HAL
open science

A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio

Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, Cédric Brun, Thomas Degueule, Didier Vojtisek

To cite this version:

Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, et al.. A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio. 2015. hal-01152342v1

HAL Id: hal-01152342

https://inria.hal.science/hal-01152342v1

Preprint submitted on 15 May 2015 (v1), last revised 2 Nov 2015 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio

Benoit Combemale
INRIA
benoit.combemale@inria.fr

Julien DeAntoni
UNS, I3S/INRIA
julien.deantoni@polytech.unice.fr

Olivier Barais
University of Rennes 1, IRISA
olivier.barais@irisa.fr

Arnaud Blouin
INSA Rennes, IRISA
arnaud.blouin@irisa.fr

Erwan Bousse
University of Rennes 1, IRISA
erwan.bousse@irisa.fr

Cédric Brun
OBEO
cedric.brun@obeo.fr

Thomas Degueule
INRIA
thomas.degueule@inria.fr

Didier Vojtisek
INRIA
didier.vojtisek@inria.fr

We present in this paper a complete solution to the Model Execution case of the Transformation Tool Contest 2015 using *the GEMOC Studio*. The solution proposes an implementation of the most complete version (variant 3) of the UML Activity Diagram language. The implementation uses different technologies integrated into the GEMOC Studio for implementing the various concerns of the language: *Kermeta* is used to modularly implement the operational semantics and to weave it into the provided metamodel, *Melange* is used to build the overall language runtime seamlessly integrated to EMF, *Sirius Animator* is used to develop a graphical animator, *the GEMOC execution engine* is used to execute the conforming models according to the operational semantics and to build a rich and efficient execution trace that can be manipulated through a powerful *timeline*, which provides common facilities like, for instance trace visualization, and step-by-step execution (incl., breakpoint, step forward and step backward). Finally, *MoCCML* is used to provide an alternative to the implementation with an explicit and formal concurrency model for activity diagrams supported by a solver and analysis tools. We evaluate our implementation with regard to the evaluation criteria provided into the case description and provide evidence of the correctness, understandability, conciseness and performance of our solution.

1 Introduction

Executability of models opens many possibilities in terms of early dynamic verification and validation (V&V) of models, such as debugging [4], model checking [2] and runtime verification [7]. In recent years, a lot of efforts have been made to provide facilities to design executable Domain-Specific Modeling Languages (xDSMLs) [3, 8, 10]. To establish an overview of the state of the art in terms of tools and methods to design and implement xDSMLs, the Transformation Tool Contest (TTC) 2015[1] has proposed a dedicated case about Model Execution [9]. This case describes a part of the execution semantics for the UML Activity Diagram language in the form of an operational semantics.

In this paper, we present a solution to the most complete variant of this case (*i.e.,* variant 3) using the GEMOC studio[2]. The variant 3 of the Model Execution case considers the complete Activity Diagram

[1]Cf. http://www.transformation-tool-contest.eu/
[2]Cf. http://gemoc.org/studio

Submitted to:
Transformation Tool Contest 2015

© Combemale et al.
This work is licensed under the
Creative Commons Attribution License.

metamodel provided. It includes various kinds of nodes (initial node, final node, fork node, join node, decision node, merge node), opaque actions, boolean and integer variables (either local or as input), and various boolean and integer expression types.

Our solution uses the GEMOC Studio, an eclipse package atop the *Eclipse Modeling Framework* (EMF) [3], which includes both a language workbench to design and implement tool-supported xDSMLs, and a modeling workbench where the xDSMLs are automatically deployed to allow system designers to edit, execute, simulate, and animate their models. As a result, our solution not only provides a model interpreter conforming to the proposed operational semantics of the UML Activity Diagram language, but also provides a graphical model animator, an advanced trace manager, as well as an alternative version that offers an explicit and formal model of computation supporting concurrency. All ressources are available from `http://gemoc.org/ttc15`.

