presents a proposal for a new tool and methodology to realize the mapping. Chapter 6 presents an example of xDSML developed using the new language proposed. Finally, Chapter 7 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.* [5] 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).

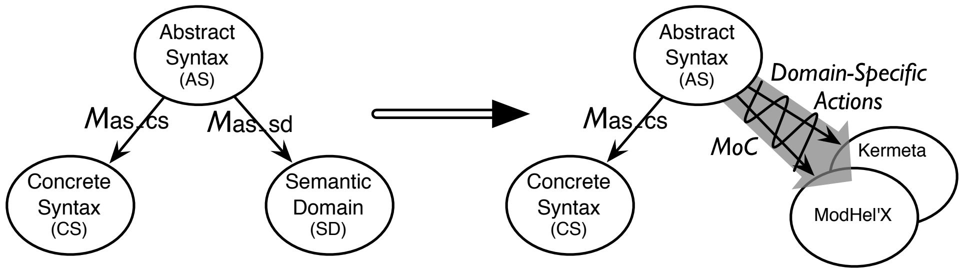

Figure 2.1: A modular approach for implementing the behavioral semantics of a DSL proposed in [2]

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 [4], 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 (DSAs) (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 (DSEs). These elements are called the *Language Units* composed an xDSML. Essentially, the AS specifies the concepts of the domain and their relations, the DSAs 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...).

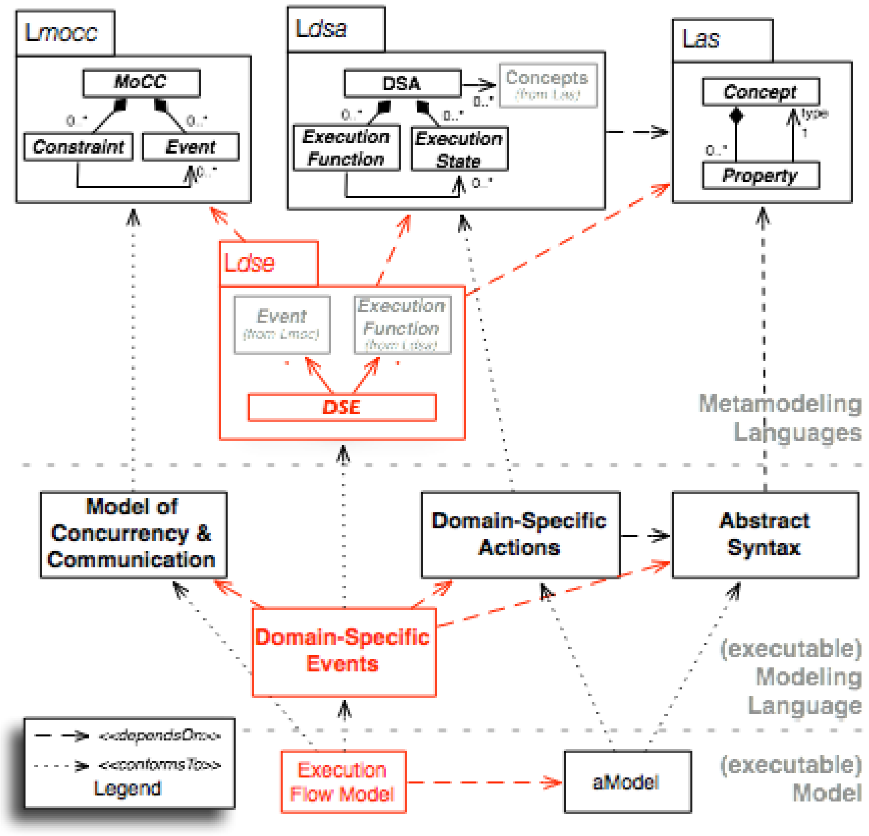

Figure 3.1: Modular Design of a Concurrency-Aware Executable Domain-Specific Modeling Language

The DSEs specify the mapping between the MoCC and the DSAs of an xDSML.

First, a DSE is mapped to one or several event(s) from the MoCC. Since the name ”event” is too generic and also appears in the term Domain-Specific Event, we propose a new denomination for the events used by the MoCC: MoccEvents.

Definition: A MoccEvent is an event constrained in the MoCC of an xDSML by relations.

Second, we have identified three different natures of Execution Functions (see Deliverable D3.3.1):

- *action*: changes the Execution Data.
- *query*: returns a representation of the Execution Data called RunTime Data (RTD).
- *helper*: a utility function useful to create more modular execution functions.

A Domain-Specific Event must be mapped to one or several Execution Function(s) of type *action*, also denominated as DSAs.

Definition: A Domain-Specific Event is a mapping between one or several MoccEvents and one or several Domain-Specific Actions.

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

Definition: A Model-Specific Action (MSA) is an instance of a Domain-Specific Action, in the sense that it is specific to a model conform to an xDSML.

Definition: A Model-Specific Event (MSE) is an instance of Domain-Specific Event, in the sense that it is specific to a model conform to an xDSML. An MSE is a mapping between one or several MoccEvent instances and one or several MSAs.

3.2 Reuse of existing Models of Concurrency

One of the main objectives of the mapping between the Model of Concurrency and Communication and the Domain-Specific Actions is to allow the reuse of existing MoCCs.

Indeed, two of the biggest benefits we can retrieve from reifying the concurrency of the semantics of an xDSML are:

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 this objective is reachable, the following challenge must be addressed:

Challenge: The Model of Concurrency and Communication of an xDSML must 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 moc, 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.

A schematic example of MoCC reuse is given in Figure 3.2. In this figure, the MoCC can be connected to ”Domain 1” because the levels of granularity fit ; however it requires a layer of adaptation to be connected to ”Domain 2”.

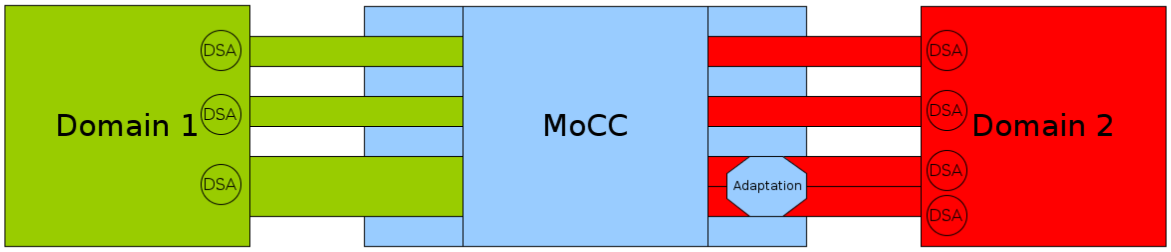

Figure 3.2: Reuse of a MoCC for two domains with different granularity of DSAs

3.3 Reuse of existing 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 anotehr MoCC. For example, in [1], a timed finite state machine is executed with either a ”rendez-vous” semantics or a ”send-receive” semantics.

Thus, the following objective must be addressed:

Challenge: The Domain-Specific Actions of an xDSML must be aggregatable.

This requirement imposes that **the granularity of the Domain-Specific Actions of an xDSML determines the possibilities of semantic variation points**. Indeed, if the Domain-Specific Actions’ 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 asbtract 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. Further versions of this document should add up on that point.

Figure 3.3 shows a schematic example of how the reuse of DSAs would work. DSAs are aggregated in order to cope with the granularity level of the different MoCCs used.

3.4 Behavioral interface for heterogeneous models composition

In previous sections 3.2 and 3.3, we have expressed the need to be able to *aggregate* elements, either from the MoCC or from the DSAs, in order to cope with the different levels of granularity. As explained in section 3.1, since the DSEs are the mapping between the MoCC and the DSAs of an xDSML, DSEs must also be the point where the aggregation of elements from the MoCC and from the DSAs should be specified.

However, 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

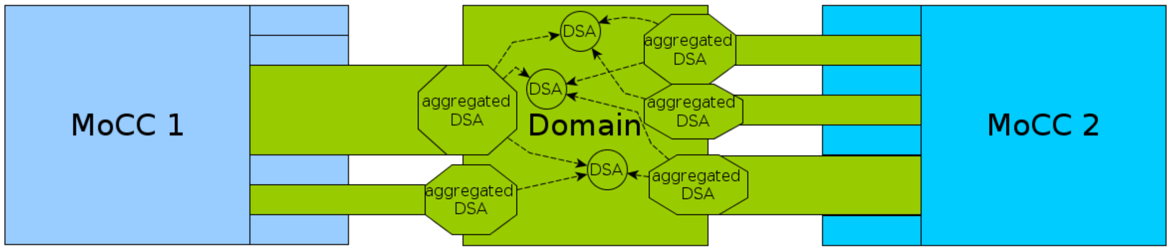

Figure 3.3: Reuse of a domain's DSAs with different MoCCs

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.

Challenge: 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.

3.5 Feedback from DSAs to MoCC

Sometimes the control flow is parameterized by data from the model which is known only at runtime (or at design-time but available through DSAs). For example that is the case for fUML[1]'s `DecisionNode` where the decision usually depends on dynamic values resulting from the previous steps of execution (e.g. test of whether a value is within a certain range, etc...). In that case, the mechanism used to transmit the needed data from the model at runtime back to the Model of Concurrency and Communication is what we call *Feedback*. Unfortunately, this mechanism is unavoidable, unless we are capable and willing to capture most of the domain at hand into the MoCC language, which obviously defeats most of the purposes of the GEMOC approach.

Challenge: There must be a mechanism to transmit data coming from the DSAs to the MoCC at runtime.

There exists several types of feedback. For now, we have only identified the need for one type of feedback (MoccEvent-targeting Feedback explained below), but we also feel that using the other types of feedback

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

could be beneficial for reusability purposes. Future versions of this document should motivate either the non-use of these other types of feedback, or motivate their existence through an example. The different types of feedback are defined below, from the lowest-level to the highest-level of abstraction.

