UML Model Execution via Code Generation

Gergely Dévai, Máté Karácsony, Boldizsár Németh, Róbert Kitlei, Tamás Kozsik
Eötvös Loránd University, Faculty of Informatics
Budapest, Hungary
Email: {deva,kmate,nboldi,kitlei,kto}@elte.hu

***Abstract*—Simulating design models makes early verification of the software's business logic possible. Model simulators can be implemented using an interpreter, but it provides limited runtime performance. This is only acceptable if the only use case of the tool is interactive model execution and debugging. If the model executor tool is to be used for automated regression testing, execution time becomes an important factor. In such cases generating code is a better option compared to interpretation.**

This paper documents our experience from an ongoing project which supports both the interactive and the automated model simulation use cases via code generation. It discusses a handful of architectural and technical questions to be solved in this setup and reports on a high performance open source UML model simulator.

I. Introduction

The two main use cases of a model simulator are (1) interactive model execution and debugging and (2) automated mass test case execution. In the first case, the tool has to

- provide graphical user interface with animation of certain diagrams (e.g. state machines),
- show object instances and their attribute values,
- support breakpoints both on graphical model elements (e.g. states and transitions) and in textual action language,
- provide usual debugging features like stopping on break points, stepping, resuming the execution etc.

In the automated test execution use case the simulator is used as part of a testing framework: The model is exercised by a set of predefined test cases in a nightly regression testing session or before each commit to the software repository. In this case, the simulator

- has to be fast, and
- has to provide a command line user interface.

Whenever a test case fails during the automated testing session, it should be possible to examine the cause of the failure in an interactive session. One possibility to achieve this is deterministic model execution: The simulator guarantees to walk through the exact same execution path each time for a fixed input. However, UML models running with several instances of active classes can be non-deterministic. A deterministic model simulator chooses only one execution path of the many possible. This severely limits the testing capabilities of the simulator. A better approach is to choose randomly from the possible paths, and make the tool generate execution traces during automated testing, which can be replayed in an interactive session.

Section III will show measurement results on the runtime performance of model execution via interpretation and code generation. The results make it clear that the requirement about high runtime performance leads to the code generation solution. In case of model execution via code generation a natural question arises: What is the difference between *model execution (simulation) via code generation* and *model compilation*? While they seem to be similar at first sight, they are highly different, due to their different purposes:

- Model simulation has to follow as many legal execution paths during intense testing as possible, to reveal possible errors. Model compilers have to generate code that takes one legal execution path and is as performant as possible.
- Model simulation has to check if invariants of the model (e.g. multiplicity constraints) are kept at runtime. The code generated by a model compiler will perform no runtime checks, or just a limited amount, in order to meet the performance requirements.
- The platform of a model simulator is the one where models are developed. Model compilation targets a given platform, independent of the development one, and takes its specifics into account.
- Model simulation has to connect to the debug framework, while model compilation has no such obligation.
- Model simulation has to be prepared to replay execution traces. Model compilers have to support logging only.
- Model simulation can be a single threaded emulation of a concurrent model. Model compilers have to emit parallel code, if that is required by the target platform.
- Model simulation has to provide quick feedback about the correctness of the business logic captured in the model. Results of the model compiler are deployed on the target platform, which can be time consuming, and the runtime results may include platform-specific errors.

The paper is organized as follows. The next section presents related work. Section III shows the results of an experiment comparing the runtime performance of model execution via interpretation, code generation and just-in-time compilation. An architecture for a code generation based model simulator is proposed in Section IV, with special attention on the animation and breakpoint support, the necessary debug symbols, the debug interface of the target language, then communication between the debugging front-end and the generated code, and, finally, the termination of the execution. Section V reports on the current status and capabilities of the tool which is built along the architecture discussed in this paper. The last section concludes the paper with a short summary.

