Grant ANR-12-INSE-0011

ANR INS GEMOC

D5.4.1 - Experimentation result analysis
Task 5.4

Version 1.0

DOCUMENT CONTROL

	–:	A:	B:	C:	D:
Written by Signature					
Approved by Signature					

Revision index	Modifications
–	
A	
B	
C	
D	

Authors

Author	Partner	Role
Melanie BATS	OBEO	Lead author
Cedric BRUN	OBEO	Contributor
Benoit COMBEMALE	INRIA	Contributor
Julien DEANTONI	INRIA	Contributor
Jerome LE NOIR	THALES	Contributor
Dorian LEROY	INRIA	Contributor
Sebastien MADELENAT	THALES	Contributor

Contents

Experimentation result analysis

1. Introduction

The GEMOC project targets a language design studio providing methods and tools to ease the design and integration of new MoCCs and executable DSMLs (xDSMLs) relying on the use of proven technologies developed in previous research projects such as Cometa, CCSL, Kermeta and the meta-modelling pattern to build xDSML in order to define:

- Modeling languages with associated methods and tools for the modeling of both MoCCs and xDSMLs;
- A single cooperative heterogeneous execution framework parameterized by the MoCCs and xDSMLs definitions;
- A global MoCCs and xDSMLs design methodology encompassing these two items;A formal specification of the previous cooperation framework to prove its completeness, consistency and correctness with respect to the cooperative heterogeneous model execution needed.

1.1 Purpose

This document aims at presenting the results of the different experimentations base on the GEMOC Studio. Four experiments were developped to validate the GEMOC solution:

- fUML
- Arduino
- Farming
- Industrial case study xCapella

1.2 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.

1.3 Intended Audience

This document mainly targets any potential laguage designers that want to use the GEMOC studio.

1.4 Summary

This document defines the context in which the GEMOC studio prototyping results were evaluated. Section 2 provides a small description of the GEMOC Studio. Section 3 remembers the requirements used to evaluate the Studio. Section 4 describes the experiments done for several domain based on the GEMOC Studio. Finally, section 6 presents the experiments conclusions.

2. The GEMOC Studio

The GEMOC Studio is an eclipse package that contains components supporting the GEMOC methodology for building and composing executable Domain-Specific Modeling Languages (DSMLs). It includes two workbenches: the *GEMOC Language Workbench* and the *GEMOC Modeling Workbench*. The language workbench is intended to be used by language designers (aka domain experts), it allows to build and compose new executable DSMLs. The Modeling Workbench is intended to be used by domain designers, it allows to create and execute heterogeneous models conforming to executable DSMLs.

The GEMOC Studio is open-source and domain-independent. The studio is available at `http://gemoc.org/studio`.

The GEMOC Studio results in various integrated tools that belong into either the language workbench or the modeling workbench. The language workbench put together the following tools seamlessly integrated to the Eclipse Modeling Framework (EMF: `https://eclipse.org/modeling/emf`):

- *Melange* (`http://melange-lang.org`), a tool-supported meta-language to modularly define executable modeling languages with execution functions and data, and to extend (EMF-based) existing modeling languages.
- *MoCCML*, a tool-supported meta-language dedicated to the specification of a Model of Concurrency and Communication (MoCC) and its mapping to a specific abstract syntax of a modeling language.
- *GEL*, a tool-supported meta-language dedicated to the specification of the protocol between the execution functions and the MoCC to support feedback of the runtime data and to support the callback of other expected execution functions.
- *BCOoL* (`http://timesquare.inria.fr/BCOoL`), a tool-supported meta-language dedicated to the specification of language coordination patterns, to automatically coordinates the execution of, possibly heterogeneous, models.
- *Sirius Animator*, an extension to the model graphical syntax designer Sirius (`http://www.eclipse.org/sirius`) to create graphical animators for executable modeling languages[1].

The different concerns of an executable modeling language as defined with the tools of the language workbench are automatically deployed into the modeling workbench that provides the following tools:

- *A Java-based execution engine* (parameterized with the specification of the execution functions), possibly coupled with TimeSquare (`http://timesquare.inria.fr`) (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 system designers to visualize, save, replay, and investigate different execution traces of their models.
- *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).
- *An heterogeneous coordination engine* (parameterized with the specification of the coordination in BCOoL), which provides runtime support to simulate heterogeneous executable models.

3. Evaluation metrics

These section reminds the metrics which were used for the GEMOC Studio's evaluation. They were already presented in the D5.1.1 document.

3.1 Static Semantics Definition

REQ1 The GEMOC studio shall provide means to assess the correctness of the static semantics at language design time.

REQ2 The GEMOC studio shall provide means to assess the correctness of the static semantics at language execution time.

[1] For more details on Sirius Animator, we refer the reader to `http://siriuslab.github.io/talks/BreatheLifeInYourDesigner/slides`

3.2 Behavioural Semantics Definition

REQ3 The GEMOC studio shall provide means to assess the correctness of the behavioural semantics at language design time.

REQ4 The GEMOC studio shall provide means to assess the correctness of the behavioural semantics at language execution time.

3.3 Model of concurrency Definition

REQ5 The GEMOC studio shall provide means to define Model of concurrency either in a declarative (i.e. using constraints la MARTE/CCSL) or an operational manner (i.e. using state machine).

REQ6 The GEMOC studio shall provide means to connect the Behavioural semantics with the Model of concurrency.

REQ7 The GEMOC studio shall provide means to assess the correctness of the connection between the MoCC and the other parts of the language at language design time.

REQ8 The GEMOC studio shall provide means to assess the correctness of the connection between the MoCC and the other parts of the language at language execution time.

3.4 Graphical Syntax Definition

The GEMOC studio shall provide means to define of the Graphic syntax thanks to the Eclipse Sirius project. This Eclipse project provides:

REQ9 The ability to define workbenches providing editors including diagrams, tables or trees.

