Grant ANR-12-INSE-0011

ANR INS GEMOC

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

Task 3.3

Version 1.0

DOCUMENT CONTROL

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

Revision index	Modifications
–	version 1.0 — First version
A	
B	
C	
D	

Authors

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

Contents

1. Introduction

1.1 Purpose

This document establishes the formal link between the language for MoCC definition and the domain specific concern in a specific 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, as it must define the minimum relevant numbers of concepts which would give in our sense an operational semantics, but also at the implementation level, as it must provide a formal semantics, respectful of that specification, to a reasonable subset of Kermeta, 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 xDSML and a modeling workbench to build and execute models of these xDSML. 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, DSA, is broadly defined as the set of attributes, references and classes needed to represent the runtime state of the model and atomic operations that modify these elements. It is an extension of the abstract syntax (AS) and dependant of the domain addressed by the DSML. The second one, defined as MoCC, is the scheduling policy of the above-mentioned operations. It handles the concurrent aspects of the xDSML.

To map MoCC on DSA and to favor their reuse, a third section has been added as the Domain Specific Events (DSE). DSE are triggered either by the MoCC or by external interacting models (Behavioral Model Composition Operators). When triggered, they call the corresponding DSAs and feedback results to the MoCC. The overall architecture and the methodology are described in the WP1 (see D3.3.1 document). MoCC aspects are further described in the WP2 and composition operators in WP3.

In this deliverable, we focus on two aspects. The first one concerns 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. The second one deals with the specification of the correct use of the DSA by the MoCC.

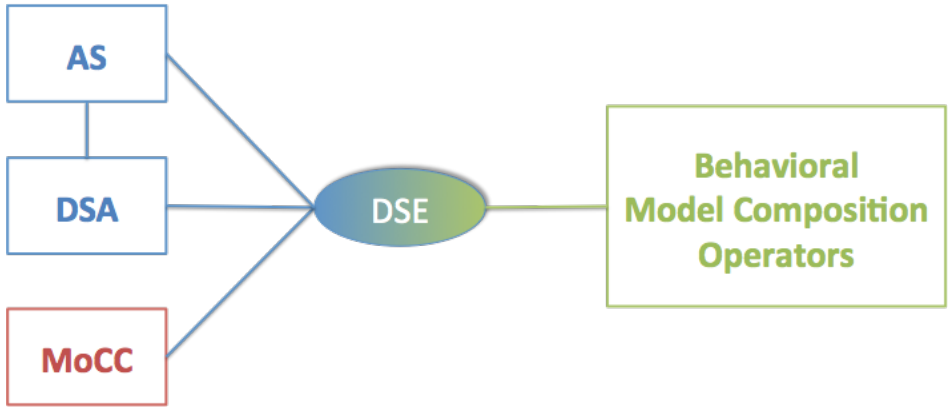

Figure 1.1: An eXecutable Modeling Language "a la" GEMOC

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 perimeter and the goals of this document. Chapter 2 presents the constraints on the DSA part of an xDSML (including execution data and execution functions) both of the constructions of the language and their use to define execution functions. Chapter 3 lists the constraints on DSA that are implied by the use of a specific DSE. Chapters 4 and 5 explains how the requirements listed in the previous chapters can be verified either by static analysis tools (before any execution of xDSML model) or dynamically (at runtime, when a test model is executed).

2. DSA

The DSA part of a language aims to define the operational semantics of an xDSML without dealing with 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. This runtime data is called execution data. The second part is composed of execution functions which are primitive actions of the execution semantics that manage execution data.

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.

2.1 Execution data definition

Execution data is data that has to be managed at runtime to execute a model. Execution data is defined as an Ecore metamodel (with possibly OCL constraints). This Ecore metamodel is generally an extension of the AS as runtime data is often attached to elements of the AS. Elements of the AS are not 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 DSML 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 execution function in an aspect that encapsulates the execution semantics. Furthermore, it allows to define several semantics for the same abstract syntax.

2.2 Execution functions

An execution function is equivalent to a method with a **specification** and an **implementation**.

The specification states the **prototype** of the function (parameters and return value). Execution functions are 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).

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** (preconditions and postconditions). 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 guarantee 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 an UML protocol state machine.

The **implementation** is composed by statements which allow to achieve the purpose of the function. Implementation can access all its parameters. Nevertheless, it can only change elements of the execution data, not the AS ones.

2.2.1 Taxonomy of execution functions

Several kinds of execution functions have been identified (figure 2.1):

action Actions 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, an action cannot call another action.

helper Helpers are auxiliary methods. They cannot be called from the DSE. They can only be called by execution functions. Their purpose is only to factorize code that is useful to several execution functions. Thus, helpers avoid to have duplicated code that favor evolution.

query A query is an execution function which is side-effect 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 execution data and AS.

