Execution Framework of the GEMOC Studio (Tool Demo)

Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, Benoit Combemale

Execution Framework of the GEMOC Studio (Tool Demo)

Erwan Bousse
TU Wien, Austria
bousse@big.tuwien.ac.at

Thomas Degueule
Inria, France
thomas.degueule@inria.fr

Didier Vojtisek
Inria, France
didier.vojtisek@inria.fr

Tanja Mayerhofer
TU Wien, Austria
mayerhofer@big.tuwien.ac.at

Julien Deantoni
Université Côte d'Azur, I3S, France
julien.deantoni@polytech.unice.fr

Benoit Combemale
Univ. Rennes 1 and Inria, France
benoit.combemale@inria.fr

Abstract

The development and evolution of an advanced modeling environment for a Domain-Specific Modeling Language (DSML) is a tedious task, which becomes recurrent with the increasing number of DSMLs involved in the development and management of complex software-intensive systems. Recent efforts in language workbenches result in advanced frameworks that automatically provide syntactic tooling such as advanced editors. However, defining the execution semantics of languages and their tooling remains mostly hand crafted. Similarly to editors that share code completion or syntax highlighting, the development of advanced debuggers, animators, and others execution analysis tools shares common facilities, which should be reused among various DSMLs. In this tool demonstration paper, we present the execution framework offered by the GEMOC studio, an Eclipse-based language and modeling workbench. The framework provides a generic interface to plug in different execution engines associated to their specific metalanguages used to define the discrete-event operational semantics of DSMLs. It also integrates generic runtime services that are shared among the approaches used to implement the execution semantics, such as graphical animation or omniscient debugging.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; I.6 [*Simulation and modeling*]: Types of Simulation—Discrete event

Keywords domain-specific modeling language, language and modeling workbenches, model execution, debugging

SLE '16, October 31-November 01, 2016, Amsterdam, Netherlands

ACM 978-1-4503-4447-0/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2997364.2997384

1. Introduction

The integration of domain-specific concepts and best practices into Domain-Specific Modeling Languages (DSMLs) can significantly improve software and systems engineers productivity and system quality (Hutchinson et al. 2011b). Domain-specific models are used in the development processes to reason and assess specific properties over the system under development as early as possible. This usually leads to a better and cheaper design as more alternatives can be explored. While many models only represent structural aspects of systems, a large amount express behavioral aspects of the same systems. Behavioral models are used in various areas (*e.g.*, enterprise architecture, software and systems engineering, scientific modeling, etc.), with very different underlying formalisms (*e.g.*, business processes, orchestrations, functional chains, scenarios, protocols, activity diagram, etc.).

The development of DSMLs has only been recently recognized as a significant software engineering task in itself (Hutchinson et al. 2011a). To support the tedious task of implementing tool-supported DSMLs, specialized tools (aka. language workbenches) have been proposed during the last decade to provide generic or generated syntactic services for DSMLs. From a DSML specification, Eclipse-based tools, such as Sirius[1] and Xtext (Eysholdt and Behrens 2010), automatically provide advanced services for graphical and textual edition. This includes parsers, syntax highlighting, auto completion or "quick fix" suggestion systems.

For behavioral models, early dynamic validation and verification (V&V) techniques are required to ensure that they are correct with regard to their intended behavior, such as simulation, testing, debugging, or runtime verification. Dynamic V&V techniques may require to execute the models, in accordance with a given specification of the execution semantics of the DSML. Such a specification would eventually provide an execution engine, either in the form of an interpreter (*i.e.*, operational semantics) or a compiler (*i.e.*, translational

[1] `http://www.eclipse.org/sirius/`

semantics). However, the expected *runtime* services on top of the execution engine remain mostly hand crafted and dedicated to the execution engine developed for each new DSML. In particular, graphical animation and interactive debugging are common facilities to observe and control an execution to better understand a behavior or to identify the cause of a defect. Similarly to editors that share code completion or syntax highlighting, the development of advanced debuggers, animators, and others execution analysis tools shares common facilities, which should be reused among various DSMLs.

In this tool demonstration paper, we introduce the *execution framework* of the GEMOC Studio, an Eclipse-based language and modeling workbench. This framework provides a generic interface to integrate different execution engines, themselves associated to their specific metalanguages to define the discrete-event operational semantics of DSMLs. The framework also integrates runtime services that can be shared among the various approaches used to implement the execution semantics. These services include graphical animation, omniscient debugging or trace management. We illustrate the framework with the integration of different engines that are applied on the development of modeling environments for various DSMLs. We demonstrate our approach through the definition of different xDSMLs with different approaches, and the presentation of the resulting environments provided by the modeling workbench to animate and debug models.

2. On Model Execution

Debugging, and dynamic V&V techniques in general, require models to be executable, which is achieved by defining the execution semantics of the executable DSMLs (xDSMLs) used to define them. The execution semantics of an xDSML is mainly implemented either as a compiler (aka. translational semantics) or as an interpreter (aka. operational semantics).

A compiler consists in generating executable code, usually targeting an execution environment that provides the tools of interest for the modeling language under development (*e.g.*, virtual machine, debugger, checkers). While this approach allows language designers to reuse existing, possibly efficient, tools for new modeling languages, it comes with two difficulties. First, the definition of the semantics is given in terms of the targeted execution environment. This makes difficult the definition and understanding of the resulting execution semantics. Second, to be useful to the user of the xDSML there is a need for a back-annotation mechanism to trace back the execution results in terms of the initial model.