II. Related work

In this section we give an overview of open source tools that can simulate UML (or UML like models). The richest set of model elements is supported by the BridgePoint tool [12], open sourced [2] in 2014. It is originally based on the Shlaer-Mellor method [15]. The tool is Eclipse-based. It consists of UML diagram editors, a model simulator and model compilers. The simulator animates state machines, supports breakpoints and provides other standard debugging features. BridgePoint uses an interpreter to execute models. Our experience shows that its runtime performance is enough for the interactive use case, but it needs improvement in the mass test execution use case.

Foundational UML, or fUML for short [10], is a standard defining formal execution semantics for a subset of UML. The goal of fUML is to be a basis for defining precise semantics for richer UML subsets (like the PSCS [11] standard). A Java-like textual syntax for fUML, called Alf [9] is also standardized. There are fUML and Alf reference implementations available. The fUML reference implementation is written in Java, and follows closely the formal semantics definition of the standard. The Alf reference implementation is integrated in the Eclipse environment using Xtext for parsing and OCL [16] for semantic checks. Alf execution is provided by transformation back to fUML activities and leveraging the fUML implementation. The efficiency of model execution was ignored in these implementations.

The above mentioned reference implementations themselves do not provide model debugging or animation support. Moka [8] is an extension to the Papyrus UML editor [13]. It simulates UML activity diagrams and provides basic debugging support, such as breakpoints on actions. Moka uses the diagrams of Papyrus as graphical front-end for simulation and debugging, and uses a modified version of the fUML reference implementation for the execution logic. Moka also provides an interface which allows the definition of new execution engines. We also use this technology to inegrate our work with Papyrus, see section IV for the details.

Moliz [7] is a testing and debugging framework for fUML activities. It defines a test specification language, and extends the fUML reference implementation with debugging and tracing capabilities. The execution traces are used to decide if a given test case passes or fails. Since Moka and Moliz use the fUML reference implementation, the performance limitations discussed earlier also apply to these projects, questioning their scalability in the mass test execution use case.

Topcased [14] is a set of plugins for the Eclipse platform, which is mainly aimed at the implementation of critical embedded systems. It uses Papyrus for model editing, and adds simulation capabilities. Topcased also provides visual model simulation for state machines, but no breakpoint support is implemented, neither have we found any way to use it for automated testing. The project is discontinued and is under migration to the PolarSys consortium [17]; however, the model execution capabilities of Topcased are not planned to be migrated. The long-term plan is to use Moka for that purpose.

Regarding the closed source commercial products, the Rational Rhapsody tool [3] also needs to be mentioned as it provides model execution capabilities.

III. Interpretation, code generation and JIT

An interpreter stores the internal state of an executed model (object instances, values of their attributes and actual state machine states, message queues of signals etc.) as data. It queries the model to find out the next action to take and changes the model execution state accordingly. Another possibility to achieve model execution is to compile the model to program code, then build and run it. It is also possible to combine interpretation and compilation using just-in-time-compilation (JIT). In this case the model is executed by an interpreter, but frequently executed or critically slow fragments are compiled, built and loaded into the process of the interpreter.

In order to compare the performance of the three discussed options, we have created an experiment with models limited to a single state machine. A predefined number of instances of the state machine is created and a given number of signals are sent to each of them. During the triggered a transitions, action code snippets (assignments, conditionals and basic arithmetic) are executed in the transition effect, state entry and exit.

In the compilation cases, state machine logic is implemented via nested switch-case statements. The interpreter uses an event matrix to look up the next state using current state and the received event. In case of JIT, the action code statements are compiled and the state machine logic itself remains interpreted. Two variants of the this solution have been implemented: one that generates Java source code and uses the Java compiler to compile it to bytecode, while the other one generates bytecode directly using the Javassist library [6].

