Evaluation of Open Source Model Interpreters

István Gansperger[1], Máté Karácsony[2], Róbert Kitlei[1], Gábor Ferenc Kovács[1], Boldizsár Németh[1], Tamás Kozsik[1], Sándor Sike[1], Ádám Tarcsi[1], Gergely Dévai[2]

[1]ELTE-Soft Nonprofit Ltd.

[2]Ericsson

December, 2014

1 Contents

1 Introduction

This document evaluates available open source model interpreters based on their specific features, deficiencies, compatibility with other tools and modeling standards and their compliance level to the Ericsson requirements against the future model level execution solution. Some specifics about their implementations are also highlighted and code or solution reuse possibilities are discussed.

2 Open Source Model Interpreters

1.1 ALF and fUML reference implementations

1.1.1 Introduction

ALF is an action language with concrete syntax for foundational UML (fUML), which specifies the semantics for an executable subset of UML. The reference implementations of these specifications are written as interpreters in Java and can be downloaded from mds artifact repository.

1.1.2 Specifics

The following are true for both interpreters:

- Standalone command line application
- Single-threaded execution
- Reading models from textual representation, even from multiple files
- Generating textual execution traces

- They provide the maximal level of conformance for each specification (L3 for fUML and extended conformance for ALF)
- Very accurate, low level simulation of the execution semantics

1.1.3 Deficiencies

I was unable to start the standalone fUML interpreter with a custom model file, because usually a NullPointerException was thrown. However, the ALF interpreter seems to work correctly, and it uses the same execution engine, so probably the model files were wrong. As this is a single-threaded, semantically accurate implementation, it may have performance problems with huge models. There is no visual feedback during the interpretation, only an execution trace is generated.

1.1.4 Compatibility

Both interpreters are using standard formats, namely XMI and ALF files. This provides great interoperability with other tools.

1.1.5 Feature compliance

As the ALF interpreter uses fUML interpreter for execution, only the latter is analyzed in the table below.

Requirement	fUML reference implementation
High Priority Requirements	
Model interpretation	Yes. Simulates semantics according to the specification.
Interactivity	No. The execution engine cannot be influenced directly, but when an activity reads or writes the standard input or output streams, it is handled correctly.
Mass execution	Yes. Possible by starting independent interpreter processes e.g. from a batch file.
Scalability, performance	The implementation serializes all activity to a single thread. As lot of objects are created and destroyed during activity simulations, it is expected to perform poorly.
Visual feedback	No. Only a textual execution trace is generated.
Test coverage statistics	No direct coverage on states or action language, but it may be able to be restored from the execution trace.
Deterministic & nondeterministic mode	Only deterministic mode is available, and the execution order cannot be controlled. However, the built-in strategies can easily be changed in the source code.
Timers	No, because there are no timed events in fUML at all.

Connection to native code	No
Tracing	Yes, as it is the primary output of the interpretation.
Medium Priority Requirements	
Model debugging session	No
Test and production model elements	No
Possibility to influence simulation speed	No
Low Priority Requirements	
Conditional breakpoints	No
Profiling: number of running instances etc.	No
Connection to debuggers of implementation languages	No
Persisting/loading model execution state	No

1.1.6 Architecture

The ALF interpreter uses JavaCC for parsing. It contains the whole fUML and ALF description in XMI format, and also as Java interfaces. Mapping from ALF abstract syntax to fUML behavioral models is also provided. Not only behavioral, but also structural fUML elements could be created in the input files as the interpreter implements the maximal, *extended conformance* level of ALF. The interpretation is done according to the following steps:

1. Parse input files in ALF concrete syntax into object instances of abstract syntax
2. Map the abstract syntax to the corresponding fUML representation
3. Execute the fUML interpreter on the resulting model

The fUML interpreter reads the models from XMI files using a streaming XML parser. Input elements are decoded into model objects and stored in a common object repository. The source tree includes the whole fUML specification as XMI files, with UML infrastructure, superstructure, and the foundational library. The interpretation is done according to the following steps:

1. Input files are parsed with a streaming XML parser
2. Decoded model elements are stored in the repository

3. When the selected class or behavior has been found in the repository, its execution starts
4. An execution queue is used to serialize all further activities
5. Simple built-in strategies are used to make deterministic choices where needed

The source code of both interpreters is well-organized and relatively easy to understand. Examples and test suites are also provided with each project. Build systems are based on Apache Ant.