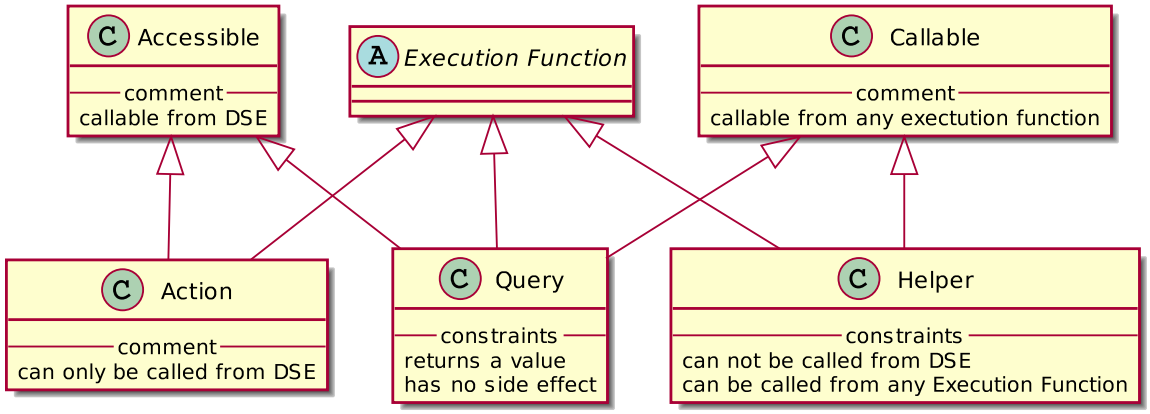

Figure 2.1: Taxonomy of execution functions

Annotations (or keywords) are explicitly used to define the nature of the DSA. They are either actions, 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 function by analysing its body and the DSE part of the xDSML. For example, an action is an execution that is called by the DSE. Nevertheless, we advocate that it is a better solution to make explicit the intention of the xDSML designer. Furthermore, it will favor reuse of DSA because each of its execution functions is clearly typed and thus new DSE can reuse them more easily. Indeed, a DSE must only use actions and queries. It could be dangerous that an helper is promoted as an action by a new DSE. Indeed, helpers are introduced to factorize code among execution functions are not supposed to be called directly for 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 an action cannot call another action as the helper promoted as an action is called by other actions.

2.2.2 General Constraints

Here we list general constraints on execution functions. These constraints come from the WP1 work package (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 like listeners in a Swing program. Time (and related durations) will be managed either by the MoCC or will be reified at 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. An execution function cannot call an *action*.
7. An *execution function* may call an *helper* or a *query*.
8. A *query* is side-effect free (it cannot modify execution data).

2.2.3 Implementation of execution functions: DSA action language

The DSA action language allows to define the specification of execution functions as well as their implementation.

2.2.3.1 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. call of function

2.2.3.2 Termination of execution function

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. DSE

The DSA are called via the DSE which are a kind of proxy or adapter between the DSA one one side and the MoCC or the environment of the xDSML on the other side. When a DSE event occurs, one or several actions are called. Furthermore, several DSE may occur at the same simulation step. It implies that different actions may be called simultaneously or in an arbitrary order. If the called actions have side effects on the same elements of the execution data, it implies that the execution is non deterministic (except if the new values set during the calls are all the same for all the actions).

The action implementation, and thus the helper implementation they use, have to be checked in order to determine if two actions can be called simultaneously. It mainly consists in checking that there is no data that is written by an action while it is written or read by other actions.

4. Static Verification of Constraints

4.1 Motivation

Static verification is a mean to guarantee that a property will hold for any execution of the 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 static analysis tools.

4.2 Analysis of the implementation of DSA

Except the one dealing with contracts or loop variants, the constraints on DSA listed in the previous sections can be checked by doing a static analysis of the execution functions body.

As the action language is Kermeta, it is possible to access the model of all the DSA. By analysing these models, we will be able to check whether the properties listed in the previous sections holds 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 result 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 DSA

There is a first set of constraints to check that are intrinsic to execution functions (and thus independent of the way the DSE will link the DSA to the MoCC or the environment). These constraints are listed in section 2.2.2 (constraints on the DSA).

4.2.2 Constraints induced by the DSE

Some properties are not only dependent of the DSA but of the way these ones are used by the MoCC through the DSE. In particular, DSE may choose to call simultaneously several DSA. When these DSA are known, it is possible to check that they are independent one from the others: if a DSE changes an element of the execution data, then no other DSA called simultaneously should 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.

4.3 Verification of formal contracts

The formalization of contracts defined as preconditions and postconditions 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. That is, does the execution of the body establishes the postconditions and invariants assuming preconditions 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 preconditions 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 preconditions. The first one can be assessed at the DSE level relying on the same kind of technologies than 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 than the DSA.

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

5. Dynamic Verification of Constraints

Contracts can be used to instrument the DSA code, that is to add verification in execution functions to check that preconditions are effectively fulfilled when they are called and that postconditions are fulfilled when they return. It will then allow 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 to detect defects near to their seed. Errors are thus easier to locate and then to 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 postconditions are generally sufficient to detects most programming errors. There is good chances that this rule also applies to the contracts of DSA.

On the implementation side, preconditions and postconditions where part of previous implementations 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 DSA side of an xDSML. DSA are composed of execution functions which are written in a specific action language (Kermeta3) and execution data. 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 DSA.

In this first version of the deliverable, we have identified both the expected constructions of the DSA language and the restrictions that must apply on the implementation action language. Verifications can be done on the statements of the action language but they can also exploit the contracts. Most of these verifications can be performed statically. As an example, restrictions on the action language can be done by analysing 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 used verification tools. 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.