An interpreter is a virtual machine for a modeling language in charge of executing any conforming models. The interpreter defines the data structure representing the execution state of a model, and the execution rules in terms of endogenous, possibly in-place, transformations of such an execution state.The execution state can be either an extension of the syntax of the modeling language (*e.g.*, the execution state of a statechart could be specified by an additional collection containing the current states) or a separate data structure defined by its own metamodel (*e.g.*, the execution state of a Petri net may be represented as a matrix). While the main benefit of this approach is to define the execution semantics directly in terms of the concepts of the modeling language, the main drawback is the necessity for any new xDSML to implement all the tooling based on the execution semantics, *e.g.*, a debugger. Developing such a complex tool for a xDSML remains a difficult and error-prone task.

Various approaches have been investigated in the last decade to implement operational semantics (*e.g.*, (Mayerhofer et al. 2013; Engels et al. 2000; Tatibouët et al. 2014; Combemale et al. 2013)), with their own pros and cons (formality, underlying paradigm, abstraction level, or specific concern such as concurrency). Moreover, despite the specificity of each xDSML and the differences between their semantics, there is a common set of runtime facilities which can be expected for all languages. For instance, one may expect the following debugging facilities: control of the execution (pause, resume, stop), representation of the current state (*i.e.*, , model animation), breakpoint definition, step into/over/out and step forward/backward. To support the various approaches to implement the discrete-event operational semantics of xDSMLs, and to drastically reduce the development cost of runtime tools, we present in the next section the generic execution framework provided in the GEMOC studio. This execution framework is used to integrate different approaches to define the operational semantics of xDSMLs (incl., the meta-language and the associated execution engine), and provides various generic facilities for model simulation, graphical animation and debugging.

3. GEMOC Execution Framework

3.1 Overview of the GEMOC Studio

The GEMOC Studio[2] is an Eclipse package atop the Eclipse Modeling Framework (EMF), which includes:

- The *GEMOC Language Workbench*: to be used by language designers to build and compose new xDSMLs,
- The *GEMOC Modeling Workbench*: to be used by domain designers to create, execute and coordinate models conforming to xDSMLs.

The different concerns of a DSML, as defined with the tools of the language workbench, are automatically deployed into the modeling workbench. They parametrize a generic execution framework that provides various generic services, such as graphical animation, debugging tools, trace and event managers, timeline visualizations, etc.

3.2 Overview of the Execution Framework

Figure 1 shows an overview of the execution framework offered by the GEMOC studio. At the middle, the xDSML de-

[2] `http://gemoc.org/studio`

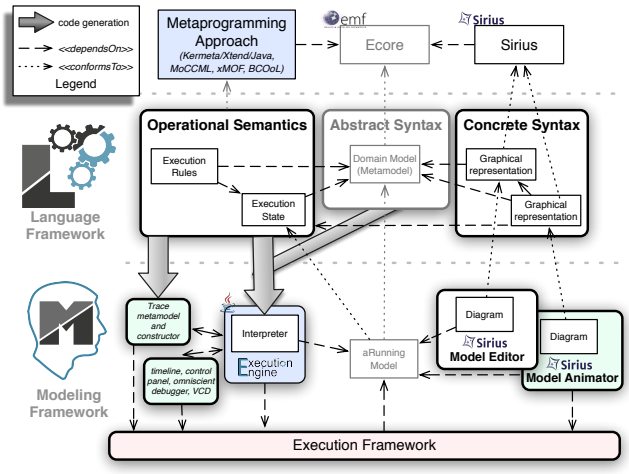


Figure 1: Overview of the GEMOC Execution Framework

fined in the language workbench is depicted. It is composed of abstract and concrete syntaxes, and of operational semantics. For a given xDSML, the operational semantics are defined using a specific metaprogramming approach (*e.g.*, specific model transformation languages). Since the GEMOC Studio is based on EMF, Ecore is used to define the abstract syntax, and at runtime the executed model is a set of EMF objects. For defining the concrete syntax, the Sirius toolkit is used. For more information and examples on the language workbench, please refer to the official documentation[3].

At the bottom, the modeling workbench supported by the *execution framework* is shown. This workbench allows the user to define an executable model conforming to an xDSML, and execute it using an execution engine and a selection of addons. An *addon* is developed using the execution framework and provides a set of runtime services (*e.g.*, animation or debugging). An *execution engine* is developed using the execution framework and is specific to a metaprogramming approach. It is responsible for integrating the interpreter of the considered xDSML – developped using the same metaprogramming approach as the engine – with the addons provided in the studio. This implies sending notifications to addons regarding the execution (*e.g.*, beginning of the run, start or end of a rule, etc.). By reacting to notifications, an addon may query the engine to ask for information, which can be used to provide a view that gets updated during the execution, or control the execution of the model, or even modify the model. Some addons are generated (*e.g.*, trace management), while others are generic and compatible with any engine. At runtime, the executed model contains a dynamic execution state that is modified by the interpreter and by addons.

3.3 Interface of the Execution Framework

There are many approaches to define discrete-event operational semantics for an xDSML. Each approach has different characteristics, such as how to initialize the execution state of a model, or how to provide a way to control the application of the semantics. In order to manage all these different situations, the GEMOC execution framework provides an API to define and integrate (discrete-event) execution engines. This API defines an engine as a component with two main operations: *initialize* to load an xDSML and a model, and to prepare the transformation; and *execute* to run the transformation. In addition to these two operations, the API comprises operations to add or remove addons, to access the current stack of execution steps, to get or set the engine status (started, paused, stopped, etc.), to access the execution context (*e.g.*, the model, the execution mode) or to start/stop the engine. This API is used both by the generic part of the framework dedicated to starting an execution, and by the addons which may need to access information or control the execution.