REQ10 The ability to integrate and deploy the aforementioned environment into Eclipse.

REQ11 The ability to customize existing environments by specialization and extension.

3.5 Animation

REQ12 The GEMOC studio shall provide means to extend the Graphic syntax for animation purpose.

REQ13 The GEMOC studio shall provide means to execute a model from an xDSML in an interactive or batch manner:

- REQ13.1 The batch execution shall take as parameter both the model and input data for the whole execution and produce output data.
- REQ13.2 interactive execution should allow defining breakpoints, running in a step-by-step fashion, or up to the next breakpoint, handling hierarchical execution (in a step over/step into fashion), stopping a run interactively, selecting a DSE from the ones allowed by the MoCC at each execution step.
- REQ13.3 The GEMOC studio shall provide means to view the evolution of the execution data during a run.
- REQ13.4 The GEMOC studio shall provide means to store the evolution of the execution data during a run.
- REQ13.5 The GEMOC studio shall provide means to store the decision of the user during an interactive run in order to be able to replay it in an interactive or a batch run.

At the time of the writing of this report we add the following requirements related to model coordination.

3.6 Coordination

REQ14 The GEMOC studio shall provide means to specify coordination patterns between the behavioral semantics of heterogeneous modeling languages.

REQ15 The GEMOC studio shall provide means to generate the coordination between a set of given models according to the applied coordination patterns.

REQ16 The GEMOC studio shall provide means to interpret a given coordination between heterogeneous models.

4. Experimentations

4.1 fUML

4.1.1 Description of the experiment

The first experiment was done based on the fUML subset of UML. The Semantics of a Foundational Subset for Executable UML Models(fUML) is an executable subset of UML that can be used to define the structural and behavioural semantics of systems. It is computationally complete by specifying a full behavioural semantics for activity diagrams. This means that this DSL is well-defined and enables implementers to execute well-formed fUML models (here execute means to actually run a program). This case describes a part of the execution semantics for the UML Activity Diagram language in the form of an operational semantics.

This experiment considers a complete Activity Diagram metamodel. 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.

The operational semantics are defined with Kermeta. Listing 4.1.1 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.

```
1  @Aspect(className=ActivityNode)
2   class ActivityNodeAspect {
3   List<Token> heldTokens = new ArrayList<Token>
4   }
5
6   abstract class Token {
7   public ActivityNode holder
8   }
9
10  class ForkedToken extends Token {
11  public Token baseToken ;
12  public Integer remainingOffersCount;
13  }
```

Next step is to define the steps of computation.

Listing 4.1.1 shows an excerpt of the definition of the steps of computation (i.e., the interpreter), which manipulates the runtime concepts previously defined. The implementation follows the Interpreter design pattern, defining one operation execute per concept of the abstract syntax to be interpreted. Each method is modularly defined in an aspect, and then weaved into the suitable class of the abstract syntax. It 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 }
```

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  }
```

Based on the resulting xDSML, the language workbench includes a generative approach that automatically provides a rich and efficient domain-specific trace metamodel.

Then, Sirius Animator is used to complement the xDSML with a graphical model animator

Finally, MoCCML is used to specify the possibly timed causalities and synchronizations among the steps of computation in a formal way.

In the following listing 4.1.1, lines 1 and 2 define 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 constrained 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 in Line 6. From such a specification, and for a specific model, a symbolic event structure is automatically derived.

```
1  context ActivityNode
2        def : executeIt : Event = self.execute()
3  inv waitControlToExecute:
4  (not self.oclIsKindOf(MergeNode)) implies
5    Relation Precedes(self.incoming.source.executeIt, self.executeIt))
6
7  context Activity
8    def : start : Event = self.initialize()
9    def : finish : Event = self.finish()
10 inv NonReentrant:
11   Relation Alternates(self.start, self.finish)
12
13 context InitialNode
14 inv startedWhenActivityStart:
15   Relation Precedes(self.activity.start, self.executeIt )
```

The implementation of the UML Activity Diagram language is automatically deployed in the GEMOC modeling workbench (see Fig. 4.1).

As a result, the environment not only provides a model interpreter conforming to the 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.

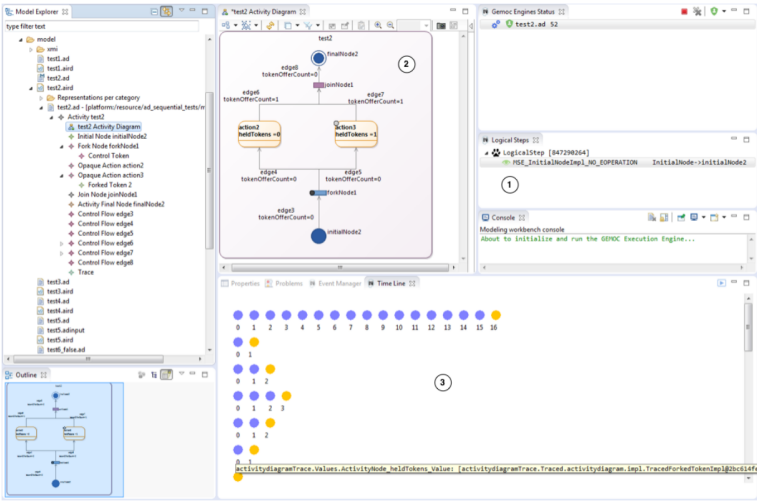

Figure 4.1: The GEMOC Modeling Workbench for the TTC15 Activity Diagram Language

The modeling workbench offers a powerful environment to system engineers for controlling the execution of their models with a debugger-like control panel(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.

All resources are available from `http://gemoc.org/ttc15`.

