Grant ANR-12-INSE-0011

GEMOC
ANR Project, Program INS

D4.4.1 – API for Trace Management (report and metamodel)

Task 4.4

Version 2.0

Document Control

	- : 30/10/2014	A :	B :29/04/2015	C :	D :
Written by Signature	François Tanguy	Julien Deantoni	Didier Vojtisek		
Approved by Signature					

Revision index	Modifications
A	
B	Updated to taken into account the branch support in the metamodel
C	
D	

Authors

Author	Partner	Role
Didier Vojtisek	INRIA	Lead author
François Tanguy	INRIA	Contributor
Benoît Combemale	INRIA	Contributor
Julien De Antoni	I3S	Contributor

Table of Content

API for Trace Management

1. Introduction

This document aims to describe the Application Progr amming Interface (API) for the trace management in the context of the ANR INS project GEMOC and is part of a collection of documents describing the GEMOC Studio which is a platform for:

- designing executable Domain Specific Modeling Languages (xDSMLs),
- executing models conforming to xDSMLs.

1.1 Purpose

Tracing program execution is a common requirement in software engineering and the same requirement applies to model execution. The execution trace metamodel is included into the GEMOC Studio in order to capture information about the execution of models. These information also known as traces are used to better understand what happened and when it happened. Trace management is also used to drive the execution by offering the ability to step forward and backward.

1.2 Perimeter

In this document the API for trace management is described as well as its integration in the GEMOC Studio. In particular, a tool and a user interface called Timeline have been implemented. The document also presents some perspectives in regards of future development based on the API as well as some limitations of the current implementation.

1.3 Definitions, Acronyms and Abbreviations

DSML: Domain Specific Modeling Language.

xDSML: Executable Domain Specific Modeling Language.

GEMOC Studio: Eclipse-based studio integrating both a language workbench and the corresponding modeling workbenches.

MoCC: Model of Computation.

DSE: Domain-Specific Event.

MSE: Model-Specific Event, an instance of a Domain-Specific Event. While a DSE is specific to a language, an MSE is specific to a model conforming to a language.

Solver: software component given a MOC capable of scheduling execution.

Language Workbench: a language workbench offers the facilities for designing and implementing modeling languages.

Language Engineer: a language engineer is the user of the language workbench.

Modeling Workbench: a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.

System Engineer: user of the modeling workbench.

DSA: Domain-Specific Action.

1.4 References

Deliverable 4.1.1 describes the architecture of GEMOC Studio.
Deliverable 4.2.1 offers a detailed description of the execution engine.
Deliverable 1.1.1 presents the methodology of writing a xDSML.
Deliverable 1.3.1 details the methodology of writing a MoCC in CCSL.

1.5 Summary

Section 1 provides an introduction to this document. In order to understand how traces are created, Section 2 gives a quick overview of the execution engine design. Section 3 describes the use cases related to execution trace management. Section 4 describes the execution trace metamodel. Its integration in the GEMOC Studio is explained in Section 5. Finally some perspectives and current limitations are presented in Section 6.

2. Execution Engine Design

This section presents a quick overview of the GEMOC execution engine. Only the important concepts for trace management are explained. For more detailed information, please refer to the deliverables 4.2.1 and 4.1.1.

Before diving in the concepts of the engine, it is necessary to introduce the concept of possible versus chosen logical step:

- Due to possible non determinism, there are potentially different futures at a specific point in the execution. A *possible logical step* is one of these futures represented by a set of simultaneous events.
- We refer to *chosen logical step* when a possible logical step has been elicited and executed.

The following UML class diagram shows three important concepts. The class `ExecutionEngine` is responsible for coordinating the execution. In particular it will split the execution into execution steps. The class `Solver` proposes some possible logical steps at each execution step. The class `Decider` is a strategy whose the responsibility is to pick one particular logical step to be executed among those proposed by the Solver.