In addition to complying with this API, the integration of an engine in the framework requires that notifications are sent to the attached addons during the execution (*i.e.*, during the *execute* operation). For this purpose, the framework defines the common notion of *execution step*, a step being the application of an execution rule of the operational semantics. Note that an engine implementation may consider that only a subset of the rules lead to execution steps (*e.g.*, by annotating the semantics). When an engine is about to apply such rules, it must create a step object containing information about the executed rule (identifiers of the rule, parameters given, etc.). Then this step object must be used to notify addons of both the start and the end of a step. Step objects are also used in the framework for other purposes, such as the storage of a stack of the steps currently being executed, or to be directly stored within an execution trace.

Since a significant part of the logic is common to all execution engines, the framework provides a basic *abstract execution engine* that can be extended into a concrete engine. This abstract engine implements part of the API described above, such as the services to manage the status, to add or remove addons, and to start or stop. In addition, this abstract engine provides internal services to notify all addons of the progress of the execution, although the task of calling these services at the right instants is left to the concrete engines. In particular, a service called *beforeStep* is provided to be called at the beginning of a step, and second called *afterStep* must be called at the end. Depending on the technique used to define the execution semantics, the integration of these operations in the operational semantics can be done in different ways (instrumentation, event listeners, etc.).

Addons are components that can be defined to provide runtime services. Such services need to be connected to the ongoing execution of a model in order to follow the execution and to extract information (*e.g.*, the content of the execution state) or to control the execution (*e.g.*, pause or provide input). To that effect, the GEMOC execution framework provides an API that defines an addon as a component with at least four

[3] `http://gemoc.github.io/gemoc-studio/publish/guide/html_single/Guide.html`

operations that are synchronously called by the engine during the execution of a model: *engineStarted* when the execution starts, *engineStopped* when the execution ends, *aboutToExecuteStep* when the engine is about to start an execution step, *stepExecuted* when an execution step finished. Within the implementation of one of these operations, an addon can access the engine and its status, the executed model, or even the graphical interface of the studio. Therewith, an addon can accomplish a large diversity of tasks, such as changing the executed model, pausing or stoping the execution, displaying information, or sending some input data to the engine.

4. Integration of Different Execution Engines

Using the API for execution engines that we described, different metaprogramming approaches have been integrated to describe discrete-event operational semantics. In particular, we implemented four main execution engines:

Java Engine. The Java engine is dedicated to operational semantics that are entirely defined using any Java-based language, such as Java, Xtend (Efftinge et al. 2012) or Kermeta (Jézéquel et al. 2015). The assembling of the various DSML concerns (incl., abstract and concrete syntaxes, and operational semantics) is made consistent thanks to Melange (Degueule et al. 2015). The *initialize* operation consists in loading the model, and searching for the entry point specified by the xDSML (*i.e.*, the *main* method of the semantics). The *execute* operation simply consists in starting this entry point. However, to interweave notifications to addons in between execution steps, it is required that the Java semantics themselves notify the engine both before and after the performed steps (*e.g.*, by calling *beforeStep* and *afterStep* provided by the abstract engine), otherwise the engine has no way to be informed of the execution. These calls can be either introduced manually at the beginning and at the end of chosen execution rules (*i.e.*, Java methods), or can be introduced externally using aspect-oriented programming (*e.g.*, AspectJ), or can even be realized using a feature provided by the metalanguage (*e.g.*, the `@Step` annotation provided by Kermeta).

Java+MoCCML Engine. The Java+MoCCML engine is dedicated to operational semantics where the (possibly concurrent and timed) control is described in the MoCCML formal language (Deantoni et al. 2015), while the execution rules are written in any Java based language (e.g., Kermeta) (Combemale et al. 2013). In this case, the *initialize* operation creates a so called concurrency model according to the operational semantics, where some relevant events are defined, constrained together and linked to execution rules defined in Java (or any Java-based language). Then the *execute* operation consists in a loop that asks for the next possible steps to the Timesquare solver (Deantoni and Mallet 2012). There are pottentially several possible steps since it considers all acceptable interleavings of the events. Once a step is selected, it calls the corresponding Java methods as required. Notifications to and from addons are received and sent directly by the engine during this execution loop.

xMOF Engine. The xMOF engine supports the execution of operational semantics defined with xMOF (Mayerhofer et al. 2013). With xMOF the data structure for representing the execution state of a model is defined with a metamodel and the rules of the semantics are defined with UML activities conforming to the fUML standard (Object Management Group 2015). For the execution of models, the xMOF engine relies on the virtual machine of fUML. In particular, the *initialize* operation of the xMOF engine loads the model to be executed as well as an optional input model defining input values to the executed model (*e.g.*, the initial token distribution of a Petri net). These models are then handed over to the fUML virtual machine for execution in the *execute* operation of the xMOF engine. The fUML virtual machine provides sophisticated mechanisms for controlling and observing the execution of a model. These mechanisms are utilized by the xMOF engine to call the operations *beforeStep* and *afterStep* at the appropriate instants.

BCOoL Engine. The engine supports the behavioral coordination of heterogeneous models, based on coordination patterns defined using BCOoL (Vara Larsen et al. 2015). The *execute* operation consists in asking a set of coordinated engines what are the next possible steps and to merge and filter these steps to provide the next possible coordinated steps according to the coordination patterns.

5. Runtime Services

Using the API for addons that we described, we implemented a set of generic runtime services that can be shared among the different execution engines of the GEMOC Studio.

Graphical Animator. The graphical animator is an addon that updates different views in order to display the current execution state of the executed model, hence helping to understand models under execution. Since the GEMOC Studio is based on Eclipse, the animator is connected to the Eclipse debug UI to display the stack of currently executed steps in the *Debug* view, and the values of the execution state in the *Variables* view. In addition, if a graphical concrete syntax was defined for the xDSML using Sirius, the animator updates a graphical representation of the execution state of the model during the execution.