1.1.7 Licensing

The ALF interpreter uses a GPLv3 license, but some of its dependencies are published under an Apache license. However, the fUML interpreter has more complicated licensing. For more information, see the following files:

- http://lib.modeldriven.org/MDLibrary/trunk/Applications/fUML-Reference-Implementation/trunk/Licensing-Information.txt
- http://lib.modeldriven.org/MDLibrary/trunk/Applications/Alf-Reference-Implementation/dist/LICENSING.txt

1.1.8 Reuse possibilities

These interpreters can be used to validate other implementations against them. A lot of model classes and interfaces can be reused from each code base. XMI models and ALF files are also reusable for testing purposes. It may be possible to evolve these solutions into multi-threaded implementations.

1.2 Moka

1.2.1 Introduction to Moka

Moka is an open source tool based on the Eclipse Modeling Framework (EMF) and distributed as an extra package for the Papyrus editor. It provides model execution with graphical animations for fUML actions. It is being developed by CEA (Commissariat à l'énergie atomique et aux énergies alternatives).

- Git repository of Moka at eclipse.org
- Short tutorial of Moka

1.2.2 Specifics of Moka

Moka allows the user to execute, debug and visualize Activity Diagrams.

- It allows breakpoints to be placed on actions, and the user can inspect the context when execution is suspended.
- Execution speed (how fast simulation proceeds to the next step) can be controlled via a slider.
- There are three modes of execution: one is single-threaded and fUML based, the other two are PSCS based (single-threaded and multi-threaded). According to the developers, the multi-threaded model uses one thread per active object.

1.2.3 Deficiencies

- In general, the system is quite slow. The Moka Execution Engine takes a couple of seconds to start up each time a debugging session is started, and execution speed is comparable to debugging a native Java program with a conditional breakpoint enabled.
- Visual input and editing of the source code is clumsy. It takes several clicks to select the name of a data type, even if the user knows exactly where to look. Moreover, search functionality in many input boxes is quite slow, taking several seconds to complete.
- In the current release, it is very hard to find out what sort of elements are placed on the Activity Diagrams. A fairly recent commit fixes this by adding a tooltip, and is expected to be part of the next release.
- Documentation is severely lacking.

1.2.4 Compatibility

Moka is based on Papyrus therefore it is Ecore-based UML2 compliant.

1.2.5 Feature compliance

Requirement	Moka
High Priority Requirements	
Model interpretation	Yes.
Interactivity	Yes, breakpoints on actions in Activity diagrams, and possibly others, too.
Mass execution	Probably possible to batch execute test cases from command line (probably using org.eclipse.papyrus.moka.debug.MokaProcess), but primarily designed for interactive use.
Scalability, performance	Multi-threaded mode (PSCS -- Precise Semantics of UML Composite Structures, an extension of fUML) runs a thread for each object; can probably scale up to 1000s of objects. Not measured yet.
Visual feedback	GUI diagrams with animations: Activity, State, possibly more.
Test coverage statistics	No support.
Deterministic & nondeterministic	Two single threaded modes (fUML and PSCS) and one PSCS multithreaded mode. Execution within active objects is deterministic.

mode	
Timers	The animation speed can be changed. Also, there is a TimeObservation in the class diagram, and transitions can be guarded by various conditions with time related names (Duration, DurationInterval, TimeExpression, TimeInterval).
Connection to native code	The multithreaded PSCS engine is implemented in org.eclipse.papyrus.moka.async.fuml.FUMLAsyncExecutionEngine, and the single threaded one in org.eclipse.papyrus.moka.composites.CompositeStructuresExecutionEngine. The former subclasses the latter; they make heavy use of the packages org.eclipse.papyrus.moka.async.fuml.Semantics, org.eclipse.papyrus.moka.async.fuml.Semantics and org.eclipse.papyrus.moka.fuml.Semantics, and they provide their functionality in Java. The mechanism, if replaced by a suitable implementation, would most probably support execution in other languages. Also, when creating a FunctionBehavior, the choices C, C++, JAVA, Natural language, OCL and Alf are present.
Tracing	No apparent support by Moka, however, Papyrus can activate traces on various elements (actions, classes, operations etc.). Note that both Papyrus and Moka feature the concept of a breakpoint (and both can be set on an action, for example), but they are not playing nice with each other: if only a Papyrus breakpoint is set, execution does not stop when it is reached.
Medium Priority Requirements	
Model debugging session	Signals, events, instances etc. cannot be added by hand once the session is started, but this might be possible using some code snippets (or maybe through sockets).
Test and production model elements	-
Possibility to influence simulation speed	No support.
Visual representation of the simulation	A slider can adjust the time (milliseconds) between animation steps.