Table 1 shows execution times in milliseconds of the four different implementations executing the same models with 1000 and 5000 instances and processing 10000 signals per instance. Two different machines/platforms have been used to increase confidence in the experiment: Configuration C_1 denotes a HP EliteBook8540w laptop with Intel Core i5 CPU @ 2.53 GHz and 4 GB RAM, running 64 bit Windows 7. C_2 is a HP EliteBook 9480m machine with Intel Core i5 CPU @ 2 GHz and 8 GB RAM, also running 64 bit Windows 7.

Technology	1000 instances		5000 instances	
	C_1	C_2	C_1	C_2
Interpreter	11436	7687	69288	41192
Generated code	496	354	2849	2339
JIT to Java	7295	4986	29796	13134
JIT to bytecode	1468	1332	23583	10060

Table 1: Execution times (ms) of interpretation, generated code and JIT

The results show that, in this experiment, interpretation turned out to be 18-24 times slower than running generated

code. The different versions of JIT compilation are better, but still 3-14 times slower than generated code. (We have carried out further experiments, with very similar results.) This means that designers of high-performance model execution engines targeting mass test execution have to seriously consider code generation as the technology to build upon.

IV. Architecture

This section discusses various aspects of an architecture that supports code generation based model execution.

A. Overview

Figure 2 shows an overview of the architecture of the tool this experience report is based on. The figure is color-coded: blue means third party components we build on, while the orange elements were created to make model execution via code generation possible.

In this setup, the *Papyrus editor* is used to edit the UML models, which are stored over the *EMF-UML2* [1] meta model. This is the input of our *code generator*, which translates the model to *Java*. The translation needs to be incremental in order to have the generated code ready for execution whenever the user requests execution in the interactive use case. The generated code is compiled (also incrementally) by the standard *Eclipse Java tooling*.

When model execution or debug session is started by the user, the compiled code is loaded into a newly created *JVM*. This Java process is the back-end of the session, managed through the *Java Debug Interface (JDI)*. The front-end (i.e. debug controls, animation etc.) is realized on one hand by the *Eclipse Debug Framework* (the standard debug tools), and on the other hand by *Moka* (the graphical model debugging and animation functions). In order to connect the front-end with the back-end, a *connection layer* is needed which relays the debugging and animation events in both directions.

In order to find the connection between the running Java code in the back-end and the animated model in the front, a set of *debug symbols* are used, which had been created by the model translation process in addition to the generated Java code. Section IV-C gives further details on these symbols.

B. Animation and breakpoint support

While the model is being executed, the user should be able to open up state machine diagrams corresponding to selected object instances, and see the current state of the instance or the active transition highlighted. In order to realize this, information about entering into states and fired transitions is needed from the back end. Similarly, if the user places breakpoints on states, transitions or on lines of action code in the model, we need to know when does the Java code in the back-end reach these particular points of execution.

We realize both animation and breakpoint support using Java breakpoints. Each breakpoint in the model is mapped to breakpoints in the generated Java code. When a state machine diagram needs to be animated, Java breakpoints are placed on the code lines corresponding to entering into states and

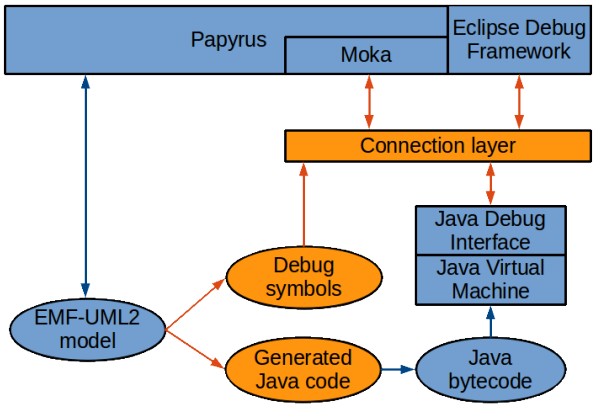

Fig. 2: Architecture overview

triggering transitions. These breakpoints are managed through the Java Debug Interface (JDI). Whenever a breakpoint is hit in the Java code, the execution is stopped and a notification is delivered from the JDI. If the breakpoint being hit corresponds to a breakpoint in the model, the user is notified and the runtime waits for user action (e.g. stepping or resuming the execution). If the breakpoint was created for animation purposes, the corresponding state or transition is highlighted for a given amount of time, and then the execution of the Java code is resumed via the JDI.