3.5.1 MoccEvent-targeting Feedback

The MoccEvent-targeting Feedback is the feedback which aims at either forcing or forbidding an occurrence of a specific MoccEvent instance. For example, in an fUML model, the feedback associated to the DSA `evaluate()` of each outgoing branch of a `DecisionNode` should provide to the MoCC the information of whether or not each branch is possible. This boolean information is encoded using two MoccEvents (for example *evaluate_returned_true* and *evaluate_returned_false*) and these two MoccEvents are used later on to constrain the firing of the activities present on the different outgoing branches of the `DecisionNode`. In that case, the feedback specification would express that if `evaluate()` returns true, then the MoccEvent *evaluate_returned_true* associated to the instance of `DecisionNode` considered must occur at the next step of execution, and similarly for a returned value of false.

3.5.2 DSE-targeting Feedback

In the MoccEvent-targeting Feedback, we have an indirect influence on the future occurrences of Domain-Specific Event instances by indicating to the MoCC interpreter that some MoccEvent instance should (or should not) occur. With the same example as the paragraph above, the MoCC specifies a causality constraint between a `DecisionNode`'s outgoing branch's MoccEvent *evaluate_returned_true* instance occurring and the same outgoing branch being executed. Therefore, a higher-level way to do this causality would be to directly specify in the feedback for the DSA `evaluate()` that if evaluated to true, then an occurrence of the DSE Fire on the possible branch should happen ; the feedback interpreter being in charge of finding the link between the specified DSE instance and the MoccEvent(s) instance(s) to force or forbid.

3.5.3 DSA-targeting Feedback

In a similar spirit, if it is possible to do DSE-targeting Feedback, then it is also possible to do DSA-targeting Feedback by looking at the DSEs' referenced DSAs. Once again, the feedback interpreter would be in charge of finding the MoccEvent(s) instance(s) to force or forbid.

Although we have only identified the need for MoccEvent-targeting Feedback, it makes sense to have these higher-level feedbacks too, especially considering MoccEvent-targeting Feedback are very intensely linked to the internal workings of the MoCC used, thus making the change of MoCC for a set of DSAs mentioned in 3.3 harder.

Another aspect of this feedback which has not yet been considered thoroughly is the range the feedback has. That is, when exactly is a feedback specification applied? For now, we have limited ourselves to applying the feedback specification as soon as possible, which means that the forcing or forbidding is applied to the very next step of execution. But we suspect that having the possibility to specify when the feedback specification should be applied could be useful. Further versions of this document should motivate this issue in more details.

The Feedback idea presented in this section raises many issues on the analysability of MoCCs. Indeed, with this mechanism, some of the information parameterizing the causalities are not explicitly present in the MoCC specification. Still, it could be possible to expose such information from the DSAs and the Feedback specification through an interface like contracts. We have not yet identified exactly which information will be required and how to communicate it for analysability. Further versions of this document should express this need in more details.

4. Current Approach

In this chapter, we present the current state of the implementation of the mapping between MoCC and DSAs using the EMF-based technologies Ecore, Kermeta 3, CCSL, ECL. This is the implementation that was used for the example in [1]. ECL is available as an Eclipse plugin in the GEMOC Studio[1].

4.1 Domain-Specific Actions

In our approach, the Abstract Syntax of the xDSML is enhanced with two elements:

- Execution Data encode the internal state of the executed model (in an Object-Oriented approach, this would be the private attributes of a class)
- Execution Functions (see the taxonomy proposed in Deliverable D3.3.1)

Deliverables D1.1.1 and D3.3.1 give more details about the implementation of the elements composing the DSAs.

The ED and Execution Functions are all implemented using Kermeta 3.

4.1.1 Execution Data

Since Execution Data are only used as an internal mechanics, they can be implemented as attributes with limited visibility. An example is given in listing 4.1. In this Petri net example, the number of tokens in a `Place` is encoded as an integer attribute called *currentMarking*.

Listing 4.1: Execution Data for Petri net

```
@Aspect(className=typeof(Place))
class PlaceAspect extends NodeAspect {
  private int currentMarking;
}
```

4.1.2 Domain-Specific Actions

DSAs are implemented as methods. Restrictions on how to use and implement these methods are discussed in more details in Deliverables D1.1.1 and D3.3.1. An example is given in listing 4.2. In this Petri net example, the initialization of the root class `PetriNet` consists in setting the marking of each `Place` to its initial value (statically available in the Abstract Syntax of the xDSML). There is also a DSA query which returns a RunTime Data, the value of the marking for each `Place`. This data can then be mapped for animation purposes.

Listing 4.2: Example of implementation of the Domain-Specific Action initialize() for Petri net

```
@Aspect(className=typeof(Place))
class PlaceAspect extends NodeAspect {
  def public int getCurrentMarking(){
    return _self.currentMarking
  }
}
```

[1] http://gemoc.org/studio-download/

```
@Aspect(className=typeof(PetriNet))
class PetriNetAspect extends NamedElementAspect {
  def public void initialize() {
    var allPlaces = _self.elements.filter(Place)
    allPlaces.forEach(place|place.currentMarking = place.initialMarking)
  }
}
```

4.2 Domain-agnostic library used by the Model of Concurrency and Communication

In CCSL, one of the core elements is the Clock. Clocks can be either explicit (declared) or implicit (resulting from an Expression). Relations are constraints placed on clocks (either explicit or implicit) and Expressions allow us to create implicit clocks (which are constrained too). A kernel of useful Relations and Expressions is provided with the language. See Deliverables D2.1.1 and D2.2.1 for more details about the MoCC language. CCSL also offers the possibility to capitalize user-defined Relations and Expressions into libraries. We use this feature to capitalize in reusable modules the domain-agnostic relations and expressions used by the MoCCs.

A simple example is given in listing 4.3. This example defines a relation named ”FirstAndOnlyOnce” which takes two parameters (two clocks *mocc_firstEvent* and *mocc_otherEvents*). By using the Relations `Precedes(LeftClock, RightClock)` and `Coincides(Clock1, Clock2)` of the kernel of CCSL and the Expression `OneTickAndNoMore(OneTickAndNoMoreClock)`, our relation ”FirstAndOnlyOnce” has the following semantics: the clock *mocc_firstEvent* ticks once and only once, and this tick happens before any tick of *mocc_otherEvents*. By using this Relation with the correct arguments in the example of Petri nets, where *mocc_firstEvent* will be the initialization of a `PetriNet` and *mocc_otherEvents* will be an Expression expressing the Union of all the other possible MoccEvents, we are able to ensure that the initialization of our `PetriNet` is done once and before anything else happen.

Listing 4.3: Example of a CCSL Library defining one domain-agnostic relation named ”FirstAndOnlyOnce”

```
Library My_mocc_library{
  imports{
    import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
       ccsllibrary/kernel.ccslLib" as kernel;
    import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
       ccsllibrary/CCSL.ccslLib" as CCSLLib;
  }

  RelationLibrary My_relations{
    /**
     * One tick of mocc_firstEvent once and before any tick of
        mocc_otherEvents.
     * mocc_otherEvents should be an expression gathering all the other events
     * (or at least all the events that would be scheduled first otherwise).
     */
    RelationDeclaration FirstAndOnlyOnce(
      mocc_firstEvent : clock,
      mocc_otherEvents : clock
    )
    RelationDefinition FirstAndOnlyOnce[FirstAndOnlyOnce]{
      Expression firstTickOfFirstEvent =
        OneTickAndNoMore(OneTickAndNoMoreClock -> mocc_firstEvent)
      Expression firstTickOfOtherEvents =
        OneTickAndNoMore(OneTickAndNoMoreClock -> mocc_otherEvents)
      Relation Precedes(
        LeftClock -> mocc_firstEvent,
        RightClock -> firstTickOfOtherEvents
      )
      Relation Coincides(
        Clock1 -> mocc_firstEvent,
        Clock2 -> firstTickOfFirstEvent
      )
    }
  }
}
```

4.3 Domain-Specific Events specification

The specification of the Domain-Specific Events is done using ECL [3]. Below are described the three main aspects of this specification.

4.3.1 Domain-Specific Event declaration

In ECL, a DSE is declared in the context of a class from the Abstract Syntax of the xDSML. An example is given in Listing 4.4. An event named *mocc_initialize* is declared on the metaclass `PetriNet`.

Listing 4.4: Example of a DSE declaration

```
import 'platform:/plugin/org.gemoc.sample.petrinet.model/model/Petrinet.ecore'

ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
    ccsllibrary/kernel.ccslLib"
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
    ccsllibrary/CCSL.ccslLib"
ECLimport "platform:/resource/org.gemoc.sample.petrinet.moc/ccsl/petrinet.
    ccslLib"

package petrinet

  context PetriNet
    def: mocc_initialize : Event

endpackage
```

4.3.2 Mapping DSE → DSA

Declaring a DSE within a context is not enough ; the DSEs are scheduled by the MoCC and when they occur they trigger their associated DSA(s). Therefore we need to map our DSE to a set of DSAs. In ECL we can only map one DSE to a single DSA. An example is given in Listing 4.5. The DSE *mocc_initialize* is mapped to the DSA *initialize()* of `PetriNet`.

Listing 4.5: Example of a DSE declaration with link to a DSA

```
import 'platform:/plugin/org.gemoc.sample.petrinet.model/model/Petrinet.ecore'
```

```
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
   ccsllibrary/kernel.ccslLib"
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
   ccsllibrary/CCSL.ccslLib"
ECLimport "platform:/resource/org.gemoc.sample.petrinet.moc/ccsl/petrinet.
   ccslLib"

package petrinet

  context PetriNet
    def: mocc_initialize : Event = self.initialize()

endpackage
```

4.3.3 MoccEvents in ECL

It is not mandatory to link an ECL Event to a DSA. In that case, we consider these events as the MoccEvents mentioned in section 3. That is, they are not part of the behavioral interface of the xDSML and do not serve the internal mapping between DSAs and MoCC explicitly. Instead, they can be used for the Feedback specification as in 3.5.1 or in the instanciation of the MoCC as described later in 4.4. So, by the interpretation that we make of ECL, listing 4.4 defines a MoccEvent while listing 4.5 defines a DSE.