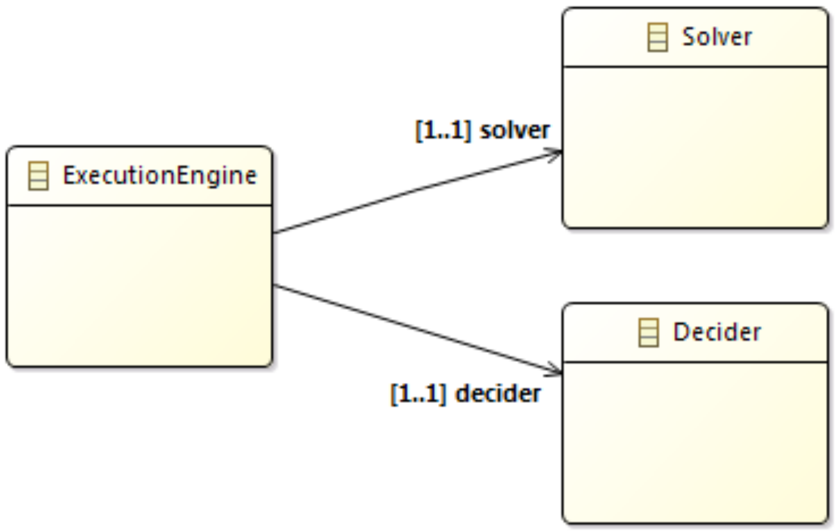


Figure 1 : Class Diagram of GEMOC Execution Engine

The following sequence diagram gives a more dynamic view of the interactions between the three components during an execution step. The execution engine is driving the execution. At first, it asks the solver to find possible logical steps (named *propositions* on Figure 2). The execution engine asks the decider to choose one logical step according to a specific strategy (e.g., random). Finally, the engine executes the logical step. This sequence is repeated during all the model execution.

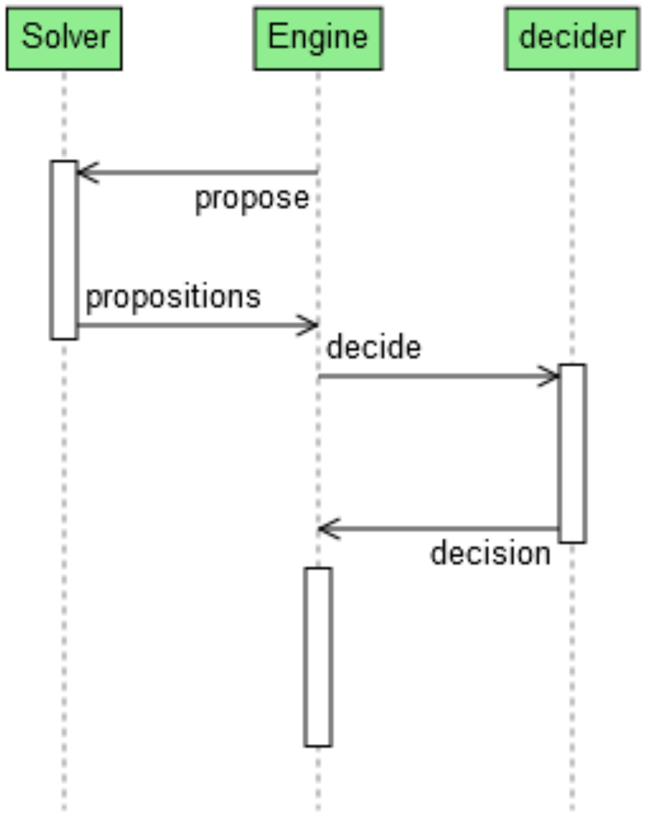


Figure 2 : Sequence Diagram of GEMOC Execution Engine

Important note: The logical steps proposed by the solver are conformed to a metamodel described in Section 4.1. The execution engine completes the logical steps with concept of choice, state of the model and state of the solver which are described in section 4.2 the execution trace metamodel.

3. Use Cases

In this section some use cases are described. It illustrates the usefulness of capturing execution traces into a trace model but it also greatly impacted the requirements of the metamodel described in Section 4.2.

3.1 Execution analysis

Execution analysis is the very first need when working with traces. Traces are used to observe what actually happened during the execution. The language engineer can then understand a specific execution by looking at the corresponding traces. This is particularly useful when looking after design flaws. In order to analyse traces, GEMOC studio has to provide some tools for trace visualization.

3.2 Execution replay

For each model execution, one execution trace can be saved. One may want to replay the exact same execution. But, due to possible non determinism, executing a second time the same model does not ensure the same result. One solution would be to use the execution trace to replay the execution. That use case implies that the trace shall embed the state of the solver and the state of the model for each execution steps.