Note that creating Java breakpoints for animation purposes might result in a high amount of breakpoints managed (created, disabled, enabled, removed) by the JDI. In order to find the right policies that ensure the scalability of the solution, we have designed experiments to test the performance of the JDI, see Section IV-D for the results.

C. Debug symbols and mappings

During compilation of Java class files, the compiler inserts the information needed to find the line of source code corresponding to a given instruction. However, interactive model debugging should support highlighting the actual state in running state machines or stepping through them as well as stepping over lines of action code or highlighting them. To provide these debugging features, a mapping between model elements and their generated Java source code is needed.

This problem is partially solved by JSR 45: Debugging Support for Other Languages [5]. It provides a standard way for correlating Java byte code with source code of textual programming languages other than Java. JSR 45 uses a data structure called Source Map (SMAP) to specify the mapping between the lines of code in the original source language (which could be for instance the action language of the modeling system) and the generated Java source code. These source maps are injected into the binary class files using the `SourceDebugExtension` attribute after their compilation.

The Java Debug Interface can then be configured to use the given source mapping while accepting and reporting breakpoint and other location information during a debug session. A single class file could have multiple attached source mapping information for multiple source languages. Each Source Map has a name, and the debugger will use this identifier to select the appropriate mapping.

Unfortunately, this source mapping facility works only for textual languages. However, for ordinary model elements, like states and transitions which are represented mostly graphically on the user interface, a virtual line mapping can be provided. For non-textual elements, the Source Map will contain virtual line numbers, and another mapping will be used to resolve these into the original UML elements. Like the Source Map, this data can also be created during Java code generation. While it is also possible to store this mapping from virtual line numbers to UML elements directly inside a class file using a custom attribute, in the tool presented in this paper it is currently serialized into a separate binary file and loaded by the debugger itself.

Because the naming rules of UML allow element names which are not valid as Java identifiers, the code generator assigns a unique Java identifier to each UML named element that has a corresponding Java element. Names in the action code are also provided with unique identifiers. The resulting identifier-to-name mapping is stored along with the virtual line mapping to let the development environment show the original names of model elements during debugging.

D. Using the Java Debug Interface

The Java Debug Interface (JDI) defines a high-level Java interface to access debugging capabilities of a running Java virtual machine. It is the front-end part of the Java Platform Debugger Architecture (JPDA) [4]. Eclipse Java development tools (JDT) also uses this technology to implement its debugger. As JDI supports the inspection and manipulation of the connected virtual machine's state through a simple API, it is a convenient choice to provide interactive visual debugging for UML models. The API is provided under the `com.sun.jdi` package, bundled with JDK distributions.

JDI provides the following four ways to connect a debugger with a target process:

1) Debugger launches target
2) Debugger attaches to running target
3) Target attaches to running debugger
4) Target launches debugger

The default Java process launching mechanism of Eclipse JDT uses the third option, as it creates a listening connector and then launches the target process with special command line options. These options are commanding the target virtual machine to connect to the debugger process through a socket. From this point, we can use a VirtualMachine object in the debugger to send commands to and receive events from the target virtual machine. This mechanism of JDT could be entirely reused to start a model executor process. Only the underlying `IDebugTarget` instance should be replaced to a custom implementation. `IDebugTarget` is an abstraction defined by Eclipse Debug Framework (EDF), which is used to control a debugger from the user interface (like stepping, suspending and resuming), and to coordinate the presentation of the data fetched from the target process (for instance threads, stack frames and local variables).

