Grant ANR-12-INSE-0011

ANR INS GEMOC

D3.3.1 - Formalization and restriction for the DSML operational semantics

Task 3.3

Version 2.0

DOCUMENT CONTROL

	–: 2014/05/22	A:	B:	C:	D:
Written by Signature	Xavier Crégut *et al.*	Florent Latombe			
Approved by Signature					

Revision index	Modifications
–	version 1.0 — First version
A	version 2.0 — Added section about the feedback
B	
C	
D	

Authors

Author	Partner	Role
Xavier Crégut	IRIT - Université de Toulouse	Lead author
Joël Champeau	ENSTA Bretagne	Contributor
Benoît Combemale	INRIA	Contributor
Julien DeAntoni	I3S lab- Université de Nice Sophia Antipolis	Contributor
Florent Latombe	IRIT - Université de Toulouse	Contributor
Marc Pantel	IRIT - Université de Toulouse	Contributor
François Tanguy	INRIA	Contributor
Matias Vara larsen	I3S lab- Université de Nice Sophia Antipolis	Contributor

Contents

1 Introduction **5**
1.1 Purpose . . . 5
1.2 Perimeter . . . 5
1.3 Definitions, Acronyms and Abbreviations . . . 5
1.4 Summary . . . 7

2 Formalization and Restriction of the Domain-Specific Actions **7**
2.1 Definition of the Execution Data . . . 7
2.2 Definition of the Execution Functions . . . 7
2.2.1 Execution Function Specification . . . 7
2.2.2 Execution Function Implementation . . . 8
2.2.3 Taxonomy of Execution Functions . . . 8
2.2.4 Constraints on the Execution Functions . . . 9
2.2.5 Implementation of Execution Functions: DSA action language . . . 9

3 Formalization and Restriction of the Domain-Specific Events **10**
3.1 Coincident Domain-Specific Events . . . 10
3.2 Coherence with the MoCC concerning the Feedback mechanism . . . 10

4 Static Verification of Constraints **11**
4.1 Motivation . . . 11
4.2 Analysis of the implementation of the Execution Functions . . . 11
4.2.1 Constraints intrinsic to the DSA . . . 11
4.2.2 Constraints induced by the DSE . . . 11
4.3 Verification of formal contracts . . . 12

5 Dynamic Verification of Constraints **12**

6 Conclusion **12**

7 References **13**

1. Introduction

1.1 Purpose

This document establishes the formal link between the language for MoCC definition and the domain-specific concerns for a given xDSML. More precisely, the goal is to define both an adequate abstraction of the operational semantics modeling language used in WP1 as well as restrictions on the implementation language used in WP4 (Kermeta) to harness the interaction between xDSML and MoCC definitions (e.g. avoid side effects). This work takes place at the specification level but also at the implementation level. At specification level, it must define the minimum relevant numbers of concepts to define an operational semantics. At the implementation level, it must provide a formal semantics, respectful of that specification, to a reasonable subset of Kermeta 3, the concrete language in which most of the operational semantics of xDSMLs will be implemented.

1.2 Perimeter

The GEMOC project aims to develop a studio providing a Language Workbench to define xDSMLs and a modeling Workbench to create and execute models conform to these xDSMLs.

We rely on an operational semantics approach. The operational semantics is split into two sections as per [1] (see figure 1.1). The first one, the DSA, is broadly defined as the ED— the set of attributes, references and classes needed to represent the runtime state of the model — and the EF— the atomic operations that modify these elements. The DSA are an extension of the abstract syntax (AS) and dependant of the domain addressed by the xDSML. The second one, defined as the MoCC, is the scheduling policy of the above-mentioned operations. It handles the concurrent aspects of the xDSML.

To map a MoCC onto the DSA of an xDSML and to favor their reuse, a third facility has been added as the Domain Specific Events (DSE). DSEs are triggered either by the MoCC or by external interacting models (Behavioral Model Composition Operators). When occurring, they trigger the corresponding Execution Function(s). Some result may have to be communicated to the MoCC. The overall architecture and the methodology are described in WP1 (see Deliverable D3.3.1), while the MoCC aspects are further described in WP2 and the composition operators are described in WP3.

In this deliverable, we focus on two aspects :

1. The restrictions that must be imposed on the action language used to implement DSA in order to avoid side effects interacting with the MoCC definitions.
2. The specification of the correct use of the DSA by the MoCC.

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.