Execution Trace Addons. The execution framework provides two complementary ways to manage execution traces: a generic multibranch trace addon, or a generator of domain-specific multidimensional trace addons. Each trace addon reacts to engine notifications to capture steps and states. The *generic multibranch trace addon* captures generic traces that contain different branches, each linked to a choice made during a non-deterministic situation of the execution. The *generator of domain-specific and multidimensional* trace addon can

automatically produce an addon specific to a given xDSML to create and manage multidimensional traces (Bousse et al. 2015b).

Omniscient Debuggers. Two omniscient debugging addons are provided, each relying on a different sort of trace addon.They can provide services expected by any debugger (*e.g.*, step into/over/out, breakpoints) thanks to an integration with the Eclipse Debug UI. In addition, they provide services to explore the execution *backward* in time (back into/over/out), by relying on execution traces constructed by trace addons. They each provide a view called a *timeline*, which is an interactive graphical representation of the trace. The *multibranch debugger* relies on a multibranch trace to explore non-deterministic and concurrent executions. It allows the creation of new branches by going back in time and making different choices. The *multidimensional omniscient debuger* (Bousse et al. 2015a) relies on a multidimensional trace to explore the different dimensions of the execution, using additional stepwise operations to explore the dimensions.

VCD. The VCD addon provides a representation of the execution in the form of a timing diagram reprensented in the Value Change Dump format (VCD) defined by the IEEE Standard 1364-1995 and extended in the IEEE Standard 1364-2001. It represents the potentially parallel evolution of the calls of the execution steps. This illustrates the possibility to adapt the GEMOC studio addons to specific domains. In this case it is adapted to the Electronic Design Automation (EDA) domain, which is used to this kind of format.

Stimuli Manager. The stimuli manager is an addon provided to send *stimuli* to an ongoing execution. This addon provides a view showing all the possible stimuli that can be sent to the execution, which depends on the technique used to define the operational semantics, and on the content of the semantics. This addon is interesting for non-deterministic executions that depends on external stimuli, *e.g.*, to simulate stimuli from the external environment.

Step Decider. Lastly, the step decider is an addon provided to make choices in non-deterministic situations during an ongoing execution. It provides a view that shows all the steps possible at a given point in time, and gives the possibility to rely either on an existing decider (*e.g.*, the random decider to make a choice at random), or to manually choose the next step from the displayed possibilities. Like the stimuli manager, this addon is only interesting for non-deterministic semantics with different possible steps at some points in the execution.

6. Related Work

In the last decade, various language workbenches have been proposed with generic or generative approaches to automate the development of *syntactic* services. To name just a few, Xtext and Sirius have been developed on top of EMF to provide textual and graphical advanced editors respectively. These tools have been developed with a strong connection between the metamodeling environment and the associated services. More recently, a *Language Server Protocol*[4] has been developed to decouple the IDE and the language syntactic services. This paves the way for using the same services from various IDE that implements this protocol.

In this paper, we propose an execution framework that supports generic and generative approaches for *runtime* services (*e.g.*, graphical animator and omniscient debugging), and a generic API to connect different metaprogramming approaches and associated execution engines. Among many other projects, model execution has been previously investigated within Ptolemy (Eker et al. 2003), a framework where different Model of Computations (MoC) can be integrated based on a unique Java API. While the API can be used to implement different ways of specifying an operational semantics, the framework does not provide any way to communicate from and to user defined addons. Other interesting frameworks for model execution include, ModHel'X (Hardebolle and Boulanger 2008) and AToM3 (Lara and Vangheluwe 2002), which support multi-formalism modeling and the concurrent execution of heterogeneous models. However, both are bound to their specific metaprogramming approach.

7. Conclusion

Languages workbenches facilitate the development of DSMLs, especially for providing syntactic services. However *runtime* services are mostly handcrafted for each different xDSML and each different metaprogramming approach. Based on a common API, we proposed a framework to integrate any kind of metaprogramming approach used to define discrete-event operational semantics into an execution engine. Notably, implementing this API allows to use and reuse of generic or user-defined runtime services as *addons* that send and receive generic messages to and from the execution engines. Using this framework, we implemented four different engines, each for a specific metaprogramming approach, and a set of runtime services, such as generic debuggers that can be used for any xDSML and engine. As our project is open-source and available online, we are very open to any contributors for implementing additional execution engines (*e.g.*, to support operational semantics defined with other metaprogramming approaches) and additional runtime services.

Acknowledgments

This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011), the COST Action MPM4CPS (IC1404), the Austrian Science Fund (FWF): P 28519-N31, the Christian Doppler Forschungsgesellschaft CDL-Flex and the BMWFW (Austria). The authors also thank Dorian Leroy (Inria) and Cédric Brun (Obeo) for their help in the development of the GEMOC execution framework.

[4] `https://github.com/Microsoft/language-server-protocol`

References

E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE'15)*, pages 137–148. ACM, 2015a. doi: 10.1145/2814251.2814262.

E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *Proceedings of the 11th European Conference on Modelling Foundations and Applications (ECMFA'15)*, volume 9153 of *LNCS*, pages 45–61. Springer, 2015b. doi: 10.1007/978-3-319-21151-0_4.

B. Combemale, J. Deantoni, M. Vara Larsen, F. Mallet, O. Barais, B. Baudry, and R. France. Reifying Concurrency for Executable Metamodeling. In R. F. P. Martin Erwig and E. van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science. Springer-Verlag, 2013. URL `http://hal.inria.fr/hal-00850770`.

J. Deantoni and F. Mallet. TimeSquare: Treat your Models with Logical Time. In S. N. Carlo A. Furia, editor, *TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012*, volume 7304, pages 34–41, Prague, Czech Republic, May 2012. Czech Technical University in Prague, in co-operation with ETH Zurich, Springer. doi: 10.1007/978-3-642-30561-0_4. URL `https://hal.inria.fr/hal-00688590`.