Communication with the target virtual machine is asynchronous. The debugger can set different kind of event requests on a machine, for instance to get notification about a breakpoint hit at a specified location. The virtual machine will provide events in an event queue, corresponding to event requests. Each request could be enabled or disabled, and has a suspend policy. Based on the suspend policy of the corresponding request, the processing of an event could suspend all threads of the target machine. It also could suspend only the source thread of the event, or continue its work without interruption. When the machine is suspended by an event, the execution could be resumed manually. Events are delivered in sets by the event queue, as several events could be fired by a virtual machine at the same time.

To place a breakpoint in the model execution process, the debugger creates a breakpoint event request. This request contains only the location of the given breakpoint. The calculation of this location involves the usage of debug symbols and mappings presented in the previous section. For example, when a breakpoint is placed on the entry of a state in a state machine, the location resolution will follow the next steps:

1) Calculate the fully qualified name of the Java class which contains the code for the given state
2) Fetch the virtual line number of the given state entry
3) Get a reference to the class from the target virtual machine
4) Use the class reference and the virtual line number to get a JDI location

As a Source Map is installed on the generated class, the mapping between virtual line numbers and lines of the generated Java source code is available. It makes it possible to resolve the virtual line number to a bytecode offset inside a method. Before resolving, the debugger can select which Source Map to use from the available ones using the `setDefaultStratum` method on the `VirtualMachine` instance.

The target machine, once it hits a breakpoint, emits a breakpoint event through its event queue. The breakpoint event contains the JDI location of the breakpoint. The location specifies a virtual line number and a reference to the containing class. The resolution to a model element can be done in a similar way as in the previous case, using the debug symbols saved for the class earlier at code generation time.

When a breakpoint event suspended the virtual machine, its internal state can be inspected, or even modified. For example, Java local variables can be mapped to variables in action code of the modeling language, and presented to the user. It is also possible to implement expression evaluation, and to alter the value of a variable.

The `VirtualMachine` object can also be used to terminate the target machine or to disconnect from it, enabling

another debugger to attach to the target.

Models that have to be handled by the model simulator may contain a large number of model elements, therefore they induce a large number of potential breakpoint locations. We have conducted an experiment to determine how the number of breakpoints affects performance. The experiment was run on an Intel Core i7 CPU @ 3.4 GHz and 8 GB RAM, running 64 bit Windows 8.

BPs	10	100	1000	10,000	20,000
Passes	5000	500	50	1	1
BP hits	50000	50000	50000	10000	20000
Run	4464	5126	12175	16952	69482
Run/100	8.928	10.25	24.35	169.52	347.41
Set			112	1602	4965
Disable			66	1750	5753

Table 3: Execution times (ms) of breakpoint scenarios

The test cases in Table 3 set individual breakpoints on statements (which, in the experiment, simply increase a counter) on separate lines. The row *BPs* shows the number of breakpoints; the test cases loop over them *Passes* times for a total of *BP hits* breakpoint hits. When the execution of the virtual machine hits a breakpoint, execution is handed over to the testing application.

The tests show that the execution time is quite dependent on the number of breakpoints: for the same number of total breakpoint hits (50,000), the execution time nearly doubles when using 1000 breakpoints instead of only 10. If we increase the number of breakpoints further, the degradation is even more apparent: the average time to hit 100 breakpoints (*Run/100* row) doubles between 10,000 and 20,000 breakpoints.

Furthermore, breakpoints have to be set before execution can progress toward them. Setting a low number (less than 1000) breakpoints is almost instantaneous and not shown in the table, however, as the number of breakpoints increase, we see that the cost of simply setting them approaches 10% of passing through all of them.

These results show that a visual model debugger that is using generated code for execution must limit the number breakpoints it uses. Fortunately, the model debugger needs to keep breakpoints only on lines that correspond to model elements visible to the user (for animation) and model breakpoints set by the user. To further optimize performance, the debugging environment can disable breakpoints that are likely to be used again instead of removing them.

We have taken into consideration other methods of communication with the runtime. Because Java technology is used on both sides of the communication, using virtual memory is not favourable. Sockets and files could be used to transmit information, but the debugger provides a higher-level interface.