3.3 Partial execution replay

Based on the previous use case, one may want to replay the exact same execution but this time two parameters may be specified:

- an optional custom execution step starting point,
- an optional custom execution step ending point.

Compared to execution replay, the benefit is that the user can narrow down his focus to a specific part of the execution. In fact, the execution replay use case is a specific partial execution replay with execution step starting point set to 0 and execution step ending point set to the number of execution steps in the trace.

3.4 Execution reset at runtime

While running an execution one may want to go back to a previous logical step. It is like going in the past and resuming the execution from there. That is particularly useful when testing the possible concurrency of the model according to the chosen MoCC. Instead of starting multiple model executions, the language engineer and language users can start a single execution and go back in past as many times as she/he wants to test different scenarios and explore different executions.

4. Execution Trace Metamodel

The current solver integrated into the GEMOC studio is called TimeSquare[1]. It is an external and independent software from GEMOC. TimeSquare already defines an execution trace metamodel for saving its execution steps. It seemed quite logical to reuse that language at it is. That language, presented in Section 4.1, needs to be enhanced with concepts for saving the state of the model and the solver at each execution step. That is the reason why an extension is provided in Section 4.2.

4.1 TimeSquare Execution Trace Metamodel

Note that this execution trace metamodel is independent from the solver. That means this language can be reused by any other solver than TimeSquare. In other words, new solvers can be integrated in the GEMOC Studio and still benefit from trace management.

Important note 1: The concept of clock is not defined in this language but in a dedicated language named CCSL. Please refer to the deliverable 1.1.1 for more details about clock and to deliverable 1.3.1 for an introduction to the CCSL language.

Important note 2: The concept of model specific event (MSE) is a clock that references an operation and an object. The operation is defined in the DSA. The object is an instance of a concept of the xDSML. Every time the solver makes a clock ticking, the execution engine calls the associated operation on the object.

The figure 3 shows the concepts and their relationships. Not all concepts are of importance for GEMOC. Concepts of `LogicalStep`, `EventOccurence` and `Reference` are the main focus.

- **LogicalStep**: A `LogicalStep` contains a list of `EventOccurrence`. In Figure 2 the execution engine was given a list of possible logical steps as propositions. When a logical step is elicited by the decider, the execution engine executes the logical step and it becomes the chosen logical step.

- **EventOccurrence**: This concept represents a tick of a clock as defined in the deliverable 1.3.1. Concept of `Clock` is not part of the trace metamodel (it is defined in CCSL). However for a better understanding it is represented on the class diagram.

- **Reference / ModelElementReference**: Details about one event occurrence is given by [Reference]. In the context of GEMOC only the concept of model element reference is used. [ModelElementReference] references a `Clock` which in turn may or not reference an operation and an object.