For designing and implementing the various concerns of an xDSML, the language workbench put together the following tools seamlessly integrated to EMF:

- *Kermeta*, which offers specific annotations for Xtend[4] to support the modular implementation of an operational semantics (both runtime concepts and steps of computation) and its weaving into an EMF-based metamodel (*i.e.,* an Ecore model).
- *Melange*[5], to build the overall language runtime seamlessly integrated to EMF and to ensure interoperability between the legacy metamodel without the operational semantics, and the metamodel extended with the operational semantics.
- *Sirius Animator*, an extension of the model editor designer Sirius[6] to create graphical animators for xDSMLs.
- *MoCCML*, a tool-supported meta-language to specify a Model of Concurrency and Communication (MoCC) and its mapping to a specific metamodel and associated operational semantics of a xDSML.

The language workbench also includes a generative approach, which provides a rich and efficient domain-specific trace metamodel for any xDSMLs (for more details, we refer the reader to [1]).

Once an xDSML is implemented with the aforementioned tools of the language workbench, the xDSML is automatically deployed into the modeling workbench, which provides to system engineers an advanced environment integrated to the Eclipse debugger for model execution. In particular, the modeling workbench provides the following tools:

- A Java-based *execution engine* (parameterized with the specification of the operational semantics), possibly coupled with *TimeSquare*[7] (parameterized with the MoCC), to support the concurrent execution and analysis of any conforming models.
- A *model animator* parameterized by the graphical representation defined with Sirius Animator to animate executable models.
- A generic *trace manager*, which allows a system designer to visualize, save, replay, and explore different execution traces of their models, as well as navigating step-by-step in a given execution trace (incl., breakpoint, step forward and step backward).

[3] Cf. `https://www.eclipse.org/modeling/emf/`
[4] Cf. `https://eclipse.org/xtend/`
[5] Cf. `http://melange-lang.org`
[6] Cf. `https://eclipse.org/sirius/`
[7] Cf. `http://timesquare.inria.fr`

- A generic *event manager*, which provides a user interface for injecting external stimuli in the form of events during the simulation (*e.g.,* to simulate the environment).

In the rest of this paper we first describe in Section 2 the use of the GEMOC language workbench to design and implement the UML Activity Diagram language as defined in the variant 3 of the Model Execution case, and then show in Section 3 the result in the GEMOC modeling workbench. Finally, we evaluate in Section 4 our implementation with regard to the evaluation criteria provided into the case description and provide evidence of the correctness, understandability, conciseness and performance of our solution. Section 5 concludes and gives some perspectives for the GEMOC Studio.

2 Description of the solution

In this section we describe the design and implementation of the UML Activity Diagram language as defined in the variant 3 of the TTC'15 Model Execution case [9].

2.1 Operational Semantics

Kermeta is used to implement the operational semantics. Kermeta complements Xtend to support the definition of both the runtime concepts and the steps of computation in a separate file than the initial metamodel (reused as is), and to statically weave them in the initial metamodel. Kermeta provides static typing to safely define the operational semantics, and a compilation scheme of the operational semantics which results in a Java-based runtime seamlessly integrated to the Java code generated by EMF from the initial metamodel.

The runtime concepts can be additional classes that will be merged with the initial metamodel, or new structural features (attributes or references) either in the existing classes of the initial metamodel or in the new added classes. When new structural features have to be added to a class existing in the initial metamodel, the annotation @ASPECT is used to re-open the class.

Listing 1 shows an excerpt of the modular definition of the runtime concepts. *Token* and *ForkToken* are new concepts, while the content of *ActivityNodeAspect*, a collection of *Token*, will be merged into the concept *ActivityNode* from the abstract syntax (cf. annotation @*Aspect*). All the runtime concepts of the case have been defined similarly.

```
@Aspect(className=ActivityNode)
class ActivityNodeAspect {
    List<Token> heldTokens = new ArrayList<Token>
}

abstract class Token {
    public ActivityNode holder
}

class ForkedToken extends Token {
    public Token baseToken ;
    public Integer remainingOffersCount;
}

[...]
```

Listing 1: Modular definition with Kermeta of the runtime concepts

The steps of computation are defined in terms of operations weaved into the suitable classes, either from the initial metamodel or from the new added classes of the runtime concepts. Similarly to the structural features of the runtime concepts, when an operation has to be added to a class existing in the initial metamodel, the annotation @ASPECT is used to re-open the class.