J. Deantoni, P. Issa Diallo, C. Teodorov, J. Champeau, and B. Combemale. Towards a Meta-Language for the Concurrency Concern in DSLs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Grenoble, France, Mar. 2015. URL `https://hal.inria.fr/hal-01087442`.

T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE 2015)*, Pittsburg, United States, Oct. 2015. ACM. URL `https://hal.inria.fr/hal-01197038`.

S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: Implementing Domain-specific Languages for Java. In *11th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 112–121. ACM, 2012. ISBN 978-1-4503-1129-8. doi: 10.1145/2371401.2371419. URL `http://doi.acm.org/10.1145/2371401.2371419`.

J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proc. of the IEEE*, 91(1):127–144, 2003.

G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *Proceedings of the Third International Conference on the Unified Modeling Language (UML'00)*, volume 1939 of *LNCS*, pages 323–337. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41133-8. doi: 10.1007/3-540-40011-7_23.

M. Eysholdt and H. Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. *Object Oriented Programming Systems Languages and Applications*, pages 307–309, 2010. doi: 10.1145/1869542.1869625. URL `http://doi.acm.org/10.1145/1869542.1869625`.

C. Hardebolle and F. Boulanger. Modhel'x: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering*, pages 247–258. Springer, 2008.

J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 633–642, May 2011a. doi: 10.1145/1985793.1985882.

J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 471–480. ACM, 2011b. doi: 10.1145/1985793.1985858.

J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, 14(2):905–920, 2015.

J. Lara and H. Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306, chapter Lecture Notes in Computer Science, pages 174–188. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43353-8.

T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. xMOF: Executable DSMLs based on fUML. In *Proceedings of the 6th International Conference on Software Language Engineering (SLE'13)*, volume 8225 of *LNCS*, pages 56–75. Springer, 2013. doi: 10.1007/978-3-319-02654-1_4.

Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), V 1.2.1, January 2015. `http://www.omg.org/spec/FUML/1.2.1`.

J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14)*, volume 8767 of *LNCS*, pages 133–148. Springer, 2014. doi: 10.1007/978-3-319-11653-2_9.

M. E. Vara Larsen, J. Deantoni, B. Combemale, and F. Mallet. A Behavioral Coordination Operator Language (BCOoL). In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, number 18, page 462. ACM, Sept. 2015. URL `https://hal.inria.fr/hal-01182773`.

A. Demonstration Outline

During the demonstration of the execution framework, we will exhibit the capabilities provided by the different metaprogramming approaches integrated with their associated execution engines (Section 4), and the various runtime services integrated into the GEMOC studio (Section 5).

We will illustrate the different metaprogramming approaches integrated in the GEMOC execution framework through the development of various tool-supported executable domain-specific modeling languages (xDSMLs).

A.1 Bringing Simulation and Animation Capabilities to Arduino Designer with Kermeta

Arduino Designer is a simple tooling based on Sirius which provide a modeling language to graphically design programs (namely *sketches*) based on a given hardware configuration (Arduino board with sensors and actuators). Once a program is defined the user can automatically deploy it on an actual Arduino board. Behind the scene, Arduino Designer will generate the `.ino` files, launch the compiler and upload the binary.

Among the advantages of developing Arduino in the GEMOC Studio, the simulation and animation capabilities provide a convenient way to debug sketches at the level of abstraction provided by the modeling language, without having to systematically compile and deploy the binary on an Arduino board. 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 board.

Figure 2 shows the language workbench to design the operational semantics of Arduino Designer (left), and the modeling workbench automatically obtained to edit, animate and debug sketches on specific arduino configuration (right). All the sources are available in open source[5]. During the demonstration, we will present the different concerns to define an DSML, including the abstract syntax with Ecore, the operational semantics with Kermeta, and the graphical concrete syntax with Sirius. From such a specification, we will present the different facilities provided by the modeling workbench automatically obtained, including graphical animation, breakpoint management, trace management and forward/backward debugging.

The modeling language covers hardware and software aspects, which are defined in the abstract syntax as a domain metamodel using Ecore. 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 with Kermeta specific annotations to seamlessly extend an Ecore metamodel with dynamic information related to the execution and the execution steps. 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`.

From these annotations and the logic defined with Kermeta, we generate the corresponding Java code so that:

- addons get notified of the execution of steps,
- the execution control works with EMF transaction commands,
- the dynamic information is displayed in the variable view,
- the animator gets notified to update the views.

What remains is the definition of the graphical animation layer itself. The animator is responsible for providing customization to the diagram editor to adapt the shapes and colors based on the runtime data. It rely on the customization capabilities of Sirius.