4.1.2 Evaluation

The following table resume the requirements covered by the fUML experiment :

Requirement	fUML
REQ1	✓
REQ2	✓
REQ3	✓
REQ4	✓
REQ5	✓
REQ6	✓
REQ7	✓
REQ8	✓
REQ9	✓
REQ10	✓
REQ11	✓
REQ12	✓
REQ13	✓
REQ13.1	
REQ13.2	✓
REQ13.3	✓
REQ13.4	✓
REQ13.5	
REQ14	✓
REQ15	✓
REQ16	✓

4.2 Arduino

4.2.1 Description of the experiment

The purpose of this experiment is to bring simulation and animation capabilities to Arduino Designer. A complete description is provided on the following webpage:
`http://gemoc.org/breathe-life-into-your-designer`.

Arduino Designer is a simple tooling based on Sirius which provide a DSL to graphically design programs (namely sketches) based on a given hardware configuration (arduino with sensors and actuators). Once a program is defined the user can automatically deploy it on an actual arduino. Behind the scene Arduino Designer will generate the .ino files, launch the compiler and upload the binary. The simulation and animation capabilities provide a convenient way to debug sketches at the level of abstraction provided by the DSL, without having to systematically compile and deploy the binary on an arduino. This helps the developer to minimize the round-trip between the design of the sketch and the test of the program, and to design the sketch without having necessarily the arduino.

First step of the experiment was to design the DSL of Arduino Designer. The DSL covers hardware and software aspects. An excerpt of the metamodel is shown in the following figure. A Project is composed of a Hardware (a particular configuration of an arduino composed of Modules) and a Sketch (program to be executed on the associated configuration of an arduino).

Once we have defined the metamodel, one can expect to examine how a conforming model (program and hardware) behaves step by step. One could even simulate interactions, and all of that without having to compile and deploy on the actual hardware. Instead of defining all the interpretation logic using pure Java code, we provide specific Xtend active annotations to seamlessly extend an Ecore metamodel with dynamic information related to the execution and the execution steps (org.gemoc.arduino.sequential.operationalsemantics). The annotation @Aspect allows to re-open a concept declared in an Ecore metamodel, and to add new attributes / references corresponding to the dynamic information, and operations corresponding to the execution steps. These execution steps are usually defined according to the interpreter pattern, traversing the metamodel to declare the interpretation of instances. The operations corresponding to specific execution steps (i.e., on which we would pause the execution, hence defining the granularity of the possible step-by-

step execution) must be declared using the annotation @Step, and the starting point of the execution must be declared using the annotation @Main.

```
1  @Aspect(className=Project)
2  class Project_ExecutableAspect {
3          @Main
4          def void main() {
5                  val sketch = _self.sketch
6                  while(true) {
7                          sketch.block.execute
8                  }
9          }
10 }
11
12 @Aspect(className=If)
13 class If_ExecutableAspect extends Control_ExecutableAspect {
14         @Step
15         @OverrideAspectMethod
16         def void execute() {
17                 if (_self.condition.evaluate as Boolean) {
18                         _self.block.execute
19                 } else {
20                         if (_self.elseBlock != null) {
21                                 _self.elseBlock.execute
22                         }
23                 }
24         }
25 }
```

From these annotations and the logic defined with Xtend we generate the corresponding code so that :

- the execution control works with EMF transaction commands,
- the dynamic informations are getting displayed in the variable view,
- Sirius gets notified to update the diagram views.

Then, the animator plugin (org.gemoc.arduino.sequential.design) is responsible for providing customizations to the diagram editor to adapt the shapes and colors based on the runtime data. This relies on the customization capabilities of Sirius :

1. The definition of a Viewpoint which extends another one from another plugin.
2. The specification of style customizations which will adapt the shapes or colors depending on a predicate.
3. The contribution of few actions to the existing Arduino tooling in order to set breakpoints for instance.

The figure Fig.4.2 shows the debugging environment automatically obtained from the design of the DSL as previously described.

4.2.2 Evaluation

The following table resume the requirements covered by the Arduino experiment:

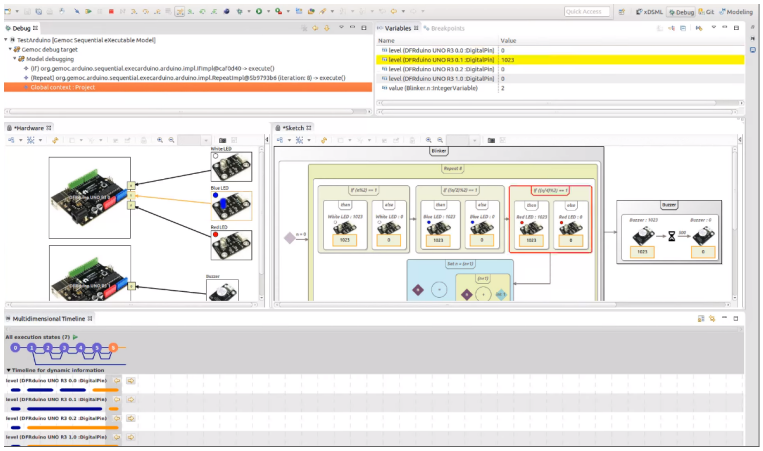

Figure 4.2: Debug session in the Arduino Designer

Requirement	Arduino
REQ1	✓
REQ2	✓
REQ3	✓
REQ4	✓
REQ5	✓
REQ6	✓
REQ7	✓
REQ8	✓
REQ9	✓
REQ10	✓
REQ11	✓
REQ12	✓
REQ13	✓
REQ13.1	
REQ13.2	✓
REQ13.3	✓
REQ13.4	✓
REQ13.5	
REQ14	✓
REQ15	✓
REQ16	✓

For details about assert explanation see Section 5.

4.3 Farming

4.3.1 Description of the experiment

This case is a proof of concept used to evaluate what DSLs and MDSD technologies can contribute to a domain where multi-disciplinary activities have the potential to have major gains in resource usage (here water). In the case of the farming use case several stakeholders are involved :

- Agricultural Scientist : owns and design the scientific models used to compute biomass and water requirements. Can try other models, change the parameters and inspect the exploitations design to reason on his research.