4.3.4 Mapping MoCC $\rightarrow$ DSE

In ECL, the mapping between DSEs and MoCC is actually implicit: each ECL Event gives birth to one CCSL clock during the compilation phase[2] and when this specific clock ticks, it is interpreted as being a DSE and its associated DSA is triggered. Thus, we cannot explicitly distinguish a DSE from the MoccEvent it is mapped to.

4.4 Instanciation of the Model of Concurrency and Communication

The domain-agnostic Relations and Expressions defined in section 4.2 need to be instanciated with the MoccEvents and DSEs declared in ECL. Listing 4.6 gives an example of MoCC instanciation: the invariant ”OneInitComesFirst” instanciates the Relation *FirstAndOnlyOnce* of the CCSL Library defined in 4.2. The arguments given to this relation are:

1. The DSE *mocc_initialize* of PetriNet which initializes the Petri net,
2. A CCSL Expression of all the DSEs *mocc_isFireable*, each mapped to the DSA which evaluates the fireability of a `Transition`.

This way, we are defining a scheduling where the initialization of the Petri net will come before we try to evaluate the fireability of any `Transition`.

Listing 4.6: Example of a MoCC instanciation

```
import 'platform:/plugin/org.gemoc.sample.petrinet.model/model/Petrinet.ecore'

ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
   ccsllibrary/kernel.ccslLib"
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
   ccsllibrary/CCSL.ccslLib"
```

[2] ECL Events being specified at the language-level, there is a generation phase based on the model to execute

```
ECLimport "platform:/resource/org.gemoc.sample.petrinet.moc/ccsl/petrinet.
    ccslLib"

package petrinet

  context PetriNet
    def: mocc_initialize : Event = self.initialize()

  context Transition
    def: mocc_isFireable : Event = self.isFireable()

  context PetriNet
    inv OneInitComesFirst:
      let allOtherEvents: Event = Expression Union(self.elements->select(
          element|
        ((element).oclIsKindOf(Transition))).oclAsType(Transition)->first().
            mocc_isFireable)
      in Relation FirstAndOnlyOnce(self.mocc_initialize, allOtherEvents)

endpackage
```

We can then generate the Model of Concurrency and Communication specific to our model. A simple example is given in listing 4.7, where the Petri net is composed of only one `Transition` named*MyTransition* and the PetriNet (root element) is called *BasicExample*. The two DSEs *mocc_initialize* and *mocc_isFireable* have given birth to one CCSL Clock each (since there is one element in the model conforming to the context of *mocc_initialize*, that is the `PetriNet` element ; and one element in the model conforming to the context of *mocc_isFireable*, that is the `Transition` *MyTransition*). Also, the invariant (MoCC instanciation) has given birth to one CCSL Relation *FirstAndOnlyOnce*.

This CCSL specification can then be interpreted by the MoCC solver and its result is interpreted by the GEMOC Execution Engine.

Listing 4.7: Instanciation of the MoCC for a simple Petri Net

```
ClockConstraintSystem BasicExample {
  imports {
    import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
        ccsllibrary/kernel.ccslLib" as kernel ;
    import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
        ccsllibrary/CCSL.ccslLib" as CCSL ;
    import "platform:/plugin/org.gemoc.sample.petrinet.moc/ccsl/petrinet.
        ccslLib" as petrinet ;
    import "platform:/resource/org.gemoc.sample.petrinet.instances/test_1/
        BasicExample.xmi" as PetriNet ;
  }
  entryBlock mainBlock
  Block mainBlock {
    Relation BasicExampleFirstAndOnlyOnce_OneInitComesFirst
          [ FirstAndOnlyOnce ] (
            mocc_firstEvent -> BasicExample_mocc_initialize,
            mocc_otherEvents -> MyTransition_mocc_isFireable
        )
    Clock BasicExample_mocc_initialize : clock
          -> evt_BasicExample_mocc_initialize ("BasicExample->initialize")
    Clock MyTransition_mocc_isFireable : clock
          -> evt_MyTransition_mocc_isFireable ("MyTransition->isFireable")
  }
}
```

4.5 Limitations of the current approach

Below are explained the main conflicting points in using CCSL and ECL as described in the approach above. An improved proposition is given in Chapter 5.

4.5.1 Definition of a MoCC

A definition of MoCC used in GEMOC is given in Section 2 of Deliverable D2.1.1. However, for the sake of simplicity, MoCC can be abstracted a set of constraints over events. The term ”event” being too generic, we call these events ”MoccEvents”.

In the current implementation described above, it is unclear what the MoCC exactly is. First, from our xDSML design, the MoCC being at the language level, it cannot be the generated extendedCCSL file. So the MoCC lies somewhere between the MoCC Library, the DSE specification and the MoCC instanciation. The MoCC Library would be a good candidate as it is reusable and domain-agnostic ; but then the problem is that using the MoCC Library one way or another can give very different semantics: this defeats the purpose of reusing MoCCs since we would need to look how the MoCC is instanciated. Concretely, two xDSMLs using the same MoCC could have very different behaviours. Should the MoCC instanciation be considered as part of the MoCC? The MoCC instanciation heavily relies on the AS of the domain at hand, therefore is Domain-Specific, so it cannot be fully part of the MoCC.

This problem shows that we need to separate, in the MoCC instanciation, the domain-specific parts and the domain-agnostic parts. For example, in the MoCC instanciation shown previously, we can identify the domain-specific parts. These domain-specific parts are highlighted in red in listing 4.8. The non-highlighted parts of the listing should be capitalized next to the MoCC Library to constitute a fully reusable MoCC.

Listing 4.8: Identifying the domain-specific elements

```
context PetriNet
  inv OneInitComesFirst:
    let allOtherEvents: Event = Expression Union(self.elements->select(
        element|
    ((element).oclIsKindOf(Transition))).oclAsType(Transition)->first().
        mocc_isFireable)
    in Relation FirstAndOnlyOnce(self.mocc_initialize, allOtherEvents)
```

4.5.2 Dependency of ECL towards CCSL

ECL has been developed independently of the GEMOC project and therefore, it can be seen as an extension of CCSL which allows us to write at the Abstract Syntax level instead of at the model level. The model-level extendedCCSL specification is generated based on the model and the ECL file.

However, this approach limits us to CCSL and its solver as MoCC in our language architecture. Coupling the GEMOC Execution Engine to ECL would disallow upcoming extensions designed for other languages who could play CCSL's role. It would be more practical to have a clearer separation between the implementations.

4.5.3 Dependency of ECL towards OCL

ECL is built as an extension to OCL [3]. OCL invariants are used to specify how to instanciate the CCSL relations from the MoCC Library based on the Abstract Syntax of the domain at hand. OCL invariants are written with a context (a class from the AS). However, some languages could use Domain-Specific Events defined within the context of several classes from the AS. A toy example of a Family DSML is given in Figure 4.1. A `Family` is composed of several `FamilyMember`s, and the only DSE is *Talk* which requires two `FamilyMember`s.

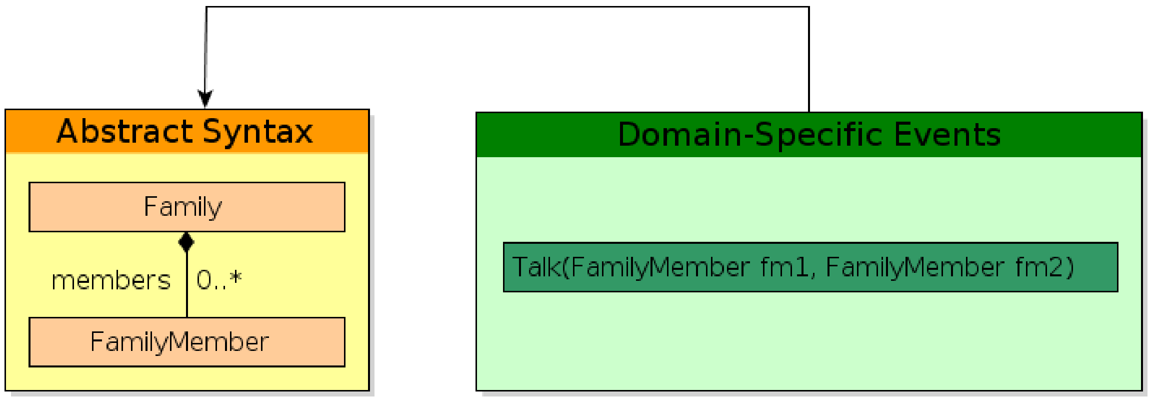

Figure 4.1: AS and DSE specification of a simple Family DSML.

Instanciated to a simple family composed of Alice, Bob and Carol (Figure 4.2), one would expect to have the following possible events:

- Talk(Alice, Alice)
- Talk(Alice, Bob)
- Talk(Alice, Carol)
- Talk(Bob, Alice)
- Talk(Bob, Bob)
- Talk(Bob, Carol)
- Talk(Carol, Alice)
- Talk(Carol, Bob)
- Talk(Carol, Carol)

Or, if a constraint hinders `FamilyMember`s from talking to themselves, one would expect the following possible events:

- Talk(Alice, Bob)
- Talk(Alice, Carol)
- Talk(Bob, Alice)
- Talk(Bob, Carol)
- Talk(Carol, Alice)
- Talk(Carol, Bob)

In other words, we need to be able to generate any combination of instances of the contexts in which the DSEs are defined. This is not possible in ECL as ECL Events are only defined within a context, that is they are only defined in the context of one class from the AS.