Listing 2 shows an excerpt of the definition of the steps of computation (i.e., the interpreter), which manipulate the runtime concepts previously defined. The implementation follows the design pattern *Interpreter*[8], defining one operation *execute* per concept of the abstract syntax to be interpreted. Each method is modularly defined into an aspect, and then weaved into the suitable class of the abstract syntax. Listing 2 shows the overall execution of an *Activity*. All the steps of computation of the most complete variant of the case have been defined similarly.

```
1  @Aspect(className=Activity)
2  class ActivityAspect {
3
4      def void execute(Context c) {
5          _self.locals.forEach[v|v.init(c)]
6          _self.nodes.filter[node|node instanceof InitialNode].get(0).execute(c)
7
8          var list =  _self.nodes.filter[node|node.hasOffers]
9          while (list!=null && list.size>0 ){
10             list.get(0).execute(c)
11             list =  _self.nodes.filter[node|node.hasOffers]
12         }
13     }
14 }
15
16 [...]
```

Listing 2: Modular definition with Kermeta of the steps of computation

The definition of the runtime concepts and the steps of computation in a separate file offers a modular mechanism to implement the operational semantics. In addition to support the separation of concerns (abstract syntax and operational semantics), this is also a way to support different implementations of the operational semantics for the same abstract syntax (*e.g.,* in case of semantic variation points).

2.2 Language Assembling

Once the operational semantics is defined with Kermeta, *Melange* can be used from the language workbench for assembling the initial metamodel and the chosen operational semantics into an xDSML.

```
1 language UMLActivityDiagram {
2     syntax "platform:/resource/.../activitydiagram.ecore"
3     with org.gemoc.ad.sequential.dynamic.*
4     exactType UMLActivityDiagramMT
5 }
```

Listing 3: Assembling an xDSML with Melange

As a result, Melange provides the xDSML as well as a structural interface (*aka.* model type [6]) in the form of a new metamodel that can be used to define additional tooling such as model transformations (*e.g.,* trace manager and execution engine) or animators, which use the operation semantics (runtime concepts or steps of computation). In addition to provide the assembling of the expected xDSML as well

[8]Cf. http://en.wikipedia.org/wiki/Interpreter_pattern

as the interoperability between the initial metamodel and the metamodel with the operational semantics, Melange also provides other features not required in this solution such as language inheritance and model transformation reuse.

2.3 Trace Management

Based on the resulting xDSML, the language workbench includes a generative approach that automatically provides a rich and efficient domain-specific trace metamodel. Instead of relying on complete snapshots of the executed model to construct a trace, this metamodel precisely captures what is the execution state of a model conforming to the xDSML through an efficient object-oriented structure based on the runtime concepts of the xDSML. In addition, the structure provides rich navigation facilities to browse a trace according to various dimensions (*e.g.* the value of a field or the occurrences of an event). For more details we refer the reader to [1].

2.4 Animation Facilities

Optionally, *Sirius Animator* can be used to complement the xDSML with a graphical model animator. *Sirius Animator* allows to either extend the graphical representation of an existing model editor defined with Sirius, or to define a separate graphical representation, based on the runtime concepts. This graphical representation is then be used to visualize the state of a model during its execution.

In our solution, we defined a new graphical representation (called *viewpoint specification* in Sirius) on top of the provided metamodel for UML activity diagrams, augmented with the runtime concepts to be visualized at runtime.

2.5 Explicit and Formal Concurrency Model

Because concurrency is a more and more important concept, one can use *MoCCML* to specify the MoCC. A MoCC specifies in a formal way the, possibly timed, causalities and synchronizations among the steps of computation. Based on *MoCCML*, non determinism and parallelism is clearly and formally identified in the operational semantics and can be varied or refined [5]. Analysis tools are also provided in the GEMOC studio to make analysis of the MoCC.