- Farmer : is able to assess what if scenarios at his exploitation level : ”What if i had one more equipement ? What if I would grow weat instead of corn ? ”. Is also able to estimate the quantity of water he will have to book and buy to the Institutional Resource Manager as those contracts are yearly made.
- Institutional Resource Manager : Benefit from a better planning and assessment of the use of the water resource. Could, in case of drought, make better informed decision as to which exploitation will not have the water they booked in order to minimize the impact on the biomass growth.

Four DSLs are defined and used in an integrated way to provide tooling and analysis capabilities over an exploitation.

- Activities DSL : used capture a static definition of the rules related to some cultures : how they are split in activities and what might trigger an activity. It also provides a textual syntax (based on Xtext) which makes it easier to specific the requirements and to compose expressions.
- Exploitation DSL : used to capture the structure of a given exploitation. Its break down in fields, the available resources and how the workgroups are setup in the exploitation.
- Simulation DSL : used to capture simulation settings and result : the climate data for a given location and the schedule and ressource allocation to achieve the culture.
- Scientific DSL : used to capture the parameters and data required to proceed to a biomass and irrigation analysis.

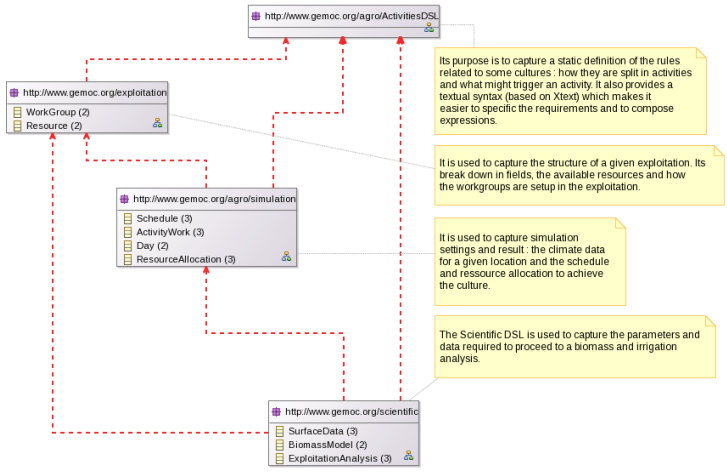

Figure 4.3: Farming DSLs

A textual syntax is used to describe the agricultural activites. Text is more suited for this case as it is composed of expressions and predicates which can be composed togethers.

A graphical syntax is used to describe the exploitation structure. Graphics are more suited in this case as they give a birds-eye view and we can use the shapes size and colors to convey information which helps the analysis :

- the size of the fields shapes matches the area of those fields.
- the color of the fields represents the fact that they are fodder and watered or not.
- gauges are used to represent the ratio of the watered fields (W) fodder fields (F) and non watered non fodder fields (N) which is assigned to a given workgroup.

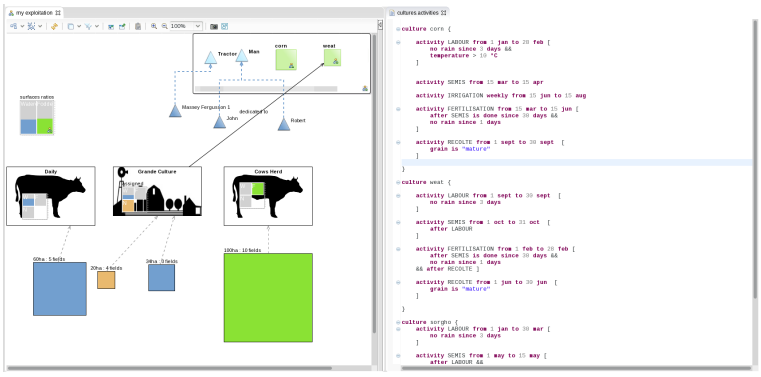

Figure 4.4: Farming syntaxes

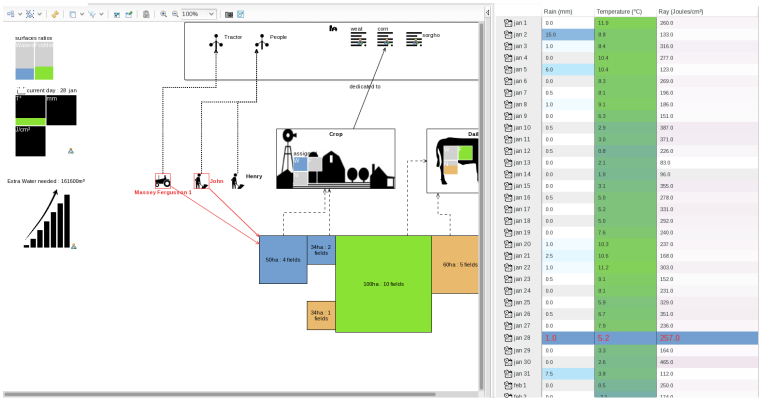

Figure 4.5: Farming DSLs

Also this diagram can be animated to go through the days and directly see who works on which field and the corresponding weather.

Using as an input the set of predicates associated with each step of a culture and, the surfaces to be used and the climate data per day, a model representing the work to do is computed. This model effectively capture the work to do and the corresponding resource allocations, then a solver is triggered to find a planning solution which fits the requirements (if possible).

From the schedule, the exploitation definition, the climate data and a model of the biomass growth an irrigation and biomass analysis can be conducted. It determines when a surface has to be watered and from this and the daily temperatures, rays and rain compute the biomass evolution. Using this information one can plan how much water will be required for the culture for the year, one can also assess scenarios with more resources (peoples, machines) or with surfaces assigned to other kind of cultures.