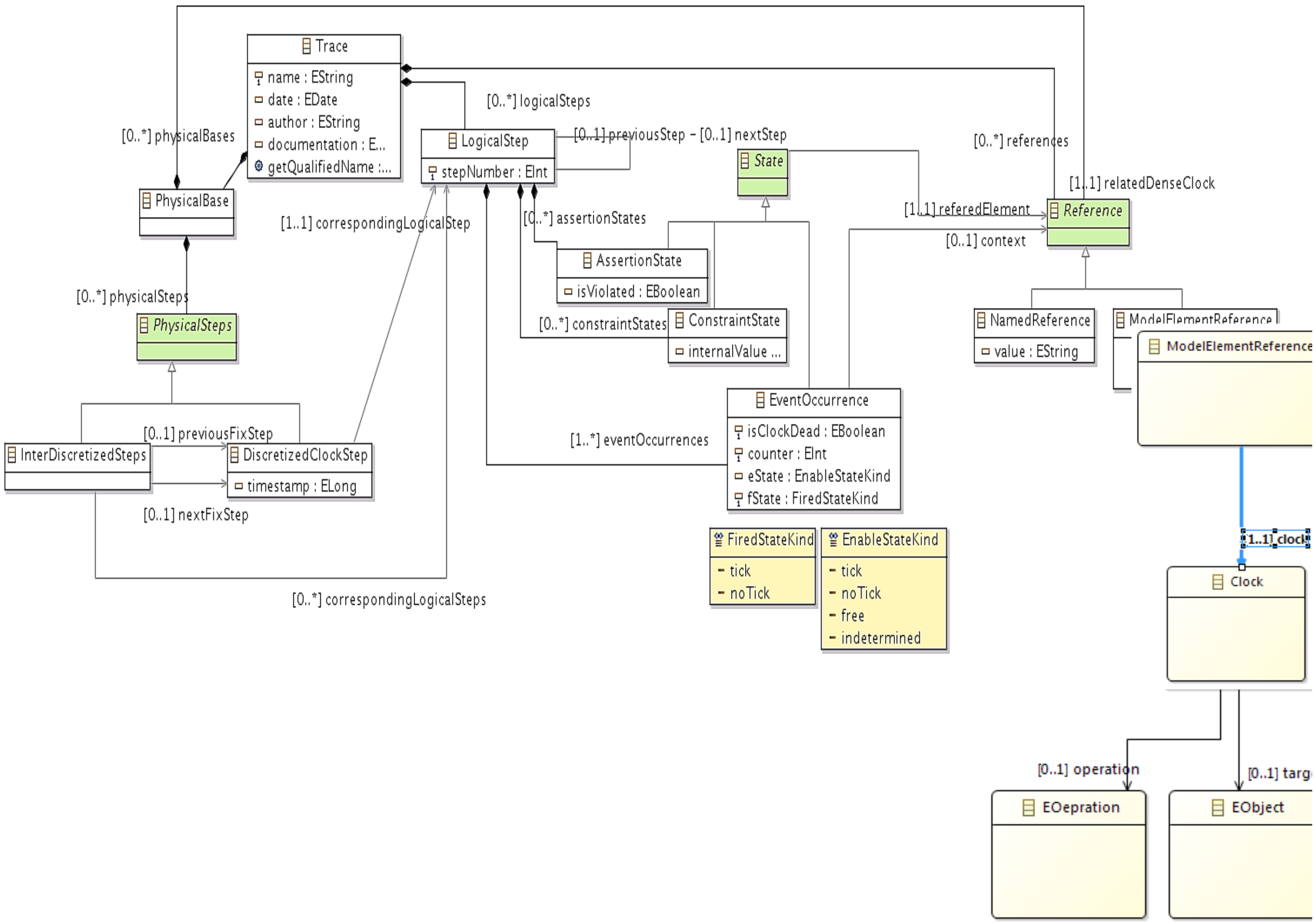


Figure 3 : Class Diagram of Timesquare's Execution Trace Metamodel

4.2 GEMOC Trace Management Metamodel

Figure 4 shows the trace metamodel that is specific to the GEMOC Studio and in particular to the execution engine. For sake of simplification of the implementation, the reader can note that even is some concepts are similar, it not reusing the metamodel presented in the previous section. The concept of `LogicalStep` is similar to the `LogicalStep` in the previous metamodel but instead of getting a graph of clocks which aren't used in the engine trace, this new `LogicalStep` stores only a set of `MSEOccurence` which store the required data. These data are used by the GEMOC engine to be able to reset its state to a previously run step and possibly restart the execution from there.

An `ExecutionTraceModel` captures the choices made during the execution. A `Choice` stores a list of possible logical steps that are proposed by the solver. Among those ones, one logical step is chosen to be executed and is referred through the reference chosenLogicalStep. In order to navigate in the trace model, a choice is linked with its previous and next choices.

A `Choice` is used as a storage to keep track of the solver's state `SolverState` as well as the state of the model under execution `ModelState`.
The concept of `Branch` allows to store variant execution traces in the same trace model. Such variant (branch) is typically generated when the engine goes back to a `ContextState` and runs a different `Choice` (ie. Continue the execution by selecting another possible `LogicalStep`)