Listing 4 shows an excerpt of the MoCC specification. Line 1 and 2, it defines an event in the context of an *ActivityNode* (*i.e.*, for all its instances). For each occurrence of this event the *execute* function is called. All these events are constraints by some relations. For instance, in the classical case, the execution of a node is done after its predecessor has been executed (see the *Precedes* relation line 6). In the context of *Activity* appears a kind of loop since the activity can not start if not stop (line 11) and its *start* actually executes the initial node of the activity (line 15), *i.e.*, the starting point of the causality chain written line 6. From such specification, and for a specific model, a symbolic event structure is automatically derived.

```
context ActivityNode
  def : executeIt : Event = self.execute()
  inv waitControlToExecute:
        (not self.oclIsKindOf(MergeNode)) implies
        Relation Precedes(self.incoming.source.executeIt, self.executeIt))

context Activity
  def : start : Event = self.initialize()
  def : finish : Event = self.finish()
  inv NonReentrant:
      Relation Alternates(self.start, self.finish)

context InitialNode
  inv startedWhenActivityStart:
      Relation Precedes(self.activity.start, self.executeIt  )
```

Listing 4: Excerpt of the explicit and formal model of concurrency for activity diagrams

3 Result

The implementation of the UML Activity Diagram language described in the previous section is automatically deployed in the GEMOC modeling workbench (see Fig. 1).

The modeling workbench offers a powerful environment to system engineers for controlling the execution of their models with a debugger-like control panel[9] (①), visualizing the execution of their models thanks to the graphical animator (②), and analyzing and exploring several execution traces with a graphical timeline that supports step forward and step backward (③). Finally, the modeling workbench offers several extension points that can be used to plug additional front-end or back-end, such as a timing diagram included into the modeling workbench.

4 Evaluation of the solution

In this section, we evaluate our solution by using the evaluation criteria proposed in the case description [9]. Each criteria is evaluated on three different versions of our solutions, all implemented within the GEMOC studio:

- *executionOnly*: interpreter defined with Kermeta only (incl., Section 2.1);
- *withAnimationAndTrace*: execution within the GEMOC modeling workbench, with support of the animation and of the trace management (incl., Sections 2.1, 2.2, 2.3 and 2.4);
- *withConcurrency*: execution within the GEMOC modeling workbench, with support of the animation, the trace management and the concurrency analysis (incl., Sections 2.1, 2.2, 2.3, 2.4 and 2.5).

[9]Note that when using MoCCML, the concurrent steps of computation are indicated in the control panel (the steps of computations that will be executed concurrently during a given execution step). If the MoCC is non deterministic, the control panel proposes the different permitted execution steps, one of which can be selected either manually by the system engineer, or automatically thanks to one of the proposed heuristic.