Low Priority Requirements	
Conditional breakpoints	No, conditions on breakpoints (other than them being on/off) are not supported, not even a hit counter.
Profiling: number of running instances etc.	No support.
Connection to debuggers of implementation languages	No support.
Persisting/loading model execution state	No support.

1.2.6 Architecture

Moka uses the facilities of Papyrus to read the input model. It stores its runtime configuration as key-value pairs, and it registers a MokaProcess (a specialisation of an org.eclipse.debug.core.model.IProcess) in the system for a debug launch. It uses the GUI features of the Papyrus graphical elements to show any changes on the UI. According to one of the developers, action transitions are executed in the order that they are present in the model, even if the multi-threaded model is being used.

1.2.7 Licensing

Eclipse Public License 1.0 (EPL)

1.2.8 Reuse possibilities

Moka can be considered a reference implementation of the PSCS model (whose beta description is available here), and it shares its codebase with the fUML/Alf reference implementation. Since the full semantics of the model are implemented in the code, Moka can serve as a validator for our project, however, its general structure is probably too inefficient for us to closely follow. More directly usable is a middle layer of engine implemented in Moka, which serves as an event dispatcher between the debugger architecture in Eclipse and an execution engine. Since this is a thin layer, it is expected to be possible to reuse it with little or no modification. The way the debug target communicates to the execution engine, through sockets, should also be considered for reuse, and we can also examine how the Papyrus graphical elements are accessed and manipulated.

1.3 MoliZ

1.3.1 Introduction to MoliZ

Moliz is a tool for interpreting activity diagrams in Eclipse. It supports visual debugging.

Information about Moliz:

- Project homepage
- Moliz on Google Code
- Model testing example

1.3.2 Specifics of MoliZ

- Moliz has a sister project xMof, that is a metamodeling language integrating fUML with MOF

1.3.3 Deficiencies

- It crashes with NullPointerException on running any examples included.
- It is not documented at all.
- In practice the development stopped in 2013.

1.3.4 Compatibility

Moliz converts the Ecore UML2 models into fUML to execute them.

1.3.5 Feature compliance

Requirement	MoliZ
High Priority Requirements	
Model interpretation	Yes
Interactivity	Yes, breakpoints are supported
Mass execution	No (this is delegated to xMOF)
Scalability, performance	Unknown
Visual feedback	Yes, it does highlight the active action.
Test coverage statistics	No
Deterministic & nondeterministic mode	Unknown
Timers	Unknown
Connection to native code	Unknown
Tracing	It has a console logger
Medium Priority Requirements	
Model debugging session	Unknown

Test and production model elements	No
Possibility to influence simulation speed	No
Visual representation of the simulation	Unknown
Low Priority Requirements	
Conditional breakpoints	Unknown
Profiling: number of running instances etc.	No
Connection to debuggers of implementation languages	No
Persisting/loading model execution state	Unknown

1.3.6 Architecture

It has its own Ecore-based model representation that closely resembles the representation in the reference implementation. The representation of the model elements and the execution of them are separated. The execution engine also seems to be an improved version of the one implemented in the reference implementation. It does represent every value in a dynamically typed way. The execution of the model is sequential and deterministic.

1.3.7 Licensing

MoliZ is published under the Eclipse Public License 1.0.

1.3.8 Reuse possibilities

We could copy parts of the representation of the fUML models (org.modelexecution.fuml project) and change them to our needs. We could also reuse code from the execution of the model (org.modelexecution.fumldebug.core project) if we implement a classic interpreter. If we reuse code from this project we should also check what changed compared to the reference implementation.