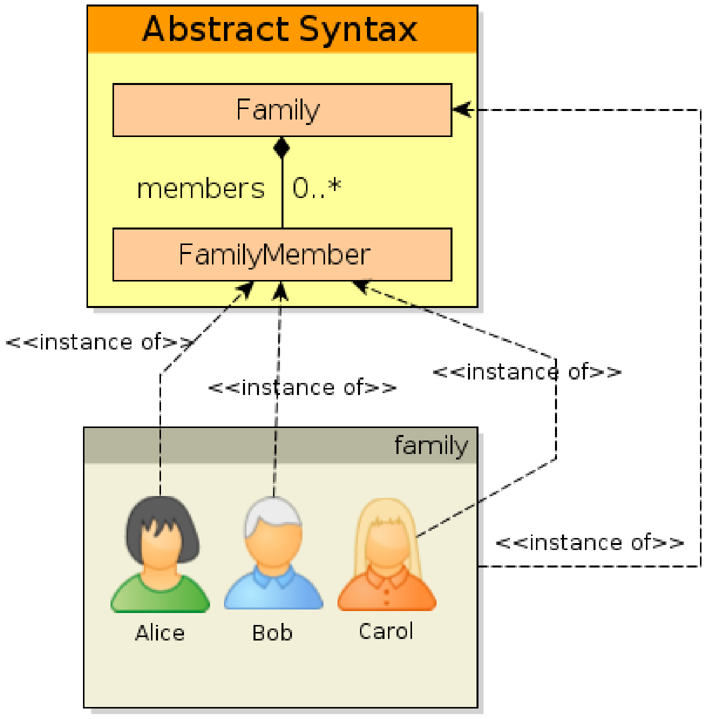

Figure 4.2: A `Family` composed of three `FamilyMembers`: Alice, Bob and Carole.

4.5.4 Feedback from DSA to MoCC

In ECL there is no way to specify the feedback needed for xDSMLs developed using GEMOC tools and methodology. Either another language or a new feature is required.

4.5.5 Lack of tracability between CCSL and ECL

When doing MoccEvent-targeting Feedback as described in subsection 3.5.1, the specification is written at the language level: the specification will be valid for every instance of the metaclass concerned and the MoccEvent targetted is the same for all these instances. But we need to be able to target a specific MoccEvent instance (that is, a CCSL Clock obtained after the compilation phase). However, there is no tracability mechanism between CCSL Clocks and ECL Events. This is due to the fact that ECL was developed after CCSL as a practical facility for specifying CCSL constraints at the language (AS) level. Therefore, the generation obtained from the ECL file should provide some facilities in order to provide, through an API, the links between CCSL clocks and ECL Events (and the other way around).

4.5.6 Events for internal use and for behavioral interface are mixed

We do not have the possibility to define which DSEs are to be seen as part of the behavioral interface of the xDSML and which are to be seen as only internal mechanisms to bridge Domain-Specific Action(s) with the MoCC as explained in 3.4.

4.5.7 Lack of mechanisms to help the reuse of DSAs and MoCCs

In Sections 3.2 and 3.3, we have shown the need for a mechanism to aggregate elements from the MoCC in order to connect a finely-grained MoCC with a set of DSAs, and for a mechanism to aggregate elements from the DSAs together in order to connect finely-grained DSAs with different MoCCs. However, this is not possible in ECL: DSEs are only linked to one DSA, and DSEs are indistinguishable from the MoccEvent (at the model level: clock) they are mapped to. It could be possible to aggregate elements from the MoCC, by defining new Expressions built on existing Relations and Expressions from the MoCC, but it would come with the price of requiring a re-analysis of these new Expressions to ensure the properties still hold. In other words, it would be akin to defining a new MoCC, which was not the aim in the first place.

5. Prospective Description of Gemoc Events Language

In this chapter, we present some extensions envisionned to tackle the limitations of the current approach. This will lead to a new language dedicated to the specification of the DSEs: **G**emoc **E**vents **L**anguage (GEL). The DSE specification will support the mapping between the MoCC and the DSAs of the operational semantics of an xDSML.

5.1 Motivations

The current approach has limitations which were explained in Chapter 4. GEL aims at compensating these limitations. In a first implementation, GEL was designed as a language complementing ECL, since there are still some issues for which we need ECL. In the following sections are shown the main features of GEL as well as possible upcoming features which would make it a legitimate evolution for ECL in our xDSML design approach.

Note: examples are given in terms of a possible concrete syntax to better illustrate the features. ”...” is used to mask unrelated features within an excerpt of code.

5.2 Features

GEL would provide a better means for defining the Domain-Specific Events of an xDSML defined using the GEMOC tools and methodology.

What we propose in GEL is a language to define the Domain-Specific Events of an xDSML, where a DSE must then be linked to a pattern of Domain-Specific Actions and to a pattern of MoccEvents. The feedback rules for DSEs can be specified too, and DSEs can be provided with more than one argument. A transformation can then generate, using the GEL file and a model conform to the AS of the xDSML, a GEL model which contains essentially the same information as the DSE specification but at the model level. This specification is then interpreted by the GEMOC Execution Engine for the execution of the model.

5.3 Dependencies

A GEL file depends on the following elements:

- Abstrax Syntax of the language (Ecore model)
- Domain-Specific Actions of the language (Implemented in Kermeta3)
- Library used for the MoCC of the language (ccslLib file)
- Optionally: an ECL file to define the MoccEvents and their constraints instanciating the MoCC library

5.4 Event declaration in GEL

In order to differentiate the two natures of Domain-Specific Events mentioned in subsection 4.5.6, we use visibility (`public` and `private`) to separate DSEs present in the behavioral interface of the xDSML and DSEs only used internally. As explained in section 3.4, a DSE is either used exclusively internally, or available to the environment (user through a GUI or other xDSML) in which case it must also be available internally. Concretely, all the DSEs must be usable internally, and some of these DSEs, pertaining to the behavioral interface, are also visible to the environment. Therefore, it is natural to use the mechanism of visibility, which is well-known in General-purpose Programming Languages, and which corresponds exactly to this situation: all members are in particular visible internally, and only some of these members are visible to exterior elements.

An example is given in listing 5.1 and listing 5.2.

Listing 5.1: A Domain-Specific Event which is part of the behavioral interface of the language.

```
public event MyPublicDse on {...} with {...}:
    calls ...
    when ...
end
```

*Listing 5.2: A Domain-Specific Event which is **NOT** part of the behavioral interface of the language.*

```
private event MyPrivateDse on {...} with {...}:
    calls ...
    when ...
end
```

5.5 Patterns of Domain-Specific Actions

To solve the problem mentioned in subsection 4.5.7 of not always being able to reuse existing Domain-Specific Actions without too much overhead, we propose the possibility to create patterns of DSAs. These patterns aim at regrouping several DSAs so that they can be triggered by the occurrence of an instance of a DSE. For now, we have mostly identified the need for the two following patterns:

- **Bag**: triggers a given bag of DSAs in any order (the order does not matter). See listing 5.3. This pattern supposes that the data manipulated by each referenced DSAs is independent one from another. Ideally, this property could be verified by static verification. It can also be linked to the idea of Constraints (contracts on the DSAs) developed in Deliverable D1.1.1. A concrete use of this pattern could be for a Domain-Specific Event of Petri net which would fire a specific group of independent `Transition`s at the same time. If that event is allowed, it means that all the `Transition`s can be fired independently from one another and that the order of firing does not matter.

  Listing 5.3: Action pattern Bag

  ```
  private event MyPrivateDse on {...} with {...}:
      calls Bag(... .MyDsa(),
                ... .MyOtherDsa(),
                ... .MyOtherOtherDsa()
           )
      when ...
  end
  ```

- **Sequence**: triggers a sequence of DSAs in the order in which they are given. See listing 5.4. This pattern is useful when an action has been split into two DSAs and we do not need or want to observe both steps, instead we only want to observe the model after both actions have been done. For example, suppose the DSA *fire()* of a Petri net `Transition` consumes the token from the incoming `Place`s and creates tokens in the outgoing `Place`s. This action can be separated in two DSAs: *consumeTokens()* which consumes the tokens from the incoming `Place`s and *produceTokens()* which produces the tokens in the outgoing `Place`s. Suppose we only want to observe the firing of a `Transition`, with no particular interest in these two steps individually. Then we must use the Sequence pattern of actions to ensure that we first consume the tokens before creating tokens in the outgoing `Place`s.

Listing 5.4: Action pattern Sequence

```
private event MyPrivateDse on {...} with {...}:
    calls Sequence(... .MyFirstDsa(),
                   ... .MySecondDsa(),
                   ... .MyThirdDsa()
         )
    when ...
end
```

5.6 Arguments of Domain-Specific Events

Domain-Specific Events are linked to at least one Domain-Specific Action. Since a DSA is essentially a method, and a method is attached to a class from the AS of the xDSML, a DSE is linked to at least one class from the AS. Therefore Domain-Specific Events are linked to at least one class from the AS. We will call this class the *context* of the DSE. Instead of inferring the context of the DSE from the DSA(s) it refers (what would happen when a DSE is linked to several DSAs?), it is more logical to add this *context* reference directly to the DSE, and to navigate from this context to find the class from the dsa. For example, in fUML, a DSE of context `ActivityNode` could trigger the DSAs *removeControlToken()* and *createControlToken()* from class `ActivityEdge`. However the concrete target of both DSAs would be different: for *removeControlToken()* it would be the incoming `ActivityEdge`s while for *createControlToken()* it would be the outgoing `ActivityEdge`s. This is possible by using the navigation starting from the context `ActivityNode` of the DSE.

In Listing 5.5, the *context* of a DSE is found after the keyword ”on”.

Since DSAs are implemented as methods, what about their parameters? Suppose a DSA *println* needs an argument of type String. There are several possible scenarios:

- Scenario 1: the DSA is called from a DSE originating from the environment (user through a GUI, other xDSML). In that case the DSE should also have a parameter of type String, and a mapping between the DSE's parameter and the DSA's parameter should be done.
- Scenario 2: the DSA is called from a private DSE or a public DSE originating from the MoCC. In that case, there are several possibilities. One is to give the value of the argument at design time: for example, the DSE could always call the DSA with the DSE's name as the String argument. Or it could also call the DSA using as argument a String returned by another DSA called by the same DSE. But if the String comes from a DSA refered by another DSE, then it gets more complicated because the String cannot be passed to the MoCC. One possible implementation could be to have a stack of arguments for each DSA or each DSE. Further versions of this document should bring more details on this issue.

