A UML SIMULATOR BASED ON A GENERIC MODEL EXECUTION ENGINE

Andrei Kirshin, Dany Moshkovich, Alan Hartman
IBM Haifa Research Lab
Mount Carmel, Haifa 31905, Israel
E-mail: {kirshin, mdany, hartman}@il.ibm.com

KEYWORDS
UML, Model Execution, Model Simulation, Model Debugging.

ABSTRACT

Today almost every IT specialist uses models of some form or another. Models help raise the abstraction level of a system description. Although models usually describe IT systems statically, they can also be used to describe the dynamic behaviour of the system. The OMG's MDA® approach suggests describing business and application logic separately from any underlying platform technology in Platform Independent Models. The UML 2.0 provides powerful and flexible behavioural modelling capabilities.

As the focus of the development process shifts from being code-centric to model-centric, the need for an environment to debug and execute models becomes more apparent. The ability to execute models provides additional avenues for the exploitation of the models in validation, verification, and simulation. The use of executable models enables the visualization and discovery of defects early in the development cycle, avoiding costly rework at later stages.

We describe an architecture for implementing a generic model execution engine that enables the simulation of models. The main advantages of our architecture are its generic nature and its dedication to maintaining controllability and observability of the simulation. We have used this generic framework to build a UML simulator, which can be extended to support different UML profiles. The architecture also supports non-UML models.

INTRODUCTION

As the complexity of software increases, it becomes necessary to raise the abstraction level of system descriptions. The Model Driven Architecture® (Brown 2004) of the Object Management Group™ aims to raise the abstraction level and separate business logic from the underlying platform technology. The MDA approach suggests specifying a system independently of a specific target platform, transforming the system specification to a platform specific specification, and then generating code for the target platform.

To enable model driven development, programmers must be provided with the appropriate tools for supporting the entire development process, from platform-independent model specification to platform-specific code generation, and on through testing, deployment, and maintenance.

Exploring system behaviour is an essential part of the development process. There are a variety of tools that enable exploration for platform-specific code, but it is also important to analyze system behaviour in the early stages of development when the impact of defects is high and such defects are costly to repair at later stages. Thus, there is a need for tools to explore system behaviour at the model level.

This paper describes a model execution engine that is a fundamental component of any model behaviour exploration tool. Note that model execution does not necessarily provide an application run-time platform and is not intended to be a generic virtual machine since models are, by definition, at a higher level of abstraction and may omit essential details of the whole application. Our main goal is to provide a platform for model behaviour exploration in the early stages of the development process. Executing a model enables us to understand the architecture and its implications. It enables the early simulation and testing of applications and helps bridge the IT to business communication gap. A generic model execution engine provides mechanisms for realizing the behavioural semantics and observing the model behaviour. Our execution environment can be extended to support a variety of modelling languages, and semantic variation points.

In this paper, we describe the architecture of a generic model execution framework. We focus on its extensibility to enable the execution of models conforming to a variety of meta-models and its controllability and observability to enable interoperability with behavioural exploration tools, such as model debuggers and model-based test generators. We also include sections describing its application to UML model execution, related literature, and further implementation plans.

The relationships between a model execution engine, a modelling language execution framework, and debugging facilities are illustrated in the Figure 1:

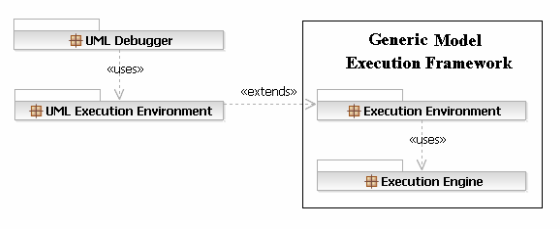

Figure 1: UML Debugger Framework Based on Generic Model Execution Engine

GENERIC MODEL EXECUTION ENGINE

This section describes a generic model execution framework. We first describe an extended set of requirements and then present a conceptual framework based on the requirements. Next, we provide the detailed architecture of our generic execution engine.

The run-time representation of model structure has been described by Riehle et. al. (Riehle at al. 2001). We extend the existing structural methods to support behavioural features.