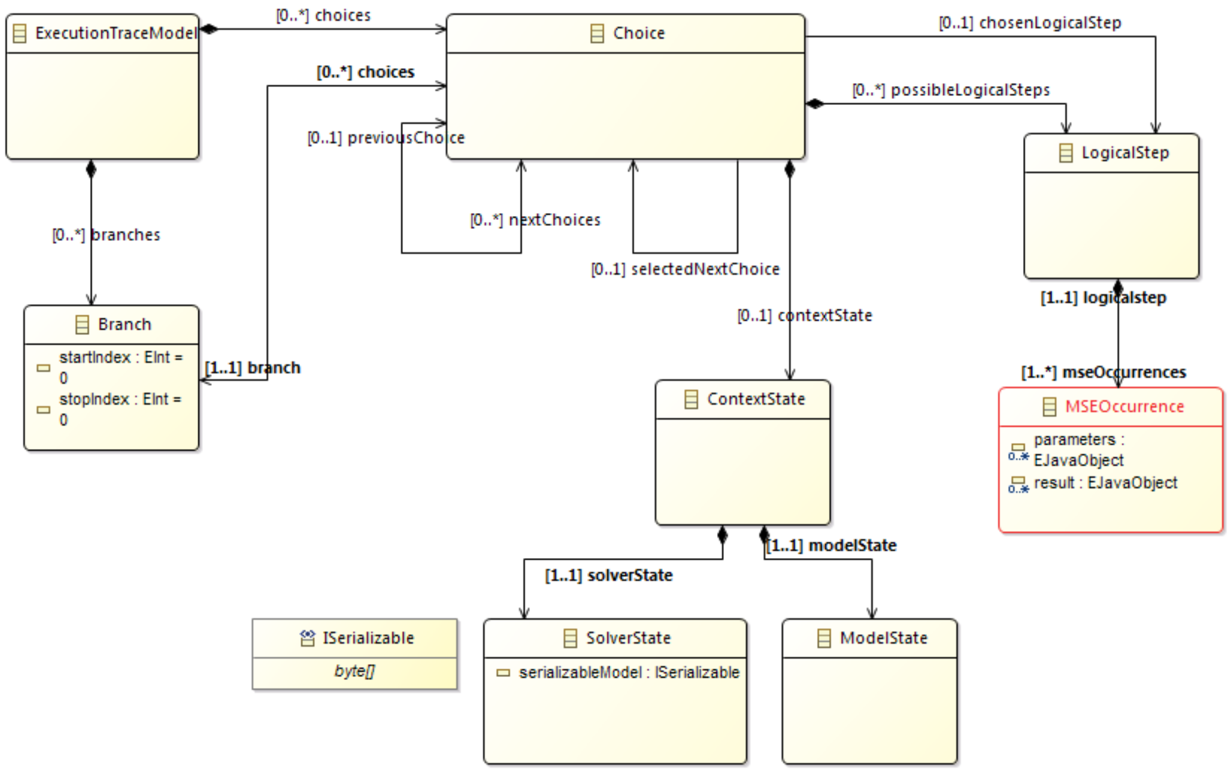


Figure 4 : Class Diagram of GEMOC's Execution Trace Metamodel

5. Integration in the GEMOC Studio

5.1 Timeline

Timeline is a graphical representation of an execution trace. It is part of the modelling workbench where models are executed. The diagram is built on top of a trace model that conforms to the metamodel described in section 4.2.

In figure 5, each circle represents a `LogicalStep`. Each vertical set of circles represents a `Choice`.

Color of a circle is described as follow:

- blue dots represents a chosen logical step among others,
- green dots represents unchosen logical steps,
- yellow dots represents proposed logical steps not yet executed.

Previous and next choices are represented on the diagram with the use of connecting line between blue circles. The execution step number is displayed at the bottom of the diagram. The vertical position of circles represents the ordered list of logical steps given by the solver. For instance, at step 0 the position of the blue circle means that this logical step was the first of a list of three. At step 3, the blue circle was the third of a list of three.