For now, Domain-Specific Events will have [1..*] parameters in GEL so as to make Scenario 1 possible. A simple example is given in listing 5.5.

Listing 5.5: Arguments of a Domain-Specific Event

```
// This version implies that the initialize() DSA will initialize all the
   owned Places of the PetriNet
public event InitializePetriNet on {PetriNet petrinet}:
    calls petrinet.initialize()
    when ...
end
// OR
// This version implies that the initialize() DSA will only initialize places
   given to the PetriNet
public event InitializePetriNet on {PetriNet petrinet} with {Place place}:
    calls petrinet.initialize(place)
    when ...
end
```

One question still remains: how should we deal with arguments which are not just references to elements from the model? For example, what happens if a DSA, and thus the associated DSE, requires a String as parameter? public DSEs may be able to address this issue (the environment should provide a String, or be prompted for one), but private DSEs may be a bit more complicated: if the DSE requiring a String is triggered due to some feedback, then the String can be given in the feedback and stocked in a queue by the Execution Engine. However, what if this private DSE is triggered by internal constraints of the MoCC and the queue is empty? Should there be an error, or a default case? See listing 5.6 for a simple example.

Listing 5.6: Dealing with arguments of a DSE

```
public event PrintLog on {System system} with {String log}:
    calls system.out.println(log)
    when ...
end
```

5.7 MoccEvents

MoccEvents are the observation points on the control flow modelled by the MoCC whose instances are used to trigger occurrences of instances of Domain-Specific Events. For instance, suppose we have a specification where a DSE MyDSE of context C (a class from the AS) is mapped to the MoccEvent MyMoccEvent. The compilation (or generation) phase instanciates this specification down to the model level. Suppose our model has two elements conforming to the class C: c1 and c2. Then we will have two DSE instances, also called Model-Specific Events: MyDSE_c1 and MyDSE_c2. We will also have two instances of the MoccEvent MyMoccEvent: MyMoccEvent_c1 and MyMoccEvent_c2. Note that in our approach, instances of MoccEvents are CCSL clocks.

In our case, MoccEvents are in particular ECL Events constrained by MoCC relations and without any link to any DSA. Instead MoccEvents are referenced by the DSEs which also reference the DSAs. If we want to replace ECL, then we need an equivalent in GEL which is the notion of `trigger` illustrated in listing 5.7. The ”for” clause specifying the context of instanciation of the `trigger`.

The main difference between ECL Events and the `trigger`s of GEL is that ECL Events are limited to one argument (their context) and thus we cannot generate the ”Cartesian product” mentioned in subsection 4.5.3. Therefore, at some point we will need to either change the way ECL generates its CCSL specification, or use the `trigger`s in GEL and a generator towards CCSL.

Listing 5.7: MoccEvents with arguments in GEL. We want to generate, in terms of CCSL clocks, all the possible combinations of ($Printer \times Message$) in the model

```
trigger MyMoccEvent for {Printer printer, Message message}
```

5.8 Patterns of MoccEvents

If the MoCC was designed too finely-grained as mentioned in subsection 4.5.7, then we may need to use patterns in order to aggregate some MoccEvents together to represent the occurrence of an instance of a DSE. For now, we have mostly identified the following patterns of events:

- Coincidence: triggers when the given MoccEvents occur in the same step

Listing 5.8: Events pattern Coincidence

```
public event ResetPlace on {PetriNet petrinet} with {Place place}:
    calls place.initialize()
    when Coincidence(MyMoccEvent(petrinet, place),
                     MyMoccEvent2(petrinet)
                    )
end
```

- Absence: triggers when the given MoccEvent does not occur in a step
- Exclusion: triggers when only one of the given MoccEvents occurs in a step
- Followed-by: triggers when, in a sequence of steps, the first MoccEvent occurred in the first step of the sequence, the second MoccEvent occurred in the second step of the sequence, etc ...

We want to limit ourselves to patterns which are ”reversible”, so that the feedback specification can also be specified using DSEs and DSAs as explained in section 3.5. In this context, ”reversible” means that we are able to prevent the pattern from happening in the next step of execution. More concretely, when interpreting a Feedback rule which wants to forbid a given DSE for the next step of execution, and this DSE is mapped to the FollowdBy pattern of two MoccEvents m_1 and m_2 (m_1 FollowedBy m_2), then the solution is to first look at the last step of execution: if m_1 is not occurring, then the pattern m_1 FollowedBy m_2 will not occur anyway at the next step of execution ; if m_1 is occurring, then we must forbid the MoccEvent m_2 from occurring.

5.9 Feedback from DSA to MoCC

Given the place the Feedback has in the xDSML design, we propose to place the Feedback specification in GEL as well. An example of feedback rule is given in listing 5.9. The rule shown impacts a GEL `trigger`, but we could also target DSEs or DSAs as mentioned in chapter 3. We also limit ourselves to feedback rules applied as soon as possible for now, although as mentioned in chapter 3, further investigation could motivate timed feedback specifications.

Listing 5.9: Feedback forcing a specific MoccEvent (trigger) to true on the next step

```
feedback of EvaluateFireability on (Transition transition):
  true: force Fireable_True on transition
  false: forbid Fireable_True on transition
end
```

5.10 Instanciation of the MoCC

The instanciation of the MoCC using GEL depends on whether we want GEL to replace ECL or want it to complement ECL.

On the one hand, if GEL is to complement ECL, then the declaration of MoccEvents and the instanciation of the MoCC can be left in ECL while the GEL specification pertains only to the DSEs declaration, the DSA patterns, the MoccEvent patterns and the feedback specifications. However, this also constrains us with some of the current limitations of ECL: no arguments for the DSEs, strong coupling with CCSL and the other problems mentioned in section 4.5.

On the other hand, if GEL is to replace ECL, then the MoccEvents declaration can be done in GEL, and we need to be able to instanciate the CCSL constraints on the GEL `trigger`s. Ideally all the constraints used for the MoCC will be present in the MoCC library file, and the only thing changing from xDSML to another xDSML using the same MoCC is the arguments passed to the MoCC relations. Therefore we need to be able to do complex navigation inside the AS of the xDSML and we need to be able refer to CCSL Expressions within GEL.

Further versions of this document should improve on this issue and clarify the path chosen as well as its motivations.

5.11 Current state of implementation

For now, a very early implementation of GEL is available[1]. It is still based on ECL, so in that case ECL is used to instanciate the MoCC and define the MoccEvents on which patterns are then made in GEL. These patterns are then attached to a DSE, and a DSE is also attached to a pattern of DSAs. As for the feedback from DSAs to MoCC, for now it is only available as a Java interface to implement, and it is possible to force or forbid either a MoccEvent (ECL Event in that case) or a DSE. There is not any concrete syntax yet. There remains a lot of discussions to be had about the instanciation of MoCCs: should it be included in GEL? And how should it be done? Further versions of this deliverable should bring more details on these issues.

[1]On the GEMOC Git repository, branch feature-patterns, folder org/gemoc/execution/engine_model/

6. Examples

6.1 Petri net

Petri nets are a mathematical modeling language used for the description of distributed systems.

6.1.1 *Abstract Syntax*

The Abstract Syntax of Petri net is given as a Metamodel using Ecore. A graphical representation is shown in Figure 6.1. The root element is `PetriNet`, which is composed of several `PetriNetElements`, which

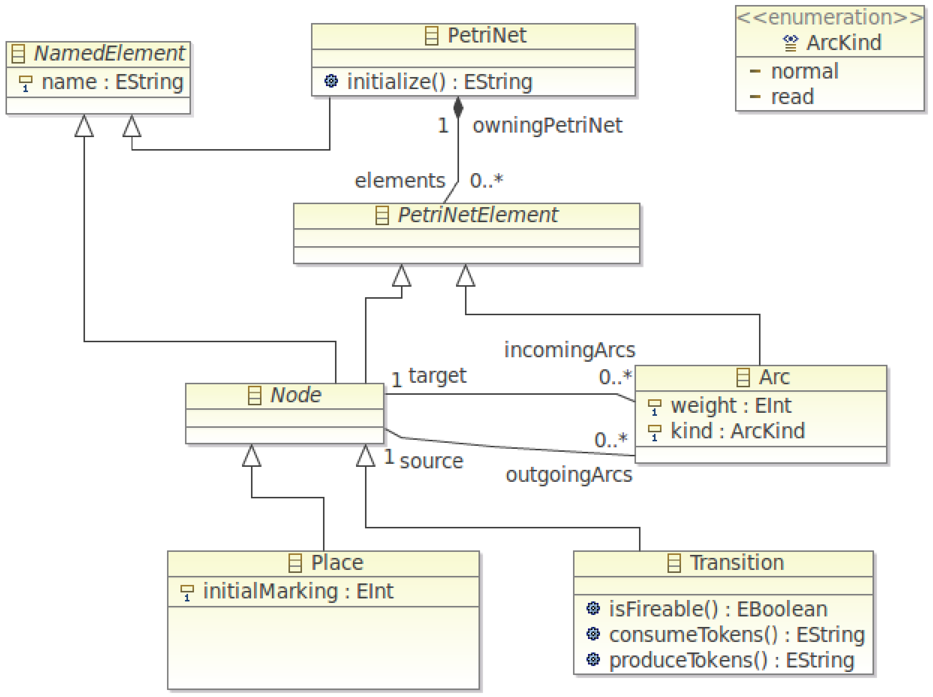

Figure 6.1: Metamodel of Petrinet

are either `Nodes` or `Arcs`. `Arcs` are of kind either ”normal” or ”read” (when read arcs are traversed, marking of the source `Place` is not decremented) and link two `Nodes` (references*source* and *target*). But these two `Nodes` must be of different natures: both *source* and *target* cannot be a `Place` at the same time, or be a `Transition` at the same time. `Place` and `Transition` are the two concrete types of `Nodes`. `Places` have an *initialMarking* attribute encoded as an integer. `Nodes` know about their outgoing and incoming `Arcs`.