This tooling made of Eclipse Plugins which can be built and archived in a so-called update-site with a single right-click. It can be deployed as a standalone product directly executable on Windows, Linux and MacOSX or installed in a pre-existing Eclipse instance.

4.3.2 Evaluation

The following table resume the requirements covered by the Farming experiment:

Requirement	Farming
REQ1	✓
REQ2	✓
REQ3	✓
REQ4	✓
REQ5	
REQ6	
REQ7	
REQ8	
REQ9	✓
REQ10	✓
REQ11	✓
REQ12	✓
REQ13	✓
REQ13.1	
REQ13.2	✓
REQ13.3	✓
REQ13.4	✓
REQ13.5	
REQ14	
REQ15	
REQ16	

For details about assert explanation see Section 5.

4.4 Industrial case study xCapella

4.4.1 Description of the experiment

Arcadia (`https://www.polarsys.org/capella/arcadia.html`) is a model-based engineering method for systems, hardware and software architectural design. It has been developed by Thales between 2005 and 2010 through an iterative process involving operational architects from all the Thales business domains. Arcadia promotes a viewpoint-driven approach (as described in ISO/IEC 42010 Systems and Software Engineering - Architecture Description) and emphasizes a clear distinction between need and solution. The *Capella* modeling workbench is an Eclipse application implementing the ARCADIA method providing both a DSML and a toolset which is dedicated to guidance, productivity and quality. The Capella DSML aggregates a set of 20 metamodels and about 400 meta-classes involved in the five engineering phases (*aka*. Architecture level) defined by ARCADIA. The *Capella* modeling workbench is based on Sirius in order to define the graphical concrete syntax of the Capella DSML. *Capella Studio* provides a full-integrated development environment, which aims at assisting the development of extensions for Capella modeling workbench. This studio is based on Kitalpha incubated at Thales for several years before being recently released in open source as one of the PolarSys projects. Kitalpha allows viewpoint designers to extend the Capella DSML. Despite the existence of behavioral models, the Capella modeling workbench does not provide any simulation capability. The Capella behavioral models are limited to: modes and states, functional chain data flows, and scenarios. Neither the behavioral semantics or the coordination between these languages are defined.

In order to support the execution of models, a dedicated executable concurrent semantics is required. We started with two of the three behavioral languages from the Capella DSML (*i.e.*, data flows and mode automata).

In addition, to capture the interaction between the models conforming these behavioral languages, we specified the behavioral coordination patterns between them (Fig. 4.6).

Our study proposes to use the GEMOC studio to support system engineers so they can tame system modeling activity and improve the confidence in the specification of the system to be built. Our goal is to

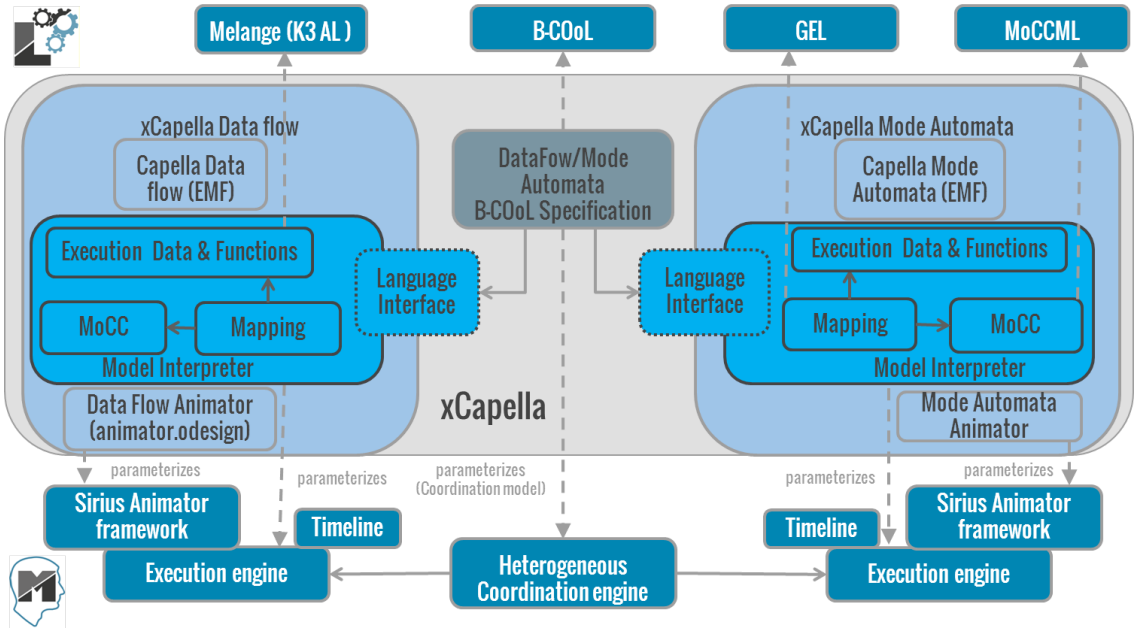

Figure 4.6: xCapella

reduce the risks concerning inconsistent functional requirements by providing a simulation environment of the existing specification, suitable to understand/analyse the system behavior.

This section presents our approach to design the concurrency-aware xDSMLs of Capella. This experiment relies on the Capella metamodel (which is publicly available[1]) augmented with a dedicated extension for mode automata. This mode automata extension has been done by using Kitalpha, integrated to the Capella studio.

Definition of behavioral semantics: To provide a behavioral semantics, we defined the semantics in two steps: (1) an extension of the metamodel with execution function and execution data and (2) the concurrent control flow definition. In this section, we focus on the definition of the mode automata semantics. The data flows semantics is not shown in this article but is used in the coordination pattern specification defined later in this paper. The mode automata DSML (Fig. 4.7 at the left side) has been extended with kitAlpha in order to add classes, attributes, references specifying the Execution Data (ED) (Fig. 4.7 at the right side). With the *Melange* tooling support, the mode automata DSML is extended by the definition of the execution functions, which define the sequential part of the mode automata operational semantics. Melange weaves the additional operation implementations specifying the execution functions (Fig. 4.7 at the bottom side).