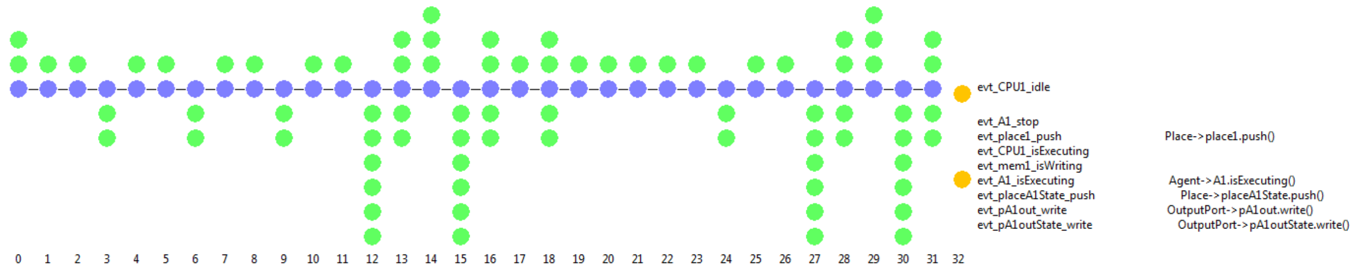
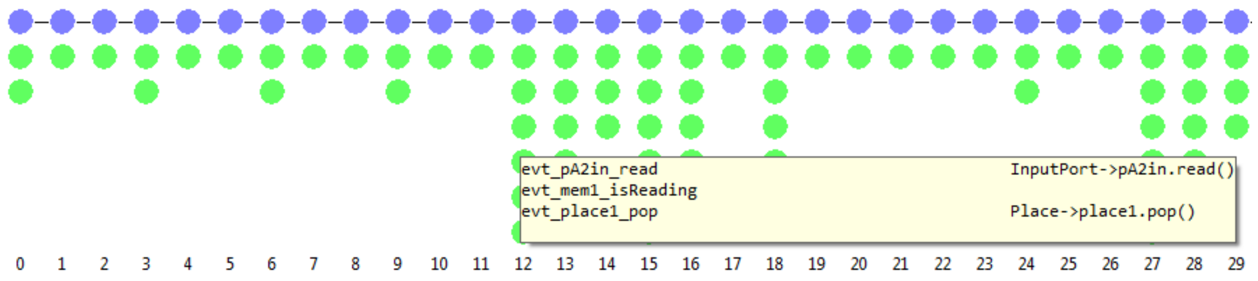


Figure 5 : Timeline Diagram showing an execution trace

Timeline offers a simple way to visualize details about **event occurrences**. Figure 6 shows that hovering the mouse over a **logical step** pops up an info text where event occurrences details are displayed. Each event occurrence is pretty printed on one line and can be split into two parts:

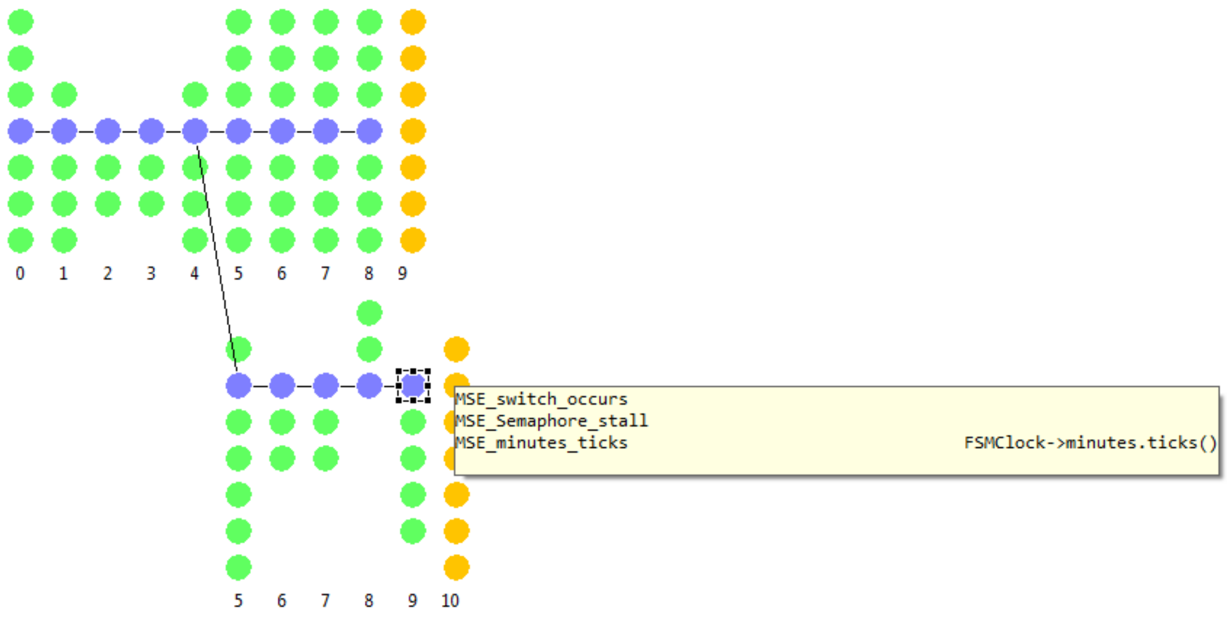
- The left part is the name of the `Clock`,
- The right part is a string that combines the name of the object and the operation referenced by the `Clock`.



```
evt_pA2in_read                              InputPort->pA2in.read()
evt_mem1_isReading
evt_place1_pop                              Place->place1.pop()
```