Finally, for technical reasons, the signature of the DSAs must be present in the AS as EOperations.

6.1.2 *Execution Data*

The only Execution Data for Petri nets in our case is the marking of the `Places`. Listing 6.1 shows how this marking is implemented in Kermeta 3.

Listing 6.1: Execution Data for Petrinet

```
@Aspect(className=typeof(Place))
class PlaceAspect extends NodeAspect {
  private int currentMarking
}
```

6.1.3 Domain-Specific Actions

In our flavour of Petri nets, we have chosen to go with 5 Domain-Specific Actions, shown in Listing 6.2.

- **PetriNet.initialize() : String**: DSA action which initializes every `Place` of the Petri net by setting the value of *currentMarking* to the value of its *initialMarking*. Returns a String for practical logging purposes.
- **Transition.isFireable() : Boolean**: DSA action which returns whether or not the `Transition` is fireable. Fireability is determined by ensuring that every `Place` linked to the `Transition` by an incoming arc has a number of marking superior or equal to the weight of the arc.
- **Transition.consumeTokens() : String**: DSA action which decreases the marking of the `Place`s linked to the `Transition` by an incoming arc by the value of the weight of the arc if the arc is not a `READ` arc. Returns a String for practical logging purposes.
- **Transition.produceTokens() : String**: DSA action which increases the marking of the `Place`s linked to the `Transition` by an outgoing arc by the value of the weight of the arc. Returns a String for practical logging purposes.
- **Place.getCurrentMarking() : int**: DSA query which returns a representation of the Execution Data (a RunTime Data).

Listing 6.2: Domain-Specific Actions for Petrinet

```
@Aspect(className=typeof(Place))
class PlaceAspect extends NodeAspect {
  public int getCurrentMarking(){
    return _self.currentMarking
  }
}

@Aspect(className=typeof(Transition))
class TransitionAspect extends NodeAspect {

  // All the incoming Places must have a current Marking superior or equal to
  // the weight of the arc between the Transition and the considered Place.
  def public Boolean isFireable() {
    var res = true
    for (arc : _self.incomingArcs) {
      res = res && (arc.source as Place).currentMarking >= arc.weight
    }
    return res
  }

  def public String consumeTokens() {
    // Consuming the tokens from the normal arcs
    _self.incomingArcs.forEach(
      arc|
        if (arc.kind != ArcKind.READ) {
          (arc.source as Place).currentMarking = (arc.source as Place).
              currentMarking - arc.weight
        }
    )
    // Create a nice log to display what happened
    var log = ...

    return log
  }

  def public String produceTokens() {
    // Produce the tokens on the outgoing Places
    _self.outgoingArcs.forEach(
      arc|(arc.target as Place).currentMarking = (arc.target as Place).
          currentMarking + arc.weight
    )
    // Create a nice log to display what happened
    var log = ...

    return log
  }

}

@Aspect(className=typeof(PetriNet))
class PetriNetAspect extends NamedElementAspect {
  // Sets the marking of each Place to its initial vlaue
  def public String initialize() {
    var allPlaces = _self.elements.filter(Place)
    allPlaces.forEach(place|place.currentMarking = place.initialMarking)
    // Create a nice log to display what happened
    var log = ...

    return log
  }
}
```

6.1.4 MoCC Library

The domain-agnostic CCSL Relations and Expressions of the MoCC used for our Petrinet xDSML can be gathered in a CCSL Library. In our case, we have 4 CCSL Relations and no Expressions. First, *FirstAndOnlyOnce(mocc_firstEvent, mocc_otherEvents)* allows us to do something once and only once, before anything else happens. Then, *EvaluationLeadsToABooleanResult(mocc_evaluation, mocc_true, mocc_false)* is used so that when we ask for an evaluation of the fireability of a `Transition`, we are guaranteed to see the result on one of the two clocks mocc_true or mocc_false given as argument. In the case where mocc_true is the one triggered, then thanks to the relation *TruePrecedesTriggered(mocc_true, mocc_triggered)*, we ensure that we will fire any `Transition` which is fireable. Finally, our main loop of execution is in the relation *MainLoopAlternance(mocc_evaluation, mocc_action)* which creates an alternance between, in our case, the evaluation of the fireability of the `Transition`s and their firing.

Listing 6.3: MoCC library for the Petrinet xDSML

```
Library petrinet_moc{

  imports{
    import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
       ccsllibrary/kernel.ccslLib" as kernel;
    import "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
       ccsllibrary/CCSL.ccslLib" as CCSLLib;
  }

  RelationLibrary petrinet_relations{
    /**
     * One tick of mocc_firstEvent once and before any tick of
        mocc_otherEvents.
     * mocc_otherEvents should be an expression gathering all the other events
     * (or at least all the events that would be scheduled first otherwise).
     */
    RelationDeclaration FirstAndOnlyOnce(mocc_firstEvent : clock,
       mocc_otherEvents : clock)
    RelationDefinition FirstAndOnlyOnce[FirstAndOnlyOnce]{
      Expression firstTickOfFirstEvent =
        OneTickAndNoMore(OneTickAndNoMoreClock -> mocc_firstEvent)
      Expression firstTickOfOtherEvents =
        OneTickAndNoMore(OneTickAndNoMoreClock -> mocc_otherEvents)
      Relation Precedes(
        LeftClock -> mocc_firstEvent,
        RightClock -> firstTickOfOtherEvents
      )
      Relation Coincides(
        Clock1 -> mocc_firstEvent,
        Clock2 -> firstTickOfFirstEvent
      )
    }

    /**
     * When there is a tick of mocc_evaluation, it is followed by
     * a tick of either mocc_true or mocc_false.
     */
    RelationDeclaration EvaluationLeadsToABooleanResult(mocc_evaluation :
       clock, mocc_true : clock, mocc_false : clock)
    RelationDefinition EvaluationLeadsToABooleanResult[
       EvaluationLeadsToABooleanResult]{
      Expression trueOrFalse = Union(Clock1 -> mocc_true, Clock2 -> mocc_false
         )
      Relation evaluationPrecedesResult[Precedes](
        LeftClock -> mocc_evaluation,
        RightClock -> trueOrFalse
      )
      Relation resultIsTrueXorFalse[Exclusion](
        Clock1 -> mocc_true,
        Clock2 -> mocc_false
      )
    }
    /**
     * When an evaluation has returned true, then either the associated
     * action should be triggered.
     */
    RelationDeclaration TruePrecedesTriggered(mocc_true : clock,
        mocc_triggered : clock)
    RelationDefinition TruePrecedesTriggered[TruePrecedesTriggered]{
      Relation truePrecedesTriggered[Precedes](
        LeftClock -> mocc_true,
        RightClock -> mocc_triggered
      )
    }

    /**
     * The core of this controlflow: alternance between evaluating what can be
         triggered and triggering.
     */
    RelationDeclaration MainLoopAlternance(mocc_evaluation : clock,
        mocc_action : clock)
    RelationDefinition MainLoopAlternance[MainLoopAlternance]{
      Relation exclusion[Exclusion](
        Clock1 -> mocc_evaluation,
        Clock2 -> mocc_action
      )
      Relation alternance[Alternates](
        AlternatesLeftClock -> mocc_evaluation,
        AlternatesRightClock -> mocc_action
      )
    }


  }
  ExpressionLibrary petrinet_expressions{

  }
}
```

6.1.5 Domain-Specific Events specification

The Domain-Specific Events are the observation points of the behavior of our xDSML. What are the observation points of the behavior of Petri nets?

First, we need to understand the semantics that we want to give to our xDSML. Ideally, one would initialize their `PetriNet`, and then find all the fireable `Transition`s. After that, they would display the fireable `Transition`s to the environment, which would select one of the `Transition`s to fire. The selected `Transition` would then be fired, first by consuming the tokens from the incoming `Place`s and then by producing tokens on the outgoing `Place`s. Then go back to finding all the fireable `Transition`s.

This can be summed up in the pseudo-code given in listing 6.4.

Listing 6.4: Algorithm in pseudo-code of the execution of a PetriNet model

```
initialize PetriNet
while(true)
```

```
    evaluate fireability of all Transitions
    fire one of the transitions (consume tokens and produce tokens)
end
```

In this pseudo-code, the DSAs mentioned are what we want to observe. These are: the initialization of the `PetriNet` (DSE *InitializePetriNet*), the evaluation of the fireability of the `Transition`s (DSE *EvaluateFireability*) and the firing of the `Transition`s (DSE *FireTransition*). However, only the selection of which `Transition` to fire requires the environment to intervene ; therefore the only public DSE should be *FireTransition*. The other DSEs should be private. The backbone of the DSE specification is available in Listing 6.5

Listing 6.5: Backbone of the DSE specification for the Petrinet xDSML

```
public event InitializePetriNet on {PetriNet petrinet}:
    calls petrinet.initialize()
    when ...
end

private event EvaluateFireability on {Transition transition}:
    calls transition.isFireable()
    when ...
end

private event FireTransition on {Transition transition}:
    calls Sequence(transition.consumeTokens(),
                   transition.produceTokens()
                  )
    when ...
end
```

Listing 6.5 is now missing only two elements: the mapping between DSE and MoccEvents, and the feedback specification. Indeed, the evaluation of the fireability of a `Transition` returns a boolean which is only known at runtime. The MoCC knows how to handle both situations, but it needs to be told about which of the two situations is actually happening at runtime. Thus, we need to add a feedback specification which will target specific MoccEvents, as shown in listing 6.6.