The execution functions are orchestrated by the definition of a data-independent concurrent control flow (the data-dependent aspects of the control flow are encapsulated in the execution functions). In our approach, this control flow is captured in the so called Model of Concurrency and Communication (MoCC). The MoCC is a set of DSEs, specifying at the language level how, for a specific model, the event structure defining its concurrent control flow is obtained. The event structure represents all the possible execution paths of the model (including all possible interleavings of events occurring concurrently). For the definition of this control flow we used MoCCML to specify our MoCC. MoCCML is a declarative meta-language designed to express constraints between events. The constraints can be capitalized into some libraries that are agnostic of any abstract syntax. The MoCC is compiled to a Clock Constraint Specification Language (CCSL) model interpreted by the TimeSquare tool. The definition of the DSEs is realized by using the Event Constraint Language (ECL), an extension of OCL which allows the definition of DSE in the context of concepts from the metamodel (see listing 4.1 where DSEs *entering* and *leaving* are defined in the context of

[1] https://www.polarsys.org/projects/polarsys.capella

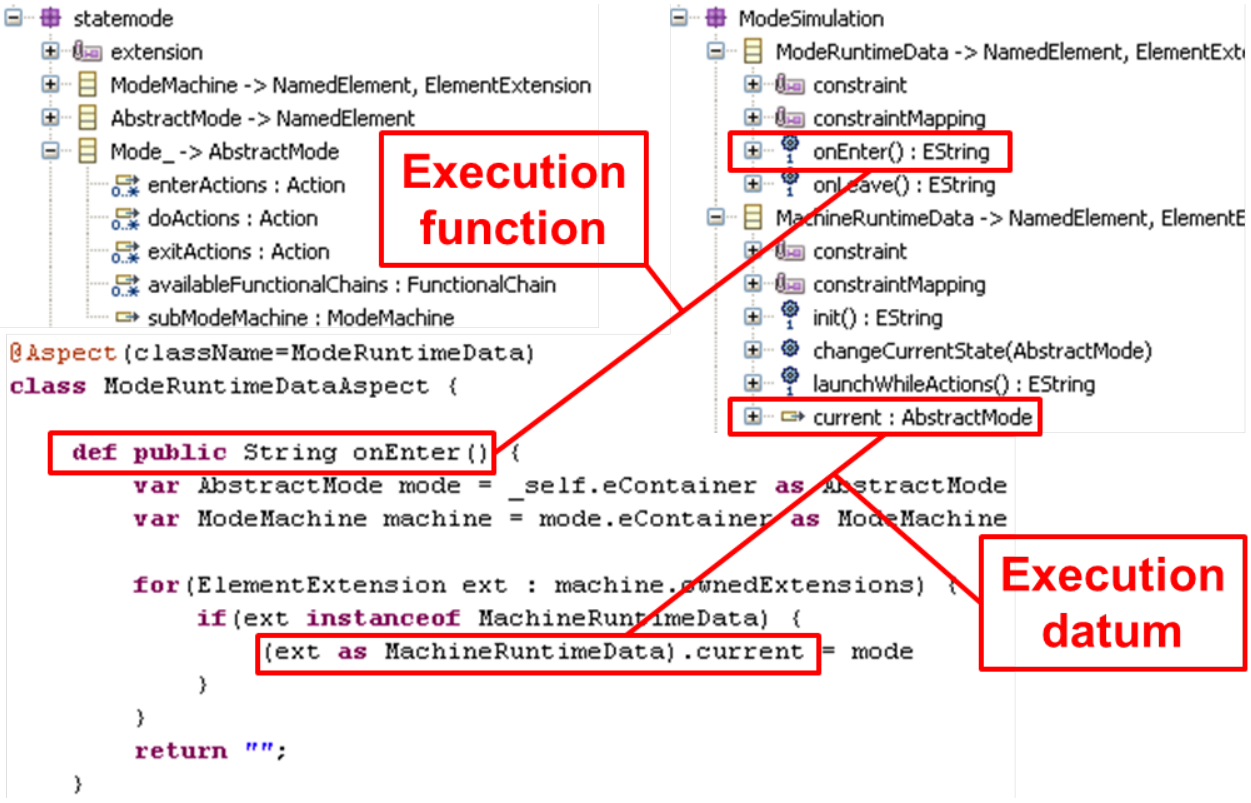

Figure 4.7: GEMOC-xCapella: Definition of execution functions and data

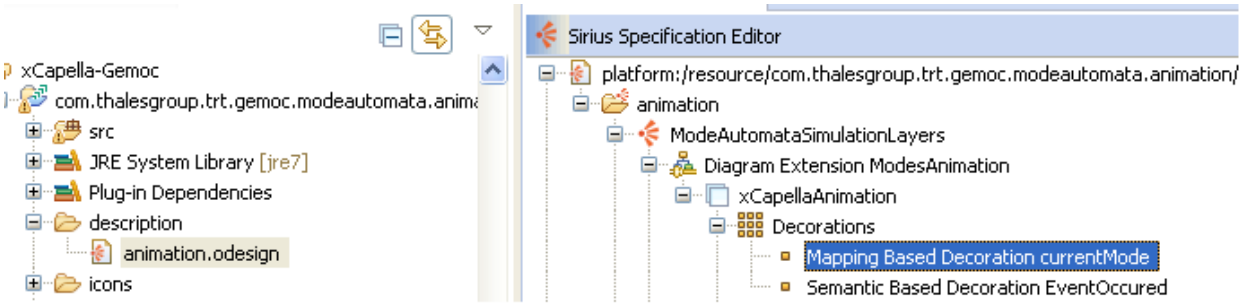

Figure 4.8: GEMOC-xCapella animation layer

an *AbstractMode*). Finally, the behavioral semantics is obtained by using a *Communication Protocol* which maps some of the DSEs from the MoCC to the execution functions (see Listing 4.1 where the DSEs are mapped to the execution functions *onEnter()* and *onLeave()* defined in the extension of Figure 4.7). This means that at the model level, when an event occurs, it triggers the execution of the associated execution function on an element of the model. Currently the implemented communication protocol is quite simple but *GEL* can be used to support more complex communication, for instance to specify data-dependent control.