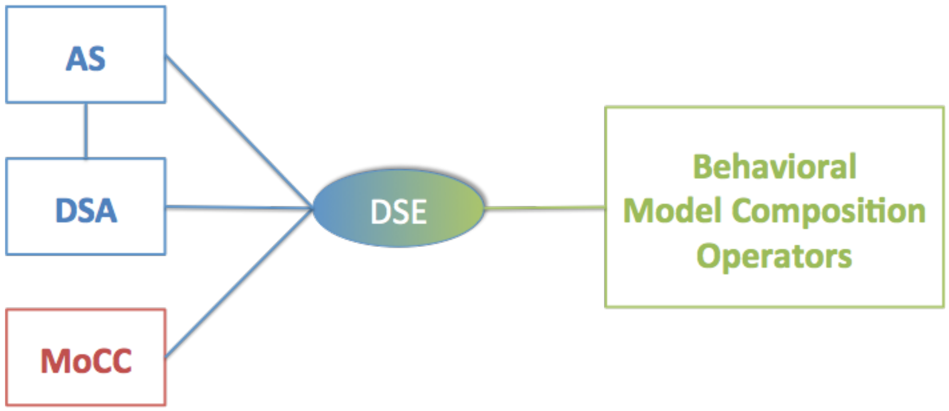

Figure 1.1: An eXecutable Modeling Language "a la" GEMOC

- **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 (part of DSA).
- **EF**: Execution Function (part of DSA).
- **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 provided an introduction to this document. It defined the perimeter and the goals of this document. Chapter 2 presents the constraints on the DSA part of an xDSML both on the language used to implement the DSA and on its uses to define the ED and the EF. Chapter 3 lists the constraints on the DSEs implied by the use of a given set of DSA and of a given MoCC. Chapters 4 and 5 explain how the requirements listed in the previous chapters can be verified either by static analysis tools (before any execution of any xDSML model) or dynamically (at runtime, when a model conform to an xDSML is being executed).

2. Formalization and Restriction of the Domain-Specific Actions

The DSA part of a language aims to define the operational semantics of an xDSML without dealing with the concurrency aspects which concern the MoCC. It is composed of two parts. The first part defines data managed at runtime as part of the execution semantics. These runtime data are called **Execution Data** (ED). The second part is composed of functions which are primitive actions of the execution semantics that manage the Execution Data. They are called the **Execution Functions** (EF).

One important choice made while defining the DSA language is the following. We do not target a light-weight language but a language that is easy to read, easy to control for correctness, and that respects the fundamental choices of the GEMOC project. Thus, the syntax may seem a bit verbose for programmers as they will be enforced to explicit all their intentions. The counterpart is that verification tools can take advantage of the intentions expressed by the programmers to more precisely check their code for errors and check its compliance according to the GEMOC approach. In particular, this proves useful for us as designing an xDSML is complex and hard to test: therefore, the more can be checked by construction, the better.

2.1 Definition of the Execution Data

The Execution Data are data that has to be managed at runtime to execute a model. They are defined as an Ecore metamodel (with possibly OCL constraints). This Ecore metamodel is generally an extension of the AS as most ED are defined in the context of a concept from the AS. Elements from the AS cannot be considered as part of the Execution Data.

The MOF 'merge' operator or the Kermeta 'Aspect' construct may be used to define the Execution Data metamodel as an extension of the xDSML's AS. In this case, the Execution Data metamodel is the Kermeta aspect, not the merged metamodel. The most important advantage of using Kermeta aspects is that the data required for expressing the execution semantics are not directly added to the Abstract Syntax. They are defined aside the DSA in an aspect that encapsulates the execution semantics. Furthermore, it allows the definition of several variations of the semantics for the same Abstract Syntax.

2.2 Definition of the Execution Functions

An Execution Function is equivalent to a method with a **specification** and an **implementation**.

2.2.1 Execution Function Specification

The specification states the **prototype** of the function (type and name of the parameters and return type). An Execution Function is defined in the context of an element of the AS or of the Execution Data metamodel. The context is considered as an implicit parameter (like **this** in Java or `self` in Python).

The specification also defines the **purpose** of the Execution Function. It is first informally defined as a text written in a natural language like English (like javadoc commentary in Java). This informal text can be formalized as **contracts** (pre-conditions and post-conditions). They formalize the assumptions made by the Execution Function designer both on the function parameters and on its expected results. If the expectations on parameters are not fulfilled, there are no guarantees that the Execution Function will behave as expected. It is a way for the xDSML designer to express constraints that must be fulfilled by the MoCC and thus to verify that the MoCC is compliant with the DSA assumptions.