E. Terminating the execution

Model execution can terminate two ways. Either the model terminates normally (active object instances get to their final states or get deleted) or the user stops the execution using the debugging controls on the user interface. The latter one, premature termination, has to be enabled, because certain models take a long time to terminate or might not terminate at all.

Two different kinds of non-termination can be identified. Either the state transitions of a state machine or one of the action code blocks of the model may contain an infinite loop. The non-termination of a state machine is easier to handle, because control is given back to the runtime at least once in every cycle.

User initiated termination of the execution can be done in different ways. First, interrupting the JVM is the simplest solution, however in general, it prevents the runtime from freeing its allocated resources, like log files written. On a Unix system, sending a kill signal to the JVM process enables it to run its resource deallocation code, but this is not a platform-independent way of stopping the virtual machine.

The other way to stop the program is to somehow communicate with it to stop its execution. It could be possible to use the debugger to this purpose, but our goal was to stop the virtual machine regardless if it has an attached debugger or not. For this, we have implemented an alternative way of communication with the runtime using sockets. The development environment can send a terminate message to the runtime when the user decides to stop the execution. When the runtime is initialized, a control thread is started to receive control messages like the terminate message over the socket connection with the environment. When it receives the terminate message it closes all open resources, and finally terminates the JVM.

V. Tool implementation

We are in the process of developing a proof-of-concept model executor tool [18] which we are releasing on a monthly basis. The tool is usable in two ways: via an interactive GUI integrated with Eclipse, and via a command line interface. This section gives an overview the capabilities of the two as of the time of the writing of this paper.

The interactive user interface is used to visually inspect the execution of a model. It uses the Moka extension of the Papyrus framework to display and edit the model elements. The tool currently supports a reduced set of UML features: only the state machine of a single instance can be executed, and only flat state machines are supported. Classes can have attributes, operations and associations with multiplicities. The state machine of the class is driven using signal events: they can be sent from action code in the model to the running instance and to external Java code, and external Java code can also generate signal events. The execution engine at each step dispatches the next available signal, and then progresses to the appropriate state. Action code is executed in a run-to-completion manner.

Visually, the most eye-catching feature of the tool is its capability to visualise model execution. Once a model is loaded, the user has to make a debug configuration, where

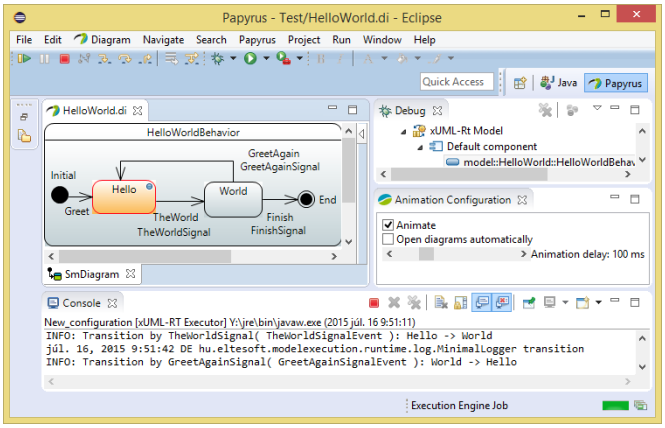

Fig. 4: The GUI of the tool

he chooses the class that will be instantiated, and a `feed` function that drives the model by generating events. When execution is started, the active state is visually highlighted on the model. The user can make execution progress to the next state manually, but by default animation is turned on: the transitions happen after a set time amount. The user can set breakpoints on the states and transitions, and when one is hit, execution is paused. The tool can also log transitions, and store execution traces which can later be replayed, producing the same execution.