Listing 4.1: Partial ECL specification of the mode automata

```
package statemode
 context AbstractMode
  def : entering : Event = self.ownedExtensions->select(E |
                                E.oclIsTypeOf(ModeRuntimeData))->first().
                                onEnter()
  def : leaving : Event = self.ownedExtensions->select(E |
                                E.oclIsTypeOf(ModeRuntimeData))->first().
                                onLeave()
```

Definition of the animation layer: We provide a new Sirius specification model (animator.odesign) which defines how the model representations change during the simulation. The animator is an extension of the concrete syntax definition which is part of Capella contributing a xCapella animation layer(Fig. 4.8) which customizes shape styles to highlight activated transition, add a decorator for the current mode and declare actions to toggle breakpoints.

The Sirius Animator framework also bring an integration with the Eclipse Debug user interface to inspect the runtime state of the execution, navigate to the corresponding diagrams and control the execution step by step.

Definition of the coordination between xData-Flow and the xMode Automata: Once both the xData-Flow and the xMode Automata have been independently developed, it is of prime importance to specify the interactions of their models. This is realized by the specification of behavioural coordination pattern in BCOoL (Behavioral Coordination Operator Language).

In our case, a mode is associated to some functional chains. The coordination pattern must specify that a functional chain is activated (but not necessarily started) when the mode automata is in a specific mode. Consequently, when a mode automata is in a specific mode, the functional chains not associated to this mode are deactivated.

The BCOoL behavioral pattern contains two parts:

- a *matching*, which defines a predicate based on the DSE context to identify what are the events to coordinate in a specific model; and
- a *MoCCML constraint*, to specify how the matched events are coordinated.

In our BCOoL specification (see listing 4.2), the *ModeEnteringActivateFunctionalChain* operator coordinates the action of entering and leaving a mode with the activation of a functionalChain. Entering into a mode is identified by the entering DSE defined in the context of an AbstractMode in the *mode automa behavior language interface* (*i.e.*, in modemachine.ecl). Instances of such DSE have to be coordinated with instances of the *activate* DSE defined in the *data flow behavior language interface* (*i.e.*, CapellaDataflow.ecl). The *matching* specifies that the *entering* and *leaving* event from a mode are coordinated with the *activate* event from the functional chain only if the functional chain is referenced by the mode (in the *availableFunctionalChains* collection).

Listing 4.2: Heterogeneous coordination operator between the data flow and mode automata languages

```
BCOoLSpec  XCapellaDataFlow-xCapellaModeAutomata

ImportLib  'platform:/plugin/.../modeAutomata.moccml'

ImportInterface  'platform:/.../CapellaDataflow.ecl' as dataflow
ImportInterface  'platform:/plugin/.../modeAutomata.ecl' as modeautomata

Operator ModeEnteringActivateFunctionalChain (enter: statemode::entering, leave: statemode
    When:
      enter.availableFunctionalChains->exists(fc | fc = activate)
    CoordinationRule:
      enableElementWhenCurrentMode(activate, enter, leave)
end Operator;
```

4.4.1.1 The GEMOC-xCapella modeling workbench

Once the xDSMLs implemented with the aforementioned tools of the language workbench, they are automatically deployed into the original *Capella* modeling workbench (integrated with the *GEMOC modeling workbench*). It results in an advanced modeling workbench integrated into the Eclipse debugger for model execution.

The GEMOC-xCapella modeling workbench (Fig. 4.9) offers an environment for system engineers to understand/control the execution of their models with :

1. a graphical feedback of their model execution. For instance, in Figure 4.9, the green arrow on *initializeSystem* state represents the current state.

2. a possibility to explore several execution traces with a graphical timeline that supports step forward and step backward. The timeline and the concurrent logical step decider can be used conjointly by a designer to choose the next step in case of non determinism or concurrent events. For instance in the timeline, each vertical list of bullets represents some possible futures at this step. Also, at any time during the simulation, the designer can go back in the past to explore an alternative future.

3. a possibility to add some breakpoints to *pause* the simulation when the element carrying the breakpoint is touched (*i.e.*, when an operation is called on it).

Additionnaly, a designer can use the *execution model*, which represents the causalities and synchronizations in the model (*i.e.*, the timemodel file) to generate the state space of all possible execution traces from the concurrency point of view.

4.4.2 Evaluation

The following table resume the requirements covered by the xCapella experiment: 135