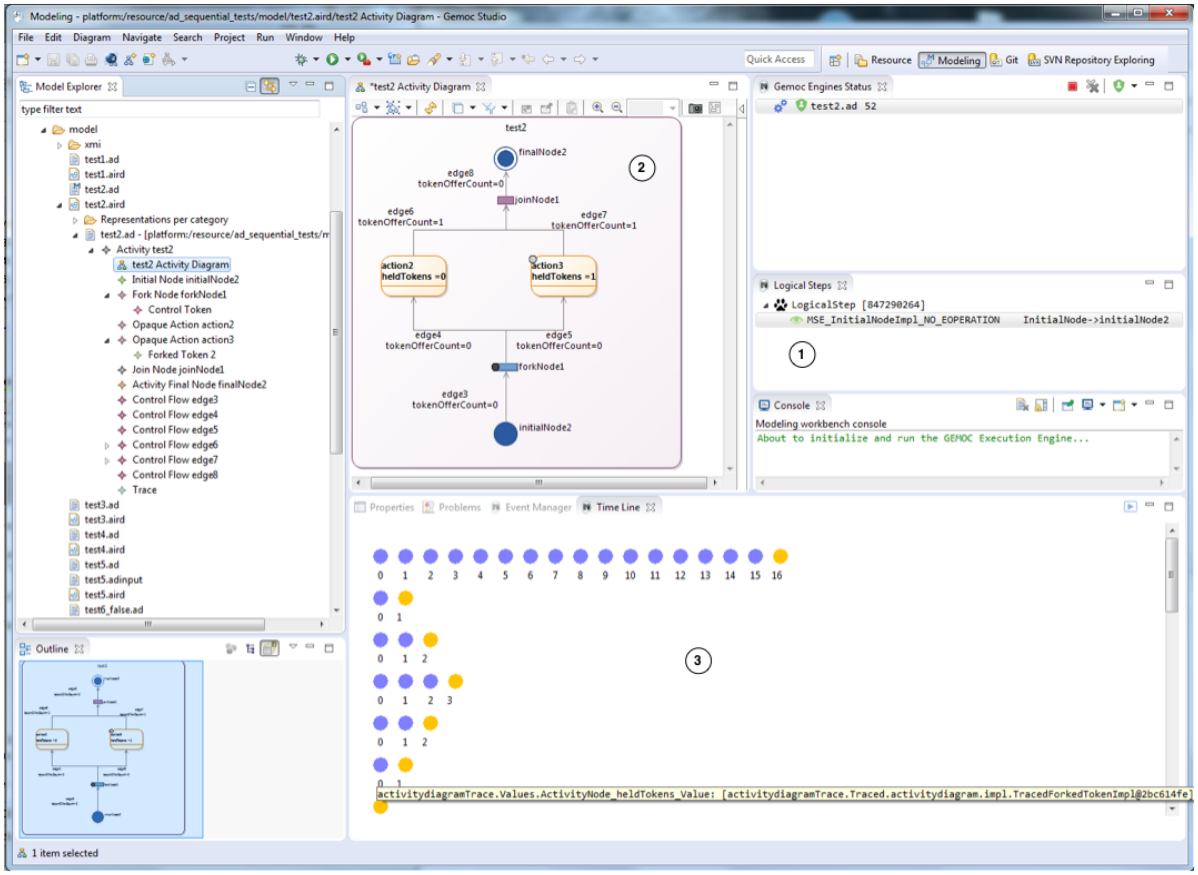

Figure 1: The GEMOC Modeling Workbench for the TTC'15 Activity Diagram Language

4.1 Correctness

The correctness of our solution is based on the test suites provided by the case. All the three versions of our solutions provide correct results.

4.2 Understandability and Conciseness

Kermeta is used to design and implement the operational semantics. Based on Xtend, Kermeta provide a powerful Java-like imperative and statically typed meta-language. The meta-language follows an object-oriented paradigm which make it directly aligned with the object-oriented Ecore metamodel provided by the case.

The implementation of the operational semantics follows the well-known and time-honored *Interpreter* design pattern which support a modular design of the operational semantics with regard to the initial metamodel which is reused as is and not affected. There is no translation into a third formalism, and this approach easily supports the definition of different variants of the semantics (*e.g.,* interpreter and compiler, different semantic variation points, etc.). Finally, the use of the open-class and static introduction mechanisms make the design of the operational semantics even simpler than the interpreter

pattern, avoiding to duplicate the initial structure into the interpreter. The operations (execute) are directly weaved into the suitable classes of the initial metamodel.

The entire implementation of the operational semantics of the variant 3 of the Model Execution case comprise 441 LOC (version *executionOnly* of our solution). This comprises the entire implementation of the interpreter, sufficient for the execution of any conforming models. The other technologies Melange, Sirius Animator and MoCCML are optional, and can be used only to provide the additional features such as model interoperability, trace management, model animation and formal concurrency management.

Note also that the analysis tools that provide the modeling workbench are not only useful for the system engineer to analyse the models, but also for the language designer to analyse the language semantics implementation.