Some useful architectural patterns in the code are:

- Running the interpreter generates execution traces. Test cases are evaluated as passing or failing based on their traces.

1.4 Topcased

1.4.1 Introduction to Topcased

Topcased is a set of plugins for the Eclipse platform that is mainly aimed at the realization of critical embedded systems. It uses Papyrus for model editing and adds a simulation workflow on top of the

models. The project is now under migration to the PolarSys platform which does not view model interpretation a first priority goal. Their plan is to use Moka for that purpose in the long run. The relevant part of the software seems to be abandoned and lacks documentation.

- Official Topcased page
- Mailing list

1.4.2 Specifics of Topcased

Topcased was rather ambitious in setting its goals but the project could not realize many of the planned features. The user interface exposes unimplemented functionalities and bugs are not rare to come along either.

- It uses the Papyrus editor so the software is Ecore-based UML2 compliant.
- Single-threaded execution
- Signals are kept in the target-object's event queue and are processed sequentially
- Easy to follow simulation, standard debugger functionality (but no breakpoints)
- Mainly focused on using visual representation but the models can be written by hand (for example as Java code)
- Uses visual activity diagrams for implementing program logic

1.4.3 Deficiencies

- Topcased's main drawback is that it is only usable for toy-sized projects. Under heavy or even medium load it breaks very quickly.
- It only has visual simulation, the applications cannot be run without the UI or the Eclipse platform (though the visual representation of the models can be turned off which might reduce the UI overhead.
- The simulation does not offer any way of testing tracing or reasonable debugging; one can only step through the states of the objects sending signals and triggering timers.
- Although there is an option to adjust the time when starting the simulation this feature does not seem to be implemented.
- Simulations cannot be automated: init signals need to be included by hand.
- Signals cannot be easily sent from outside of Topcased (it's possible to create a simple bridge as Eclipse plugin).
- There is no way to interface with different languages.
- The UML save and replay functionality is not implemented but is present on the UI.

1.4.4 Compatibility

Topcased is based on Papyurs therefore it is Ecore-based UML2 compliant.

1.4.5 Feature compliance

Requirement	Topcased

High Priority Requirements	
Model interpretation	Yes
Interactivity	Yes. Standard debugger stepping features. Can trigger events and timers. No breakpoints.
Mass execution	Only possible by running multiple Eclipse instances which is highly impractical.
Scalability, performance	Doesn't scale beyond 20 active instances.
Visual feedback	Activity and state charts are visualized.
Test coverage statistics	There are no means to run tests within Topcased.
Deterministic & nondeterministic mode	Single-threaded model so only deterministic mode. Signals are sent in order and processed from a queue.
Timers	Time cannot be scaled but there are timers which can be triggered by hand.
Connection to native code	There is no way to interface with native code from Topcased. It is possible to send signals from the outside but it is not within Topcased's scope and requires extra effort.
Tracing	There should be a feature to trace and log the simulation but it is not implemented.
Medium Priority Requirements	
Model debugging session	Yes, it's possible to affect the execution from the debugger to some extent.
Test and production model elements	No
Possibility to influence simulation speed	No
Visual representation of the simulation	Yes
Low Priority Requirements	
Conditional breakpoints	No breakpoints at all.

Profiling: number of running instances etc.	No support, but other Eclipse plugins may be used
Connection to debuggers of implementation languages	No support, but other Eclipse plugins may be used
Persisting/loading model execution state	Not implemented but present.

1.4.6 Architecture

The input for the simulation is a Papyrus model. The standard Eclipse debugging framework is used to interact with the model. The simulation will render state changes on the Papyrus model.

Topcased does not generate code, it has a simple, sequential interpreter written in Java embedded in Eclipse. It has a single-threaded model and uses Eclipse's JobManager class to schedule Jobs. When transitioning it will wait for the last Job to finish then overwrite it with the next step changing the state. There are separate jobs for each debugger command which operate on the state of the model. It uses the Eclipse debugging framework to implement this functionality.

1.4.7 Licensing

Eclipse Public License 1.0 (EPL)

1.4.8 Reuse possibilities

It may be possible to reuse the debugger architecture to some extent. It's very close to what we have envisioned apart from the missing breakpoints feature. Note however, that the way our project will communicate with Eclipse heavily depends on the method of execution (classic interpreter vs. code generation).