These specifications focus on the correct use of each Execution Function independently from the others. There may also exist constraints on the order in which the Execution Functions can be called. This scheduling will be provided by the MoCC. If the same MoCC is used with different DSAs to specify different xDSMLs, or if the same DSA is used with different MoCC to specify different xDSMLs, it might be useful to assess that the MoCC will schedule the DSA in the appropriate order. This behavioral contract can be modeled with a UML protocol state machine.

2.2.2 Execution Function Implementation

The **implementation** is composed of statements which allow it to achieve its purpose. Implementation can access all its parameters. Nevertheless, it may only modify elements from the Execution Data, not from the AS.

2.2.3 Taxonomy of Execution Functions

Several kinds of Execution Functions have been identified (see Figure 2.1):

Modifier Modifiers are the main kind of Execution Functions. They are directly related to the operational semantics as they are called by the DSE level. To enforce a clear separation of MoCC and DSA, a Modifier cannot call another Modifier. As we are investigating the possibility to have macro-Modifiers (or composite DSE) which could call other EF.

Helper Helpers are auxiliary methods. They cannot be called from the DSE. They can only be called by other Execution Functions. Their purpose is only to factorize code that is useful to several Execution Functions. Thus, Helpers avoid duplicated code and favor evolution.

Query A Query is an Execution Function which is side-effects-free and computes a result. The set of queries define the RTD (Runtime Data) as defined in WP1, that is an abstract view of the ED and AS.

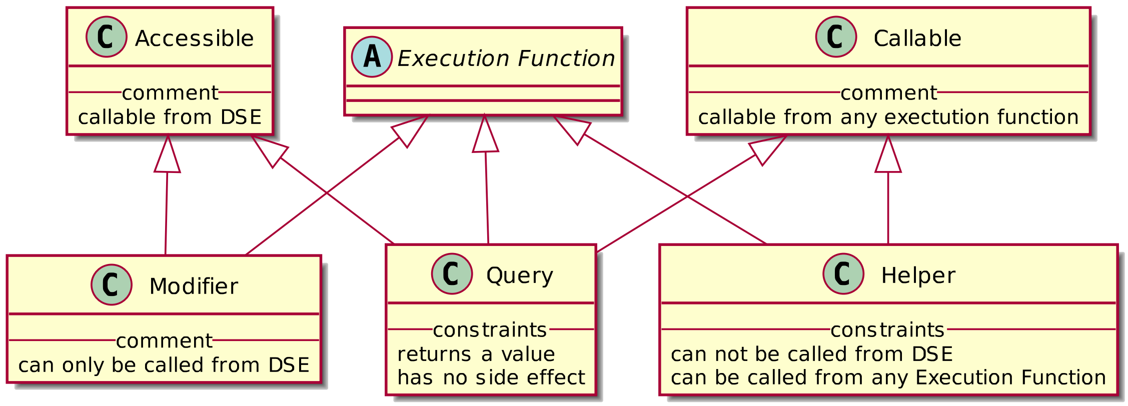

Figure 2.1: Taxonomy of Execution Functions

Annotations (or keywords) are explicitly used to define the nature of the Execution Functions. They are either Modifiers, Helpers or Queries. Using such annotations makes the intentions of the programmer explicit. Verifications can thus be performed based on these intentions. It could be possible to infer the nature of the Execution Functions by analyzing their body. For example, a Modifier is an Execution Function that is called by the DSE. Nevertheless, we advocate that it is a better solution to make the intentions of the xDSML designer explicit. Furthermore, it will favor the reuse of DSA because each of its Execution Functions is clearly typed and thus new DSE can reuse them more easily. Indeed, a DSE can only trigger Modifiers and Queries. It could be dangerous that a Helper is promoted as a Modifier by a new DSE specification. Indeed, Helpers are introduced to factorize code among Execution Functions are not supposed to be called directly from the DSE (it was not in the mind of the programmer and the xDSML designer should take such a responsibility). Furthermore, it will break the rule that states that a Modifier cannot call another Modifier as the Helper promoted as a Modifier is called by other Execution Functions.

2.2.4 Constraints on the Execution Functions

Here we list general constraints on the Execution Functions. These constraints come from WP1 (and more precisely from the D1.1.1 deliverable).

1. An Execution Function is an atomic operation.
2. An Execution Function does not take time to execute according to the execution engine (no duration). It is similar to listeners in a Swing program. Time (and related durations) will be managed either by the MoCC or will be reified in the DSA side (through specific Execution Data and Execution Functions).
3. An Execution Function does not deal with concurrency aspects (because that is the purpose of the MoCC). Thus constructs like thread creation or other constructions for concurrency are disallowed.
4. An Execution Function must only access its parameters (including the implicit one). Attributes and references of these parameters can be used to navigate in the model.
5. An Execution Function can only change Execution Data. AS elements are considered read-only. It is also forbidden to instantiate new elements from the abstract syntax.
6. A Modifier cannot call another *Modifier*.
7. An *Execution Function* may call a *Helper* or a *Query*.
8. A *Query* is side-effect free (it cannot modify Execution Data).