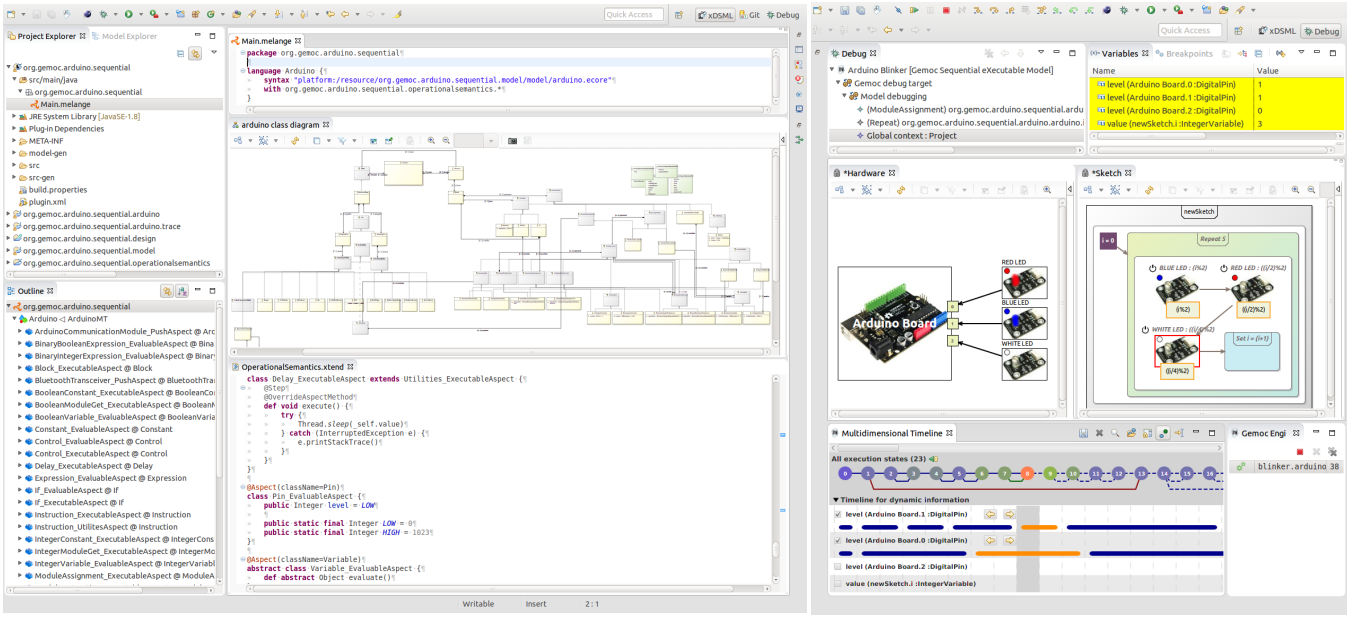
The modeling workbench shows the debugging environment automatically obtained from the design of the modeling language as previously described. The debugging environment is launched by doing *right click* $\rightarrow$ *Debug* on the model. The debugging environment is fully integrated with the Eclipse debug UI and provides the usual debugging facilities including:

- the control of the execution (pause, resume, stop), incl. step-by-step execution (step forward into, over, out),
- a graphical animation of the model during the execution to highlight the current state,
- the possibility to define breakpoints on model elements,
- the stack of the execution steps,
- the binding of the dynamic information into the variable view to show their value during the execution.

In addition, the debugging environment provides advanced features related to omniscient debugging such as an efficient management and visualization of the execution trace, the possibility to restore the model in any state previously reached during the execution, and step backward facilities (step backward into/over/out).

Finally, the timeline of the multi-dimensional debugger shows one specific execution trace per dynamic information *i.e.*, per dimension of the model. This allows the developer to navigate through the global execution trace while focusing only on the changes of a particular dynamic information represented in the *Variables* view.

[5] Cf. `https://github.com/gemoc/arduinomodeling`



(a) Language Workbench

(b) Modeling Workbench

Figure 2: Workbenches for Kermeta-based language design and model execution (example of Arduino Designer)

A.2 Operational Semantics with Kermeta and MoCCML

The emergence of modern concurrent systems (*e.g.*, Cyber-Physical Systems and Internet of Things) and highly-parallel platforms (*e.g.*, many-core, GPGPU and distributed platforms) call for Domain-Specific Modeling Languages (DSMLs) where concurrency is of paramount importance. Such DSMLs are intended to propose constructs with rich concurrency semantics, which allow system designers to precisely define and analyze system behaviors. Most of the time the concurrency model remains implicit and ad-hoc in the language design and implementation. The lack of an explicit concurrency model in language specifications prevents : the precise definition, the variation and the complete understanding of the DSML's semantics, the effective usage of concurrency-aware analysis techniques, and the exploitation of the concurrency model during the system refinement (*e.g.*, during its allocation on a specific platform).

Here, we illustrate the benefits of making explicit the concurrency by specifying the operational semantics of fUML. We implemented in the GEMOC studio the concurrency-aware executable metamodeling approach for fUML, which supports a modular definition of the execution semantics including the concurrency model of fUML defined in the formal MoCCML metalanguage (Deantoni et al. 2015) (central part of Figure 3a), the semantic rules defined in Kermeta (aka. domain-specific actions, DSA; depicted on the right part of Figure 3a), and the xDSML specification gluing together the syntactic and semantics definitions (defined on the lower part of Figure 3a). Of course the implementation of the underlying engine follows the GEMOC engine API.

By using MoCCML, the proposed semantics provides:

- explicit observation points specified by events linked to execution functions and constraint between them to specify their causalities
- possibly complex domain specific constraints embedded into reusable libraries

At any time during a run, an event that does not violate the constraints can occur. This mean that an operational semantics defined in MoCCML exhibits the interleavings of actions in the models (for example when using a *ForkNode* in an fUML model).

The constraints from the operational semantics are eventually instantiated to define the execution model of a specific model. The execution model is a symbolic representation of all the acceptable schedules for a particular model. To enable the automatic generation of the execution model, the concurrency model is weaved into the context of specific concepts from the abstract syntax of a DSML. This contextualization is defined by a mapping between the elements of the abstract syntax and the constraints of the concurrency model. The separation of the mapping from the concurrency model makes the concurrency model independent of the DSML so that it can be reuse[6].

In the GEMOC studio, the execution model is acting as the configuration of a generic execution engine, which can

[6] for instance the fUML concurrency model, here applied on a specific abstract syntax, may also be applied on the activity part of UML

be used for simulation or analysis of any model conforming to the abstract syntax of the DSML.