1.5 Yakindu

1.5.1 Introduction to Yakindu

Yakindu Statechart Tools (SCT) is a statechart diagram modeler for Eclipse. Can edit diagrams, interpret them and generate Java, C and C++ code from them.

Useful sources of information about Yakindu:

- Yakindu home page
- Yakindu on Google Code
- Yakindu User Group
- Blog of Andreas Mülder

1.5.2 Specifics of Yakindu

Yakindu has some interesting features that help editing diagrams. As Yakindu is restricted to statechart modeling, it is simpler and more user friendly than other products.

- A subdiagram is a statechart embedded in a single state.
- Support for defining orthogonal states
- Submachine states are subdiagrams reusable in multiple states
- Statechart refactorings to restructure diagrams
- Fully integrated content assist on both textual and graphical views

Yakindu uses cycle based execution semantics: Everything is synchronized to a global clock with an indivisible time unit. This is considerably different from event based semantics envisioned for executable UML.

1.5.3 Deficiencies

There are major drawbacks of using Yakindu that limit it's usability in an industrial development.

- Single purpose tool, can only model statecharts
- Cannot run multiple instances of an active object defined by a statechart
- Yakindu is not documented well. There is a tutorial that shows the basic functionality of the tool, but it can only be mastered by trial and error.
- In some corner cases the behavior is peculiar:
    - Triggering timing events can be done manually, but doing so does not make global time progress and therefore results in an incoherent state.
    - If more than one transition could happen, the firstly drawn edge will be taken.

1.5.4 Compatibility

The diagram/model cannot be imported from or exported to other formats. But it is an extension of the eclipse GMF format, so conversion tools would be relatively easy to implement.

1.5.5 Feature compliance

Requirement	Yakindu
High Priority Requirements	
Model interpretation	Yes
Interactivity	No (breakpoints are a promised feature)
Mass execution	No (testing is a promised feature)
Scalability, performance	NA (cannot run multiple instances, so it is not relevant)
Visual feedback	Yes. Statechart simulation is visualized
Test coverage statistics	No coverage on states or action language.
Deterministic & nondeterministic	Only deterministic mode is available, and the execution

mode	order cannot be controlled.
Timers	Time cannot be simulated. Timer events can be fired at any time, but this disrupts the timing of the whole model.
Connection to native code	Yes. Only Java is supported. Can call any function if it is declared as an operation.
Tracing	Yes. No explicit support for tracing, but can be done with operations.
Medium Priority Requirements	
Model debugging session	Yes, events can be manually fired and values altered.
Test and production model elements	No
Possibility to influence simulation speed	No
Visual representation of the simulation	Yes
Low Priority Requirements	
Conditional breakpoints	No (promised feature but not implemented yet)
Profiling: number of running instances etc.	No support, but other Eclipse plugins may be used
Connection to debuggers of implementation languages	No support, but other Eclipse plugins may be used
Persisting/loading model execution state	No (promised simulation snapshots feature is not implemented yet)

1.5.6 Architecture

Yakindu uses an XML format for storing diagrams. It is based on the eclipse GMF, but extends it with its own notations (for example, to define the environment the statechart operates in).

SExec code is generated from the model, and it is executed or translated to a target language. SExec is a deeply embedded DSL in Java. SExec is interpreted by a traditional interpreter.

During execution, the diagrams are rendered as SWT images and the debugger will load them.

The interpretation code is sequential. For all cycles, the execution engine traverses all the events that are raised and handles them.

1.5.7 Licensing

Eclipse Public License - v 1.0

1.5.8 Reuse possibilities

Yakindu is not ready for use in serious development projects. It can only simulate singleton objects so complex situations with multiple objects of different kinds cannot be simulated. The cost of introducing Yakindu to a project takes all the benefits of using it.

It has some architectural decisions that are likely to be useful for us. For example, converting the model to an internal representation optimized for interpretation may become useful. In the implementation of Yakindu, most classes have a separated interface. For every implementation class there is an interface that is implemented by the class. The event-handling of the application is organized into Notification chains that collect all the changes in the representation. Notification chains capture the effect of the property change events. This is a simple mechanism that allows changes in the DOM (representation) to trigger additional changes.