2.2.5 Implementation of Execution Functions: DSA action language

The DSA action language allows to define the specification of Execution Functions as well as their implementation.

2.2.5.1 Action Language Constructions

The action language provides:

1. local variable declaration
2. assignment (only on local variables and Execution Data, not on AS)
3. expressions and navigations over the model
4. control structures (choice, selection, conditional loop and unconditional loop)
5. function call

2.2.5.2 Termination of the Execution Functions

An Execution Function must terminate. The only constructs that have to be verified are the loop constructs. The unconditional loop (a foreach, that is an iterator, on a collection of an Ecore model) always terminates as all these collections are finished. The conditional loop is not guaranteed to finish. In this case, a *variant* must be defined. It is an integer value which is always positive and which strictly decreases at each loop step. Thus, unconditional loops have to be preferred over conditional loops as it is guarantee that they finish (as long as we check that they are always applied to collections from an Ecore model).

3. Formalization and Restriction of the Domain-Specific Events

The Execution Functions are triggered by the DSEs which are a sort of proxy or adapter between the DSA on one side and the MoCC or the environment of the xDSML on the other side. Let us suppose that everything on the strictly-DSA side of things has been designed so that conflicts may not arise. Let us see the constraints that must be applied to the DSEs.

3.1 Coincident Domain-Specific Events

When a step is chosen from the Solver of the MoCC, it contains several occurrences of instances of DSEs. Each of these DSE instances can trigger one or several Execution Functions. Therefore, several concurrent Execution Functions (modify and/or read the same data) may end up being executed at the same time. Thus, EF s triggered simultaneously must not have read/write conflicts on the same data. Then, we need to make sure that two DSE instances which affect the same data cannot be executed in the same step ; unless we want to specifically design a nondeterministic xDSML.

3.2 Coherence with the MoCC concerning the Feedback mechanism

The Feedback mechanism and its terminology is described in more details in Deliverable D1.3.1.

Essentially, we can consider that a MoCC represents the specification of all the possible future executions at the language level and is made up of MoccEvents and constraints on these MoccEvents. In that case, the influence of the ED on the MoCC Solver is done through the Feedback mechanism described in Deliverable D1.3.1. This mechanism requires MoccEvents which cannot have occurrences unless as a consequence of a Query's Feedback Policy. Furthermore, D1.3.1 suggests placing the Feedback Specification at the DSE level, which means that we may need some sort of verification relative to this mechanism.

Indeed, in order to implement the Feedback mechanism, we need:

1. Domain-Specific Events which trigger a Query whose return type is V upon occurrences of a MoccEvent m.
2. Some MoccEvents tagged as subjected to the Feedback mechanism.
3. Feedback relations in the MoCC between the MoccEvent triggering the Query m and the Tagged MoccEvents $t_1, t_2, ...$
4. A Feedback Policy whose rules map m with Feedback Consequences concerning $t_1, t_2, ...$

Therefore, the following verifications need to be made (where the relation Feedback(m, t_1, t_2, ...) in the MoCC means that m is mapped by a DSE to a Query and t_i are the Tagged MoccEvents affected by this Query):

1. At least one Domain-Specific Event triggers an Execution Function of type Query upon occurrences of m.

2. For each t_i, at least one Feedback Rule of the Feedback Policy of the Query triggered by the above-mentioned DSE has the allowance of t_i as Feedback Consequence.
3. The return type of the Query of the above-mentioned DSE must be the same type or a subtype of the type managed by the Feedback Filter of each Feedback Rule.

With these verifications, we make sure that what was specified in the MoCC as a Feedback relation between some MoccEvents is indeed taken into account in the DSE, and a Feedback specification is present.

4. Static Verification of Constraints

4.1 Motivation

Static verification is a mean to guarantee that a property will hold for any execution of an Execution Function. Furthermore, the xDSML designer is notified that a constraint fails before any execution of the language on a particular model. It avoids finding the right test (here the model and the scenario) that reveals the mistake. The designer can correct the mistakes and run again the static analysis tools.

4.2 Analysis of the implementation of the Execution Functions

Except the ones dealing with contracts or loop variants, the constraints on the DSA listed in the previous chapters can be checked by doing a static analysis of the Execution Functions' body.