4.3 Performance

We report in Table 1 the execution time (without load and save times) of the models provided for the performance evaluation (*Test perf 1*, *Test perf 2*, *Test perf 3_1* and *Test perf 3_2*). The execution time is provided for all the models, according to the three versions of our solution (*executionOnly*, *withAnimationAndTrace* and *withConcurrency*)[10].

	t(executionOnly)	t(withAnimationAndTrace)	t(withConcurrency)
Test perf 1	0.29	0.87	226183
Test perf 2	0.33	0.78	$\perp$
Test perf 3_1	0.37	1,01	$\perp$
Test perf 3_2	0.13	0,19	5219

Table 1: Execution time (in ms) of the performance tests

5 Conclusion and Perspective

We present in this paper our solution using the GEMOC studio to the most complete *variant 3* of the TTC'15 Model Execution case. The solution provides not only an EMF-based interpreter for UML activity diagrams, but also comes with a well integrated model debugging environment based on Eclipse, including advanced features for graphical model animation and execution trace management. We also propose an enhanced version of our solution which integrate into the operational semantics a formal and explicit model of concurrency supported by analysis tools. The GEMOC studio integrates different technologies to implement the various concern of the executability (runtime concepts, steps of computation, animator, concurrency). We evaluate our solution regarding both the benchmark provided into the case, and the criteria proposed into the case description. In particular, we give evidence for the correctness, the understandability and conciseness, and the performance of our solution.

The GEMOC studio is a play ground for research activities related to *Software Language Engineering*, including model executability. Various studies are currently investigated on related topics, including the integration with continuous time, formal analysis, optimizing compilers, semantic variability and adaptation, and application to other domains (*e.g.,* enterprise architecture and scientific modeling).

[10] In *Test perf 2* and *Test perf 3_1*, there are more then 1 billiard of billiard of billiard (*i.e.*, 2^{100}) of possible inter-leavings. The goal of the concurrent version is to show and made explicit such interleaving but in these cases, it is obviously impossible to do so.

References

[1] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale & Benoit Baudry (2015): *A Generative Approach to Define Rich Domain-Specific Trace Metamodels*. In: *ECMFA*, LNCS, Springer.

[2] Benoit Combemale, Xavier Crégut, Pierre-Loïc Garoche & Xavier Thirioux (2009): *Essay on Semantics Definition in MDE, An Instrumented Approach for Model Verification*. Journal of Software 4(9), pp. 943–958.

[3] Benoit Combemale, Xavier Crégut & Marc Pantel (2012): *A Design Pattern to Build Executable DSMLs and Associated V&V Tools*. In: *19th Asia-Pacific Software Engineering Conference*, IEEE, pp. 282–287.

[4] Jonathan Corley, Brian P. Eddy & Jeff Gray (2014): *Towards Efficient and Scalabale Omniscient Debugging for Model Transformations*. In: *14th Workshop on Domain-Specific Modeling*, ACM, pp. 13–18.

[5] Julien Deantoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau & Benoit Combemale (2015): *Towards a Meta-Language for the Concurrency Concern in DSLs*. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE).*

[6] Clément Guy, Benoît Combemale, Steven Derrien, Jim RH Steel & Jean-Marc Jézéquel (2012): *On model subtyping*. In: *ECMFA*, LNCS, Springer, pp. 400–415.

[7] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. The Journal of Logic and Algebraic Programming 78(5), pp. 293–303.

[8] Tanja Mayerhofer, Philip Langer, Manuel Wimmer & Gerti Kappel (2013): *xMOF: Executable DSMLs based on fUML*. In: *6th Int. Conf. on Software Language Engineering*, *LNCS* 8225, Springer, pp. 56–75.

[9] Tanja Mayerhofer & Manuel Wimmer (2015): *The TTC 2015 Model Execution Case*. In: *8th Transformation Tool Contest*, CEUR.

[10] Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard & François Terrier (2014): *Formalizing Execution Semantics of UML Profiles with fUML Models*. In: *17th International Conference on Model Driven Engineering Languages and Systems*, *LNCS* 8767, Springer, pp. 133–148.