On Figure 3b we can see the execution of a specific fUML model where a fork node with to output actions are ready to be executed. On the left part of the Figure the different interleavings are represented: The designer can decide to execute the action on the left of the fork node, the one on the right or both together. Of course, according to the choice made for this step the remaining of the execution can differ. However, by using the multi-branch timeline represented on the lower part of Figure 3b, the designer can go back on its decision and explore other possibilities (*i.e.*, interleavings) offered by the model. Also, on Figure 3b, the VCD addon is active so that the execution is traced according to the VCD format.

A.3 Petri Nets Operational Semantics with xMOF

Petri nets is a common formalism used to represent processes and distributed systems. We implemented the language as an xDSML for the GEMOC Studio using the xMOF language for defining the operational semantics. At runtime, we rely on the xMOF engine to execute a Petri net model, and we thereby automatically benefit from animation and standard debugging facilities.

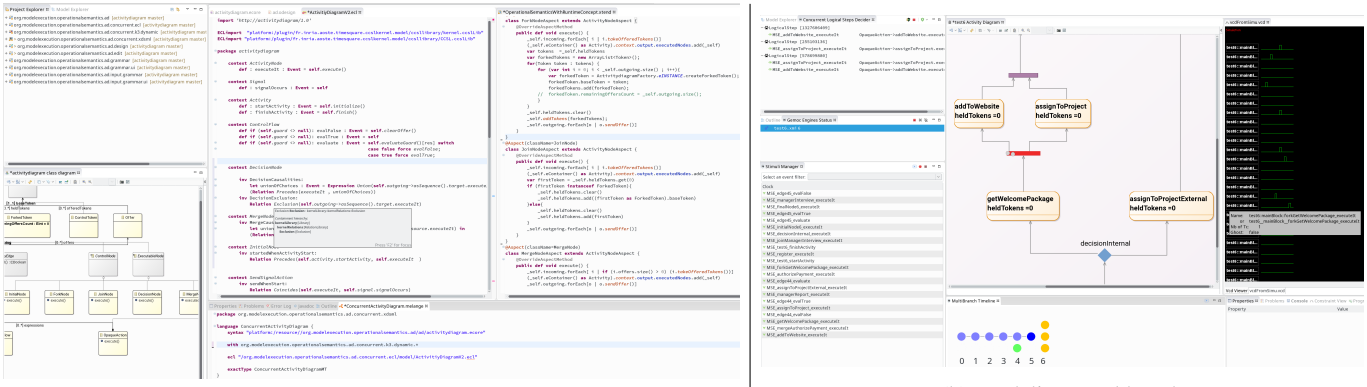
Figure 4 shows the language workbench with the Petri nets Ecore syntax and xMOF semantics (left), and the modeling workbench when executing and animating a Petri net model (left). During the demonstration, we will show that the process is very similar to using Kermeta only (see Section A.1), except that the operational semantics are defined only using xMOF. Therefore, we refer to the Section A.1 for more detailed information.

In the language workbench, we will define the metamodel of the abstract syntax using Ecore, then the operational semantics using the xMOF graphical editor, and finally the concrete standard using Sirius. Similarly to using the `@Step` annotation provided by Kermeta, we will use an annotation provided by xMOF to indicate which xMOF operations lead to execution steps.

In the modeling workbench, the deployed Petri nets xDSML will be used to define and execute a Petri net model. The debugger will be directly available to display the content of the stack, of the variables, and to animate the model.

A.4 Executing Arduino Designer in coordination with a Scenario by using BCOoL

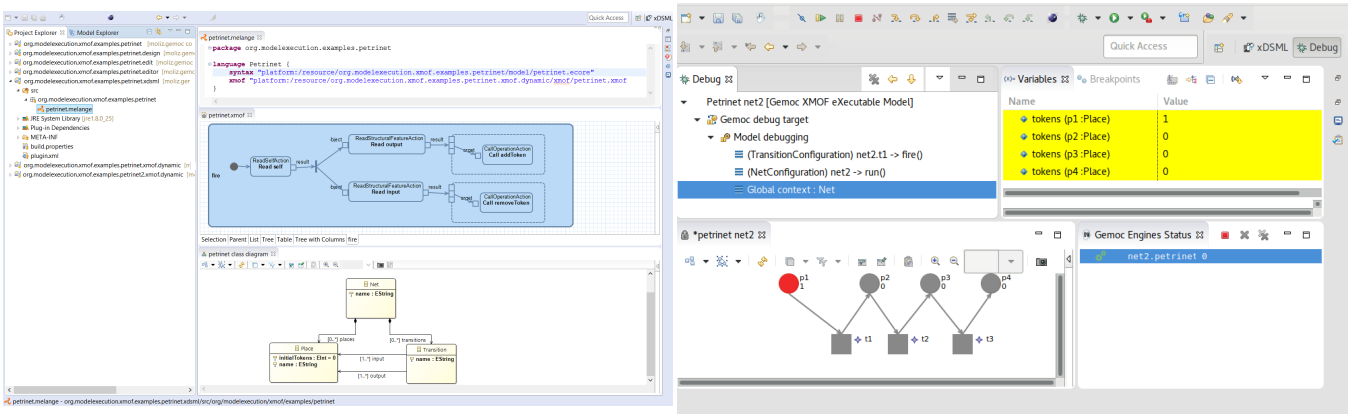
BCOoL is used to specify, at the language level, some coordinator patterns (Figure 5a), which can be applied on specific models to determinate their coordinated execution (Figure 5b). A pattern is expressed according to the concurrent operational semantics of some languages. For instance in Figure 5a, the pattern imports two languages, namely the Arduino language depicted previously and a simple scenario language. Based on the explicit domain specific events defined in these languages, BCOoL defines operators, which are defining how the models conforming such languages must be coordinated. In this case we specify the actions from the environment (the button push) by using the scenario language. Once the pattern applied, a coordination engine can execute the heterogeneous specification (Figure 5c), taking benefits of the animation, debugging and addons available for the concurrent engines.