Listing 6.6: Feedback specification for the Petrinet xDSML

```
feedback of EvaluateFireability on {Transition transition}:
    true: force mocc_isFireableTrue on transition
    false: force mocc_isFireableFalse on transition
end
```

This feedback specification implies that we need at least two MoccEvents in the `Transition` context: *mocc_isFireableTrue* and *mocc_isFireableFalse*. Now, we need to accomodate with the MoCC we are using. If the MoCC is a perfect fit for our DSEs, then each DSE will be mapped to one MoccEvent. If it is not, then we may need to create patterns of MoccEvents. For now, let us use only one MoccEvent per DSE: *mocc_initialize* for *InitializePetriNet*, *mocc_isFireable* for *EvaluateFireability* and *mocc_fire* for *FireTransition*. The full DSE specification is given in listing 6.7.

Listing 6.7: DSE for the Petrinet xDSML

```
public event InitializePetriNet on {PetriNet petrinet}:
    calls petrinet.initialize()
    when mocc_initialize on petrinet
end

```

```
private event EvaluateFireability on {Transition transition}:
    calls transition.isFireable()
    when mocc_isFireable on transition
end

private event FireTransition on {Transition transition}:
    calls Sequence(transition.consumeTokens(),
                   transition.produceTokens()
                  )
    when mocc_fire on transition
end

feedback of EvaluateFireability on {Transition transition}:
    true: force mocc_isFireableTrue on transition
    false: force mocc_isFireableFalse on transition
end
```

6.1.6 MoccEvents in ECL

Using ECL, we can declare MoccEvents (language-level observable CCSL clocks) and instanciate the MoCC. We know from the previous subsection that we will need at least 5 MoccEvents:

- mocc_isFireableTrue and mocc_isFireableFalse on `Transition` for the feedback specification;
- mocc_fire on `Transition` for the DSE *FireTransition*;
- mocc_isFireable on `Transition` for the DSE *EvaluateFireability*;
- mocc_initialize on `PetriNet` for the DSE *InitializePetriNet*.

These 5 declarations can be found in the top half of listing 6.8.

Listing 6.8: MoccEvents in ECL for PetriNet

```
import 'platform:/plugin/org.gemoc.sample.petrinet.model/model/Petrinet.ecore'

ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
   ccsllibrary/kernel.ccslLib"
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
   ccsllibrary/CCSL.ccslLib"
ECLimport "platform:/resource/org.gemoc.sample.petrinet.moc/ccsl/petrinet.
   ccslLib"

package petrinet

    -- MoCC Events to be mapped to public Domain-Specific Events.
    -- ==========================================================
    context Transition
        def: mocc_fire : Event = DSE

    -- MoCC Events to be mapped to private Domain-Specific Events.
    -- ===========================================================
    context PetriNet
        def: mocc_initialize : Event = DSE
    context Transition
        def: mocc_isFireable : Event = DSE

    -- MoCC Events needed to encode a result through feedback.
    -- =======================================================
    context Transition
        def: mocc_isFireableTrue : Event = internalMoCCEvent
        def: mocc_isFireableFalse : Event = internalMoCCEvent

    -- MoCC Instanciation.
    -- ===================
    context PetriNet
        inv OneInitComesFirst:
          let allOtherEvents: Event = Expression Union(self.elements->select(
             element|
           ((element).oclIsKindOf(Transition))).oclAsType(Transition)->first
              ().mocc_isFireable)
          in Relation FirstAndOnlyOnce(self.mocc_initialize, allOtherEvents)

        inv synchronousEvaluationOfFireability:
            Relation Coincides(self.elements->select(element|
                ((element).oclIsKindOf(Transition))).oclAsType(Transition).
                   mocc_isFireable)

        inv oneTransitionAtATime:
          Relation Exclusion(self.elements->select(element|
            ((element).oclIsKindOf(Transition))).oclAsType(Transition).
               mocc_fire)

    context Transition
        inv evaluationOfFireabilityLeadsToTrueXorFalse:
            Relation EvaluationLeadsToABooleanResult(mocc_isFireable,
               mocc_isFireableTrue, mocc_isFireableFalse)

        inv trueMustPrecedeFire:
            Relation TruePrecedesTriggered(mocc_isFireableTrue, mocc_fire)

        inv mainLoop:
            let action : Event = Expression Union(mocc_fire,
               mocc_isFireableFalse) in
            Relation MainLoopAlternance(self.mocc_isFireable, action)
endpackage
```

The bottom half of listing 6.8 are dedicated to the instanciation of the MoCC using the MoCC Library defined earlier. Here we use the Relation *FirstAndOnlyOnce* in the context of `PetriNet` so as to schedule the MoccEvent *mocc_initialize* before any *mocc_isFireable* (using a Union Expression). We also use two Relations from the kernel of CCSL: *Coincides* on all the *mocc_isFireable* so that all the evaluations of fireability happen in the same step, and *Exclusion* on all the *mocc_fire* so that only one `Transition` can be fired at a time. Then in the context of `Transition` we use the three other Relations from the MoCC Library: *EvaluationLeadsToABooleanResult* with *mocc_isFireable*, *mocc_isFireableTrue* and *mocc_isFireableFalse* so as to encode the result of the evaluation of the fireability of a `Transition`; *TruePrecedesTriggered* with *mocc_isFireableTrue* and *mocc_fire* so as to ensure that before firing a `Transition`, it was fireable; finally *MainLoopAlternance* with *mocc_isFireable* with a CCSL Expression which is the Union of *mocc_fire* and *mocc_isFireableFalse* to ensure that during the "while(true)" loop of the algorithm described in 6.4, for a given Transition, it is either being evaluated for fireability, or fired, or it was not fireable in the first place.

Since we did not need any additional MoccEvent, the DSE specification does not need to be changed.

6.2 fUML

fUML stands for Foundational UML and is a foundational executable set of UML standardized by the OMG[1].

6.2.1 Abstract Syntax

The Abstract Syntax of fUML is given as a Metamodel using Ecore. A graphical representation is shown in Figure 6.2. The root element is Activity, which is composed of ActivityNodes and ActivityEdges.

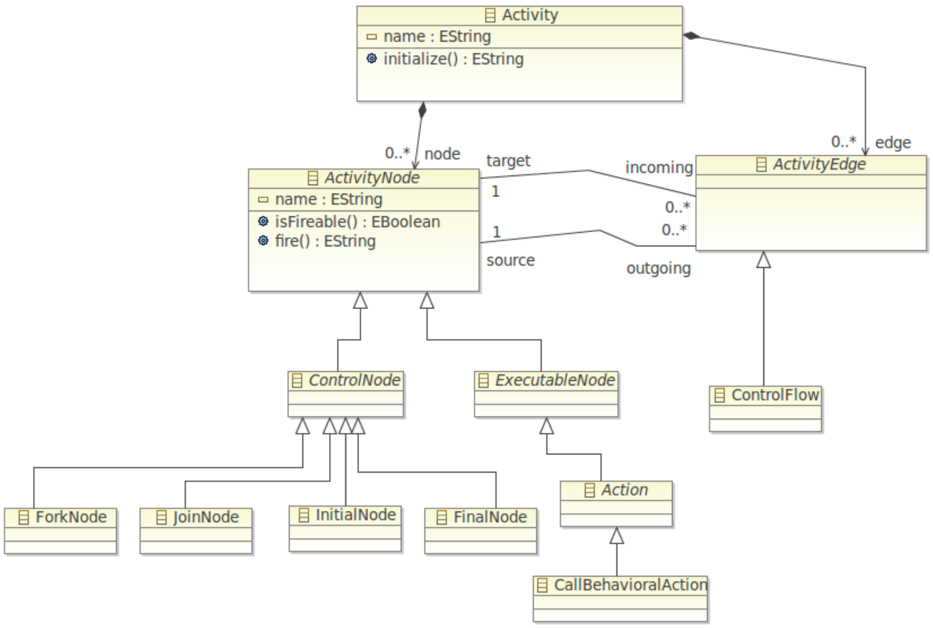

Figure 6.2: Metamodel of fUML

An ActivityEdge in our case (simplified fUML) can only be a ControlFlow. An ActivityNode can be of two nature: either a ControlNode or an ExecutableNode. A ControlNode can be either a ForkNode, a JoinNode, an InitialNode or a FinalNode. An ExecutableNode can only be an Action which, in turn, can only be n CallBehavioralAction.

Finally, for technical reasons, the signature of the DSAs must be present in the AS as EOperations.

6.2.2 Execution Data

The Execution Data for fUML is the notion of ControlToken on ActivityEdges. Listing 6.9 shows how this marking is implemented in Kermeta 3.

Listing 6.9: Execution Data for fUML

```
class ControlToken{
}
```

[1]http://www.omg.org/spec/FUML/1.1/

```
@Aspect(className=typeof(ActivityEdge))
abstract class ActivityEdgeAspect {
  private ControlToken storedControlToken
}
```

6.2.3 Domain-Specific Actions

In our simplified fUML, we have 6 Domain-Specific Actions, shown in Listing 6.10.

- **Activity.initialize() : String**: DSA action which only returns a log String.
- **ActivityNode.isFireable() : Boolean**: DSA action which returns whether or not the `ActivityNode` is fireable, by checking whether or not the input edges have a `ControlToken`. An InitialNode can only be fireable once.
- **ActivityNode.fire() : String**: DSA action which removess the `ControlToken`s from the input edges and creates `ControlToken`s on the output edges.
- **ActivityEdge.hasControlToken() : Boolean**: DSA query which returns a boolean value which is a RunTime Data since it is a representation of the internal state of the model.
- **ActivityEdge.setControlToken(ControlToken) : void**: DSA helper used in the DSA action *fire()*.
- **ActivityNode.execute() : String**: DSA helper abstract which can be used to specialize the behavior of a concrete `ActivityNode`.