Simulation snapshots are not implemented, but planned in Yakindu, but they would be even more useful to our project. Yakindu operations (calls to native Java methods) are also useful in our project, but it could be generalized to enable the execution of C++ or C subprograms as well as Java methods.

1.6 AutoFocus3

1.6.1 Introduction

AutoFocus3 is a model based development tool for distributed, reactive, embedded software systems developed by Fortiss, an institute associated with the Technical University of Munich. Its source code is available for access to external developers.

1.6.2 Specifics

AutoFocus3 is a large tool that contains a vast number of sub-modules.

- It comes integrated with Eclipse, provides its own GUI
- Uses an own, Ecore-based model
- Components' behavior is described either by code or by automata
- Execution is event based, executed on a single thread
- Can simulate the passing of a given amount of time
- Contains both interpreter and code generation, and tools for other purposes such as formal verification

1.6.3 Deficiencies

The tool works out of the box (it comes with a packaged Eclipse), and no deficiencies became apparent running it, although its documentation mentions a number of issues.

1.6.4 Compatibility

It uses an Ecore-based model, therefore it can be accessed using standard tools. The model is not UML2 nor fUML but their af3-model.

1.6.5 Feature compliance

Requirement	AutoFocus3
High Priority Requirements	
Model interpretation	Yes
Interactivity	Yes. The user can set breakpoints, step-by-step execution, time-based skipping and manual event triggering is available.
Mass execution	AutoFocus3 does not seem to be designed with that in mind.
Scalability, performance	The implementation is single-threaded and communication and messages are exchanged synchronously with respect to a global, discrete time base (according to this source).
Visual feedback	Yes, the GUI shows all components in great detail. It is even possible to devise a graphical interface for the modelled system in the tool.
Test coverage statistics	Supports state coverage and transition coverage.
Deterministic & nondeterministic mode	It has analysis facilities for nondeterminism.
Timers	As mentioned above, timed transitions are available, and it is also possible to manually skip time.
Connection to native code	The behavior of components can be described by code.
Tracing	Yes.
Medium Priority Requirements	
Model debugging session	Yes, with lots of features like manual event triggering, editing the program state etc.
Test and production model	-

elements	
Possibility to influence simulation speed	Yes, time skipping is available.
Low Priority Requirements	
Conditional breakpoints	No
Profiling: number of running instances etc.	No
Connection to debuggers of implementation languages	No
Persisting/loading model execution state	No

1.6.6 Architecture

AutoFocus3 uses an own model and user interface within Eclipse. It uses a thread to run the execution in the background step by step in a sequential way.

1.6.7 Licensing

Although it lists the licenses of some of the tools it uses, it is not explicitly stated what license AutoFocus3 itself falls under.

1.6.8 Reuse possibilities

Since AutoFocus3 uses a model of its own, direct reuse of the model related parts is probably impossible. As it does not share its codebase with the other tools, it can be interesting to compare how functionality similar to other tools is implemented here: perhaps some bits are more efficient in AutoFocus3 than elsewhere. For state transitions, entry, transition and exit are visualized separately, we might emulate this.

2 Conclusions

- All found open source solutions are far from fulfilling all requirements.
- All of them use classic interpreter technology. No code generation based interactive executor/debugger has been found.
- Alf executor, Moliz and Moka all use the same fUML reference implementation. Topcased and Yakindu have their own execution engines.
- Reuse possibilities:

- fUML & Alf:
  These interpreters can be used to validate other implementations against them. A lot of model classes and interfaces can be reused from each code base. XMI models and ALF files are also reusable for testing purposes. It may be possible to evolve these solutions into multi-threaded implementations.
- Moka:
  The way the debug target communicates to the execution engine, through sockets, should also be considered for reuse. A middle layer of engine is implemented in Moka, which serves as an event dispatcher between the debugger architecture in Eclipse and an execution engine. Since this is a thin layer, it is expected to be possible to reuse it with little or no modification.
- Topcased:
  It may be possible to reuse the debugger architecture to some extent. It's very close to what we have envisioned apart from the missing breakpoints feature.
- Yakindu:
  Converting the model to an internal representation optimized for interpretation may become useful. Yakindu operations (calls to native Java methods) are also useful in our project, but it could be generalized to enable the execution of C++ or C subprograms as well as Java methods.