As the action language we have chosen for implementing the DSA is Kermeta 3, it is possible to access the model of all the DSA. By analysing these models, we are able to check whether the properties listed in the previous sections hold or not. Simple analysis will be implemented directly. However in order to benefit from advanced analysis technologies, we also target a translation to the Why3 toolset[1]. This translation will rely on the OCL and block library action language translator to Why3 developped in the P/HiMoCo projects. Furthermore, it should be possible to integrate the results of those verifications in the GEMOC studio so that mistakes are signaled to the xDSML designer while he is writing the Execution Functions for the simple analysis, or on request for more complex ones relying on Why3.

4.2.1 Constraints intrinsic to the DSA

There is a first set of constraints to check that are intrinsic to the Execution Functions (and thus independent of the way the DSE will link the DSA to the MoCC). These constraints are listed in section 2.2.4 (constraints on the DSA).

4.2.2 Constraints induced by the DSE

Some properties are not only dependent on the DSA but also on the way they are used by the MoCC through the DSE. In particular, a DSE may choose to call simultaneously several Execution Functions. When these DSAs are known, it is possible to check that they are independent one from another: if one of the Execution Functions modifies an Execution Data, then no other DSA called simultaneously should be able to change or even read it. These verifications can be done at the language level or at the model level. It is possible to define the finger print of a DSA to know what parts of the Execution Data are read and what parts are written. The comparison of those finger prints would help in deciding whether the DSA are called consistently. This simple verification may be too constraining for the user. A more sophisticated ones might be implemented relying on external tools like Why3. These analysis might need to access the DSE and an abstraction of the MoCC.

[1] http://why3.lri.fr

4.3 Verification of formal contracts

The formalization of contracts defined as pre-conditions and post-conditions as well as the definition of invariants and variants on the AS and Execution Data can be used to verify the consistency of the DSA, internal consistency as well as consistency towards its environment (DSE and MoCC).

Internally, we verify whether the body of an Execution Function respects the contracts or not. That is, does the execution of the body establish the post-conditions and invariants assuming pre-conditions and invariants hold?

Another verification that could be performed is to check that the MoCC and the DSE respect the requirements of the DSA. In other words, are we sure that DSA will always be called correctly? This verification must take into account two aspects: on the one hand that the parameters provided by the DSE will always satisfy the pre-conditions of the DSA; and on the other hand that the MoCC will schedule the DSA through the DSE in the right order, that will ensure that the parameters of each DSA satisfy the pre-conditions. The first one can be assessed at the DSE level relying on the same kind of technologies as the DSA. The second one requires to approximate all the sequence of DSA that might be scheduled by the MoCC through the DSE and then to rely on the same kind of technologies as the DSA.

5. Dynamic Verification of Constraints

Contracts can be used to instrument the DSA code, that is, to add verification for the Execution Functions in order to check that pre-conditions are effectively fulfilled when they are called and that post-conditions are fulfilled when they terminate. It will then allow the xDSML user to detect errors at runtime on a given execution.

This approach looks like testing: it consists in checking expected properties during the execution of the program. It will not guarantee that the MoCC and DSA are compliant but only detect defects while using the GEMOC studio to execute a given model.

Nevertheless, it is a useful technique as it allows one to detect defects near their originating point. Errors are thus easier to locate and then correct. It will help the xDSML designer in the task of validating his language. Furthermore, it does not require to express all the contracts to be useful. It is recognized at the code level, that the expression of natural post-conditions are generally sufficient to detects most programming errors. There are good chances that this rule also applies to the contracts of DSA.

On the implementation side, pre-conditions and post-conditions were part of previous versions of the Kermeta language. If these aspects are still present in Kermeta 3, it would be easy to perform runtime verification of contracts.

6. Conclusion

This deliverable focuses on the design of DSA side of an xDSML. The DSA of an xDSML are composed of Execution Data and Execution Functions which are written in a specific action language (in our implementation: Kermeta 3). This deliverable targets the description of the expected constructs of such an action language and the definition of restrictions to apply on it when used to implement the DSA of an xDSML following the GEMOC approach.

We have identified both the expected constructions of the DSA language and the restrictions that must be applied on the implementation action language. Verifications can be done on the statements of the action language but they can also exploit contracts. Most of these verifications can be performed statically. As an example, restrictions on the action language can be done by analyzing its body. Nevertheless, verifications dealing with contracts may be too difficult both for the xDSML designer (it is hard to be exhaustive in the definition of contracts) and for the verification tools used. Thus, it is certainly a good help to also investigate the possibility to perform the verifications at runtime. Those verifications may help in detecting defects in the xDSML under construction.

7. References

[1] 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.