Listing 6.10: Domain-Specific Actions for fUML

```
@Aspect(className=typeof(Activity))
class ActivityAspect {
  // DSA action
  def public String initialize(){
    return "[" + _self.getClass().getSimpleName() + ":" + _self.getName() + ".
        initialize()]The␣InitialNodes␣of␣the␣Activity␣are:␣" + _self.node.
        filter(InitialNode).toString()
  }
}

class ControlToken{
}
@Aspect(className=typeof(ActivityEdge))
abstract class ActivityEdgeAspect {
  // DSA query which returns a RunTime Data
  def public Boolean hasControlToken(){
    return _self.storedControlToken != null
  }

  // DSA helper which is used by the DSA actions
  def package void setControlToken(ControlToken token){
    _self.storedControlToken = token;
  }
}

@Aspect(className=typeof(ActivityNode))
abstract class ActivityNodeAspect {
  // DSA query using the DSA query hasControlToken()
  def public Boolean isFireable(){
    for(ActivityEdge edge : _self.incoming){
      if(! edge.hasControlToken()){
        return false
      }
    }
    return true
  }

  // DSA action using a DSA helper to modify the Execution State
  def public String fire(){
    for(ActivityEdge edge: _self.incoming){
      edge.setControlToken(null)
    }
    for(ActivityEdge edge: _self.outgoing){
      edge.setControlToken(new ControlToken())
    }
    var log = "[" + _self.getClass().getSimpleName() + ":" + _self.getName() +
        ".fire()]"
    var concreteLog = _self.execute()
    return log + concreteLog
  }

  // DSA helper in case we need to add other effects to the execution of each
      concrete classes
  def protected String execute()
}
```

6.2.4 MoCC Library

The semantics of fUML activities are similar to Petri nets, thus we are able to reuse the MoCC Library defined in the previous example.

6.2.5 Domain-Specific Events specification

The Domain-Specific Events are the observation points of the behavior of our xDSML. What are the observation points of the behavior of simplified fUML?

First, we need to understand the semantics that we want to give to our xDSML. Ideally, one would initialize their `Activity`, and then find all the fireable `ActivityNode`s. After that, they would display the fireable `ActivityNode`s to the environment, which would select one of the `ActivityNode`s to fire. The selected `ActivityNode` would then be fired. Then go back to finding all the fireable `ActivityNode`s.

This can be summed up in the pseudo-code given in listing 6.11.

Listing 6.11: Algorithm in pseudo-code of the execution of a simplified fUML model

```
initialize Activity
while(true)
    evaluate fireability of all ActivityNodes
    fire one of the ActivityNodes
end
```

In this pseudo-code, the DSAs mentioned are what we want to observe. These are: the initialization of the `Activity` (DSE *InitializeActivity*), the evaluation of the fireability of the `ActivityNode`s (DSE *EvaluateFireability*) and the firing of the `ActivityNode`s (DSE *FireNode*). However, only the selection of which `ActivityNode` to fire requires the environment to intervene ; therefore the only public DSE should be *FireNode*. The other DSEs should be private. The backbone of the DSE specification is available in Listing 6.12

Listing 6.12: Backbone of the DSE specification for the simplified fUML xDSML

```
public event InitializeActivity on {Activity activity}:
    calls activity.initialize()
    when ...
end

private event EvaluateFireability on {ActivityNode node}:
    calls node.isFireable()
    when ...
end

private event FireNode on {ActivityNode node}:
    calls node.fire()
    when ...
end
```

Listing 6.12 is now missing only two elements: the mapping between DSE and MoccEvents, and the feedback specification. Indeed, the evaluation of the fireability of an `ActivityNode` returns a boolean which is only known at runtime. The MoCC knows how to handle both situations, but it needs to be told about which of the two situations is actually happening at runtime. Thus, we need to add a feedback specification which will target specific MoccEvents, as shown in listing 6.13.

Listing 6.13: Feedback specification for the simplified fUML xDSML

```
feedback of EvaluateFireability on {ActivityNode node}:
    true: force mocc_isFireableTrue on node
    false: force mocc_isFireableFalse on node
end
```

This feedback specification implies that we need at least two MoccEvents in the `ActivityNode` context: *mocc_isFireableTrue* and *mocc_isFireableFalse*. Now, we need to accomodate with the MoCC we are using. If the MoCC is a perfect fit for our DSEs, then each DSE will be mapped to one MoccEvent. If it is not, then we may need to create patterns of MoccEvents. For now, let us use only one MoccEvent per DSE: *mocc_initialize* for *InitializeActivity*, *mocc_isFireable* for *EvaluateFireability* and *mocc_fire* for *FireNode*. The full DSE specification is given in listing 6.14.

Listing 6.14: DSE for the simplified fUML xDSML

```
public event InitializeActivity on {Activity activity}:
    calls activity.initialize()
    when mocc_initialize on activity
end

private event EvaluateFireability on {ActivityNode node}:
    calls node.isFireable()
    when mocc_isFireable on node
end

private event FireNode on {ActivityNode node}:
    calls node.fire()
    when mocc_fire on node
end

feedback of EvaluateFireability on {ActivityNode node}:
    true: force mocc_isFireableTrue on node
    false: force mocc_isFireableFalse on node
end
```

6.2.6 MoccEvents in ECL

Using ECL, we can declare MoccEvents (language-level observable CCSL clocks) and instanciate the MoCC. We know from the previous subsection that we will need at least 5 MoccEvents:

- mocc_isFireableTrue and mocc_isFireableFalse on `ActivityNode` for the feedback specification;
- mocc_fire on `ActivityNode` for the DSE *FireNode*;
- mocc_isFireable on `ActivityNode` for the DSE *EvaluateFireability*;
- mocc_initialize on `Activity` for the DSE *InitializeActivity*.

These 5 declarations can be found in the top half of listing 6.15.

Listing 6.15: MoccEvents in ECL for simplified fUML

```
import 'platform:/plugin/org.gemoc.sample.fuml.model/model/fumlsimplified.
    ecore'

ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
    ccsllibrary/kernel.ccslLib"
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/
    ccsllibrary/CCSL.ccslLib"
ECLimport "platform:/resource/org.gemoc.sample.petrinet.moc/ccsl/petrinet.
    ccslLib"

package fumlsimplified

    -- MoCC Events to be mapped to public Domain-Specific Events.
    -- ============================================================
    context ActivityNode
        def: mocc_fire : Event = DSE

    -- MoCC Events to be mapped to private Domain-Specific Events.
    -- ===========================================================
    context Activity
        def: mocc_initialize : Event = DSE

    context ActivityNode
        def: mocc_isFireable : Event = DSE

    -- MoCC Events needed to encode a result through feedback.
    -- ========================================================
    context ActivityNode
        def: mocc_isFireableTrue : Event = internalMoCCEvent
        def: mocc_isFireableFalse : Event = internalMoCCEvent

    -- MoCC Instanciation.
    -- ===================
    context Activity
    inv OneInitComesFirst:
      Relation FirstAndOnlyOnce(self.mocc_initialize, self.node->select(
         element|
        ((element).oclIsKindOf(ActivityNode))).oclAsType(ActivityNode)->first
           ().mocc_isFireable)

        inv synchronousEvaluationOfFireability:
            Relation Coincides(self.node->select(element|
                ((element).oclIsKindOf(ActivityNode))).oclAsType(ActivityNode)
                   .mocc_isFireable)

        inv oneActivityNodeAtATime:
          Relation Exclusion(self.node->select(element|
            ((element).oclIsKindOf(ActivityNode))).oclAsType(ActivityNode).
               mocc_fire)

    context ActivityNode
        inv evaluationOfFireabilityLeadsToTrueXorFalse:
            Relation EvaluationLeadsToABooleanResult(mocc_isFireable,
               mocc_isFireableTrue, mocc_isFireableFalse)

        inv trueMustPrecedeFire:
            Relation TruePrecedesTriggered(mocc_isFireableTrue, mocc_fire)

        inv mainLoop:
            let action : Event = Expression Union(mocc_fire,
               mocc_isFireableFalse) in
            Relation MainLoopAlternance(self.mocc_isFireable, action)
endpackage
```

The bottom half of listing 6.15 are dedicated to the instanciation of the MoCC using the MoCC Library defined earlier. Here we use the Relation *FirstAndOnlyOnce* in the context of `Activity` so as to schedule the MoccEvent *mocc_initialize* before any *mocc_isFireable* (using a Union Expression). We also use two Relations from the kernel of CCSL: *Coincides* on all the *mocc_isFireable* so that all the evaluations of fireability happen in the same step, and *Exclusion* on all the *mocc_fire* so that only one `ActivityNode` can be fired at a time. Then in the context of `ActivityNode` we use the three other Relations from the MoCC Library: *EvaluationLeadsToABooleanResult* with *mocc_isFireable*, *mocc_isFireableTrue* and *mocc_isFireableFalse* so as to encode the result of the evaluation of the fireability of a `ActivityNode`; *TruePrecedesTriggered* with *mocc_isFireableTrue* and *mocc_fire* so as to ensure that before firing a `ActivityNode`, it was fireable; finally *MainLoopAlternance* with *mocc_isFireable* with a CCSL Expression which is the Union of *mocc_fire* and *mocc_isFireableFalse* to ensure that during the "while(true)" loop of the algorithm described in 6.11, for a given `ActivityNode`, it is either being evaluated for fireability, or fired, or it was not fireable in the first place.

Since we did not need any additional MoccEvent, the DSE specification does not need to be changed.

7. Conclusion

The separation of concerns in GEMOC between the control flow of a language and its actions 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 and are able to provide a first solution for some of these challenges, solution which still needs to be fully implemented. Some issues are still quite open, like the different types of feedbacks and the use of timed feedbacks, or the addition of arguments to Domain-Specific Events. Further improvements on specification and implementation should provide us with a more detailed description of the needs of the mapping between MoCC and DSAs.

8. 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] Julien Deantoni and Frédéric Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Rapport de recherche RR-8031, INRIA, July 2012.

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

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