Figure 4 shows a model being executed, stopped at a breakpoint on state `Hello`. As logging is turned on by default, the bottom of the screen shows the most recent state transition log messages. As execution is paused, the user can conveniently turn off the animation option in the *Animation Configuration* view if he so chooses, and then continue execution with the *Suspend* toolbar button, the leftmost one depicted. The *Debug* view must be open and the executing state machine selected for the *Suspend* button to be enabled; while it is possible to have several models executing at the same time, it is usually hard to follow and inadvisable, as they have to use the same display.

The command line tool is used to automate test case execution. It uses the same execution engine as the GUI tool, therefore it expects similar arguments: model, class, feed function, and path settings for source and generated files, logs and traces. Figure 5 shows a sample output of the command line tool after having run the same model as Figure 4. Options about where to place the generated files and how detailed the log messages should be are explicitly visible here; they are also available on the GUI tool, set to reasonable defaults.

VI. Summary

In this paper we have analysed technical aspects of code generation based UML model simulation. The main motivation for this solution is higher runtime performance, required by non-interactive model-level testing use cases.

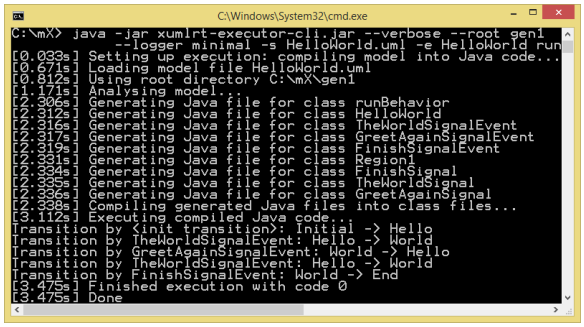

Fig. 5: The command line interface of the tool

The technical content of the paper is based on the design and implementation of a model simulator in industrial cooperation. The goal of the paper is to share the experience we gained from this project with the community working with executable modeling tools.

Acknowledgment

We express our gratitude to Ericsson for the financial support of this research.

References

[1] EMF-UML2 project. http://wiki.eclipse.org/MDT-UML2.
[2] Executable Translatable UML Open Source Editor. https://www.xtuml.org/.
[3] IBM. Rational Rhapsody family. www.ibm.com/software/products/en/ratirhapfami.
[4] Java Platform Debugger Architecture (JPDA). http://docs.oracle.com/javase/8/docs/technotes/guides/jpda/.
[5] Java Specification Requests 45: Debugging Support for Other Languages. https://jcp.org/en/jsr/detail?id=45.
[6] Javassist library. http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/.
[7] Stefan Mijatov, Philip Langer, Tanja Mayerhofer, and Gerti Kappel. A Framework for Testing UML Activities Based on fUML. In *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa) co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, pages 1–10, 2013.
[8] Moka. http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.
[9] Object Management Group. Action Language for Foundational UML (ALF), standard, version 1.0.1. http://www.omg.org/spec/ALF/, 2013.
[10] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), standard, version 1.1. http://www.omg.org/spec/FUML/1.1/, 2013.
[11] Object Management Group. Precise Semantics of UML Composite Structures (PSCS), standard in preparation, version 1.0 beta 1. http://www.omg.org/spec/PSCS/1.0/Beta1/, 2014.
[12] OneFact. BridgePoint xtUML tool. http://onefact.net/.
[13] Papyrus. http://wiki.eclipse.org/Papyrus.
[14] Nadege Pontisso and David Chemouil. Topcased combining formal methods with model-driven engineering. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 359–360. IEEE, 2006.
[15] Sally Shlaer and Stephen J. Mellor. The Shlaer-Mellor method. *Project Technology white paper*, 1996.
[16] Technical Committee ISO/IEC JTC1, Information technology, in collaboration with the Object Management Group (OMG). Object Constraint Language (OCL). Standard, International Organization for Standardization, Geneva, Switzerland, April 2012.
[17] Topcased migrates to PolarSys. http://polarsys.org/topcased-migrates-polarsys.
[18] xUML-RT Model Executor. http://modelexecution.eltesoft.hu/.