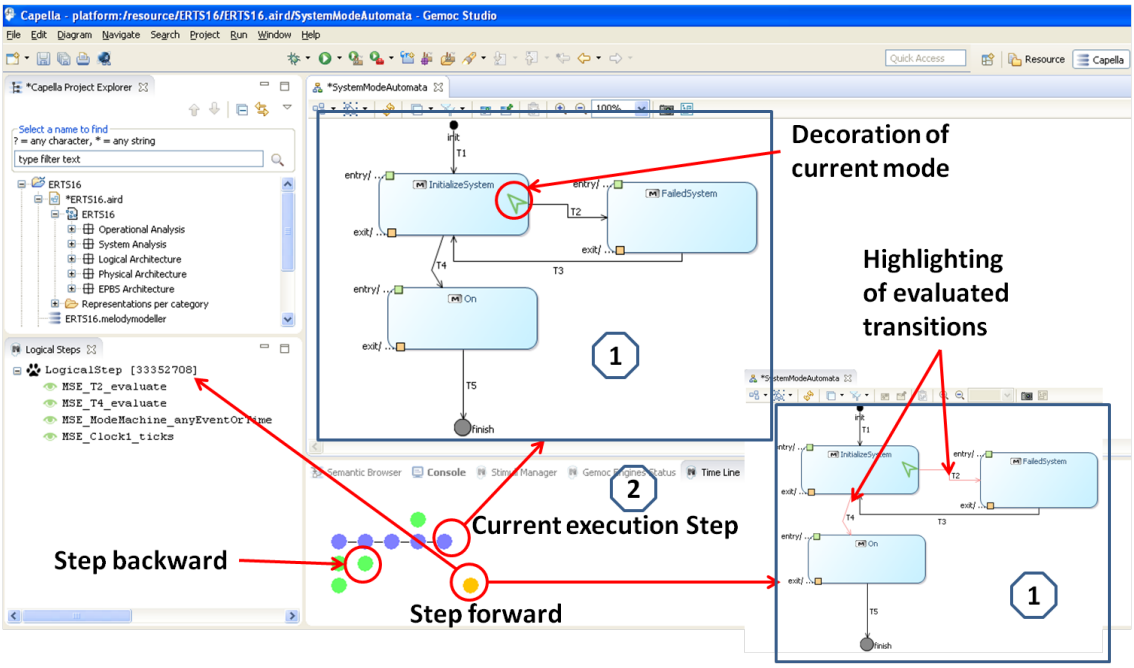

Figure 4.9: GEMOC-xCapella modeling workbench

Requirement	xCapella
REQ1	✓
REQ2	✓
REQ3	✓
REQ4	✓
REQ5	✓
REQ6	✓
REQ7	✓
REQ8	✓
REQ9	✓
REQ10	✓
REQ11	✓
REQ12	✓
REQ13	✓
REQ13.1	
REQ13.2	✓
REQ13.3	✓
REQ13.4	✓
REQ13.5	
REQ14	✓
REQ15	✓
REQ16	✓

For details about assert explanation see Section 5.

5. Experimentation results

The following table summaries the requirements covered by the different cases.

Requirement	fUML	Arduino	Farming	xCapella
REQ1	✓	✓	✓	✓
REQ2	✓	✓	✓	✓
REQ3	✓	✓	✓	✓
REQ4	✓	✓	✓	✓
REQ5	✓	✓		✓
REQ6	✓	✓		✓
REQ7	✓	✓		✓
REQ8	✓	✓		✓
REQ9	✓	✓	✓	✓
REQ10	✓	✓	✓	✓
REQ11	✓	✓	✓	✓
REQ12	✓	✓	✓	✓
REQ13	✓	✓	✓	✓
REQ13.1				
REQ13.2	✓	✓	✓	✓
REQ13.3	✓	✓	✓	✓
REQ13.4		✓	✓	✓
REQ13.5				
REQ14	✓	✓		✓
REQ15	✓	✓		✓
REQ16	✓	✓		✓

- REQ1 : EMF is used to implement the static semantics. EMF provides validation tools integrated with the EMF tree editor.

- REQ2: EMF is used to generate Java code which will be used at execution time. The JDT offers a great integration with the Eclipse debugger.

- REQ3 : Kermeta is used to implement the operational semantics. 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 steps of computation are defined in terms of operations weaved into the suitable classes, either from the initial metamodel or from the newly added classes of the runtime concepts. 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. 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. Once the operational semantics is defined with Kermeta, Melange is used from the language workbench for assembling the initial metamodel and the chosen operational semantics into an xDSML.

- REQ4 : Kermeta results in a Java-based runtime seamlessly integrated to the Java code generated by EMF. Just like for the EMF generated code, with the Kermeta runtime the JDT offers a great integration with the Eclipse debugger.

- REQ5 : MoCCML is used to specify the possibly timed causalities and synchronizations among the steps of computation. Non-determinism and parallelism are clearly identified in the operational semantics. Analysis tools are also provided in the GEMOC Studio to analyze the MoCC.
- REQ9 : Sirius is used to create a graphical editor for the static and behavioural semantics.
- REQ10 : The implementation of the language is automatically deployed in the GEMOC modeling workbench.
- REQ11 : Sirius comes with extension capability. The GEMOC studio use them to extends static representation for animation pupose.
- REQ12 : Sirius Animator is 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. A new graphical representation is defined on top of the metamodel, augmented with the runtime concepts to be visualized at runtime.
- REQ13 : Interactive and batch executions are possible with the deployed environment.
- REQ13.1 : Interactive and batch executions are possible with the deployed environment.
- REQ13.2 : The Sirius Animator provides a good integration with the Eclipse debugger. The representation is then used in an interactive mode to visualize the state of the model during its execution.
- REQ13.3 : The Sirius Animator provides a good integration with the Eclipse debugger. The evolution of the model is then visible on the representation during its execution.
- REQ13.4 : Based on the xDSML, the language workbench includes a generative approach that automatically provides a rich and efficient domain-specific trace metamodel.
- REQ13.5 : Based on the xDSML, the language workbench includes a generative approach that automatically provides a rich and efficient domain-specific trace metamodel.

  All of the requirements are entirely implemented and supported by the Gemoc Studio except REQ13.5. REQ13.5 has been partially addressed, i.e. execution traces are stored and saved after a given execution, such execution trace would be loaded for re-execution.

6. Conclusion

The GEMOC Studio integrates different technologies to implement the various concerns of the executability (runtime concepts, steps of computation, animator, concurrency). We evaluate the studio regarding the requirements defined and give evidence for the validity of the studio.

The GEMOC Studio is a playground 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).