Figure 6 : Timeline Diagram showing details about a logical step

In Figure 7, the user has run his model up to the 9th step, then decided to reset the engine to the 4th step and continue from that point. He is able to compare the possible logical steps content directly by hovering his mouse over the bullets.
When going back in the past using this timeline's reset feature, the states of the model and solver are restored to the selected logical step also called restauration step. What has been executed between the last logical step and the restauration step is lost. But this is valuable information to keep in the trace.



```
MSE_switch_occurs
MSE_Semaphore_stall
MSE_minutes_ticks                           FSMClock->minutes.ticks()
```

Figure 7 : Timeline Diagram showing execution of a branch

Below are listed some other noticeable features:

Navigation: The diagram has been built with easy navigation in mind. Execution trace can contain a large number of choices. Therefore, the diagram offers a friendly way to zoom in the trace and out of trace as well as a smooth horizontal scrolling capability.

Dynamicity: The diagram gets updated during model execution. More precisely the diagram receives notification from the engine that a logical step has been executed.

Interactive: A double click on a blue circle resets the states of the executed model and solver at this very particular execution step as described in the use cases section 3.4. Thus allowing to run a variant execution in a branch.

5.2 Plugins

Both the language for execution trace management and the timeline are implemented in plugins. They are listed below:

- fr.inria.aoste.trace
- fr.inria.aoste.trace.edit.ui
- org.gemoc.execution.engine.trace.model.edit
- org.gemoc.execution.engine.trace.model.editor
- fr.obeo.timeline

6. Perspectives and issues

6.1 Trace model serialization issue

A trace keeps information about the model objects that have changed during execution. Designing an xDSML with GEMOC implies the use of K3 for writing the DSA. At this stage, execution data are implemented. One issue can be met when execution data are not serializable. In this case the trace model will not save properly.

6.2 Memory issue

As described in section 4.2, a trace model references copies of executed model elements. It is a very simple approach but it is not a good approach from a memory point of view. The risk of high memory consumption exists and can lead to an out of memory situation. Work on reducing trace memory footprint is underway, refer to Bousse and al.[2].

6.3 Filter the clocks and model specific events

Timeline shows a lot of information in particular the details about clocks and model specific events. But sometimes when there are too many of them it can become quite difficult to understand. In fact it would easier if the timeline was able to show only the clocks or only the model specific events or both. This could be done using filters in the user interface.

6.4 Replay

Because of the model serialization issue listed in section 6.1, the replay feature defined in section 3.2 has not been implemented. Anyway it remains an upcoming feature to have especially for model testing as described in the next section.

6.5 Model Unit Testing

When it comes to design a xDSML, the language designer will modify incrementally the DSA and the MoCC. Incrementally adding features to a language is dangerous if there is no way to test that what has been done before is still correct.

If an execution trace known to be correct could be used to perform assertions on the executed model and hence verify that the DSA and the MoCC are still correct, it would be possible to write such a test tool thanks to the replay feature as described in section 3.2.

7. References

[1] Julien Deantoni, Frédéric Mallet. TimeSquare: Treat your Models with Logical Time. Carlo A. Furia, Sebastian Nanz. TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012, May 2012, Prague, Czech Republic. Springer, Objects, Models, Components, Patterns, 7304, pp. 34-41, Lecture Notes in Computer Science - LNCS. <http://dx.doi.org/10.1007/978-3-642-30561-0_4> bibtex: http://haltools.inrialpes.fr/Public/AfficheBibTex.php?id=hal-00688590

[2] Erwan Bousse, Benoit Combemale, Benoit Baudry. Scalable Armies of Model Clones through Data Sharing. *Model Driven Engineering Languages and Systems, 17th International Conference, MODELS 2014*, Sep 2014, Valencia, Spain. https://hal.inria.fr/hal-01023681