(a) Language Workbench

(b) Modeling Workbench

Figure 3: Workbenches for (Kermeta/MoCCML)-based language design and model execution



(a) Language Workbench

(b) Modeling Workbench

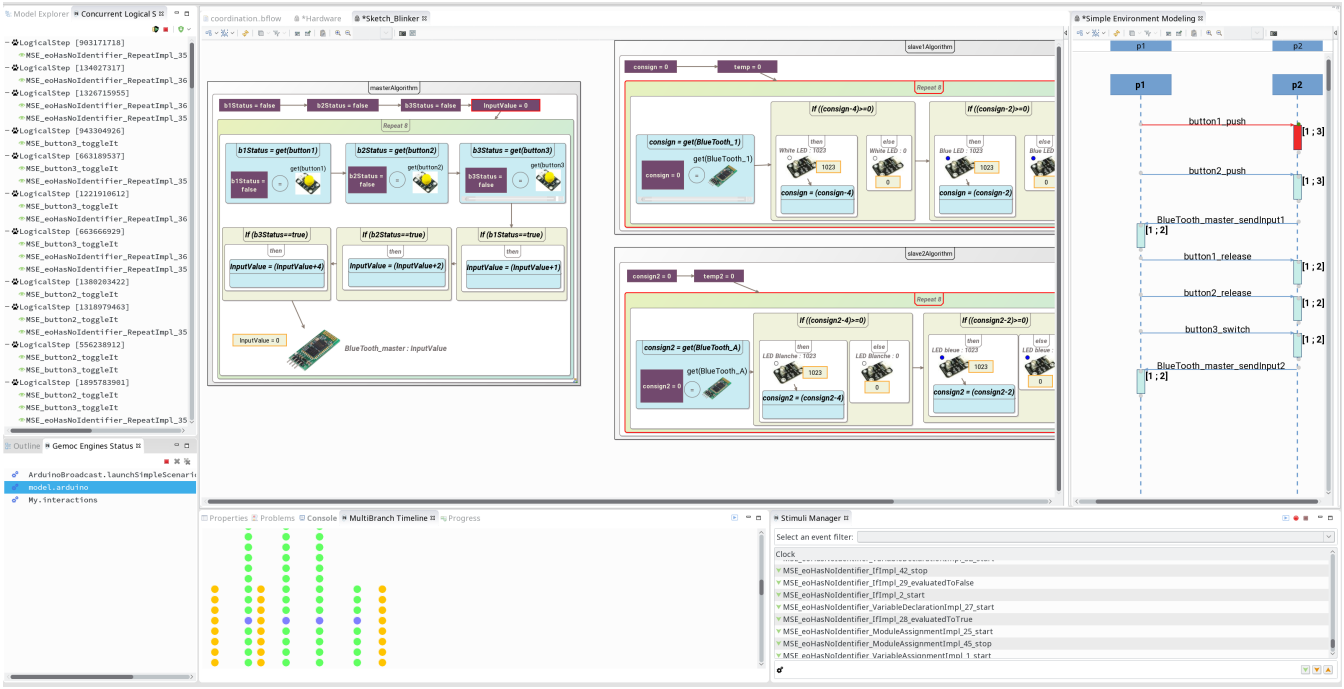
Figure 4: Workbenches for xMOF-based language design and model execution (example of Petri nets)

```
ImportLib "platform:/resource/org.gemoc.sample.bcool.arduinoml_scenario/operator/facilities.moccml"

ImportInterface "platform:/resource/org.gemoc.scenario.dse/ecl/scenario.ecl"  as SD
ImportInterface "platform:/resource/org.gemoc.arduino.concurrent.ecl/ecl/arduinoML.ecl" as AML

Spec one

Operator Sync_AML_SD_receiveAndToggle (dseReceive : SD::AnyReceivingOccurs, dseToggle : AML::toggleIt)
     MatchingCorrespondance: when "dseReceive.name.toString().startsWith(dseToggle.name)";
     CoordinationRule:
               facilities.RendezVous(dseReceive, dseToggle)
end operator;

Operator Sync_AML_SD_sendAndBluetoothPush (dseReceive2 : SD::AnyReceivingOccurs, dsePush : AML::send)
     MatchingCorrespondance: when "dseReceive2.name.toString().startsWith(dsePush.name)";
     CoordinationRule: facilities.RendezVous(dseReceive2, dsePush)
end operator;
```

(a) Language Workbench: coordination pattern definition

```
BCOoLFlow SimpleExample
{
        //the coordination pattern(s) to use
        import "platform:/plugin/org.gemoc.sample.bcool.arduinoml_scenario/operator/AML_SD.bcool";

        // Models on which to apply the specification
        Model aml_model ( "platform:/resource/org.gemoc.sample.arduino.concurrent.broadcastexample/model.arduino");
        Model sd_model ("platform:/resource/org.gemoc.sample.scenario.simpleexample/My.interactions");

        //application of the operators
        Flows
              apply Sync_AML_SD_receiveAndToggle on (sd_model, aml_model) ;
              apply Sync_AML_SD_sendAndBluetoothPush on (sd_model, aml_model) ;
        end Flows;



        // The launchers have information about how to launch the execution of individual models
        Launchers
                launcher URI "org.gemoc.sample.arduino.concurrent.broadcastexample/BroadcastExample.launch";
                launcher URI "org.gemoc.sample.scenario.simpleexample/My_interaction.launch";
        end Launchers
}
```

(b) Modeling Workbench: coordination pattern application



(c) Modeling Workbench: heterogeneous execution of an Arduino Designer model and Scenario model

Figure 5: Workbenches for BCOoL-based model coordination of Arduino Designer and a Scenario language