Requirements

The following set of requirements for the model execution engine form the basis of our work. We require that the execution engine have the following properties:

- Extensibility – support a variety of modelling languages. For example, it should support UML with its behavioural features, including activities, state machines and interactions.
- Observability – provide interfaces for other behaviour exploration tools to allow model-level observation and manipulation.
- Controllability – allow control of the execution process.
- Parallelism – support parallel execution of multiple flows running in the same or different contexts.
- Flexibility – support different implementations of behaviour for the same model element (semantic variation points).
- Scalability – support large models.
- Simplicity – support simple definition and modification of model elements behaviour.

Observability and controllability are generally not required from an execution engine. We incorporate them in order to achieve interoperability between the engine and other development tools including debuggers, test generators, etc.

Execution Engine

It is possible to implement the behavioural logic of any particular modelling language from scratch using any of the existing programming languages such as Java or C++. However, providing a standard set of mechanisms that are common to most modelling languages can simplify the process of describing the model behaviour and therefore significantly reduce the effort required to develop behaviour exploration tools.

Our execution engine provides a fundamental set of execution mechanisms that can be used for a variety of modelling languages. The execution engine allows full control and observation of the entire execution process. Moreover, our execution engine is easy to extend so it can facilitate the implementation of specific modelling language logic. It can also be scaled to support large scale industrial models.

Supporting parallel execution using a programming language such as Java is not simple and does not provide controllability and observability of the execution by other tools. For this reason, we concentrated our attention on providing a specialized engine mechanism for the parallel execution of model elements. The major specializations we adopted are the support of non pre-emptive process execution and the assumption of a single platform. This does not prevent the engine from performing functional simulation of distributed systems.

Interrupt Controller

We provide an open mechanism for the control, modification and observation of the engine functionalities, for example, modifying the scheduling algorithm or overriding commands. This enables other behaviour exploration tools to customize the engine framework. To support this mechanism, the execution engine provides a set of “soft interrupts” that are controlled by an interrupt controller. Every event to be observed (e.g., rescheduling, entering an idle state, receiving a command) generates an interrupt. The engine provides an interrupt handler for some of these interrupts. The interrupt controller allows external tools to implement different interrupt handlers by replacing the responses to the interrupts, without modifying the engine code. For example, replacing the rescheduling interrupt with code that first checks break-point conditions and then executes the original rescheduling interrupt, can add support for break-points as a feature of a model debugger.

Execution Environment

One way to describe model behaviour is to translate the model directly to a target programming language, replacing model elements with the corresponding statement in the implementation language. For example, if the modelling language is UML and the implementing language is Java, a decision node can be translated to the “if” or “switch” statements. Translation of model logic directly into a programming language will improve the performance of its execution, but it requires designing a dedicated compiler for each modelling language. Another problem with this solution is that in order to achieve model-level observability we must keep traceability between the implementation logic and the original model. In the case of a decision node, we

must specify that the generated "if" statement relates to the corresponding UML model element. This correspondence is much more complex for other modelling constructs such as tokens in a Petri net.

Rather than carrying out direct translation, we keep a one-to-one relationship between model elements and their run-time representation. This is done using the notion of the model element model element *Instance* to achieve the required model-level observability. An instance describes the behaviour and run-time data of a particular model element. Each model element (e.g., activity A, action B) must have a corresponding Instance describing its behavioural logic. Each model element can have any number of instances at the same time, since the containing model element (e.g., activity, state machine) may be instantiated several times. For example, the *ActivityEdgeInstance* Java class can describe the run-time logic of all activity edges in the model, but each one of the edges will have a corresponding object of the *ActivityEdgeInstance* Java class during execution of the edge logic.

The *execution environment* contains all information specific to the modelling language, including Instance descriptions; it is also responsible for their creation. The execution environment provides the execution framework with flexibility by specifying methods for mapping between model elements and their run-time representations to allow the support of semantic variation points. An execution environment must be provided for each of the modelling languages whose models we want to execute, in contrast with the execution engine mechanisms that are common to all modelling languages.

Interpretation vs. Compilation

As discussed above, a model element is represented at run-time by its *Instance*. There are two different ways to define instance logic. One way is an interpretation and the other way is as a compilation.

The interpretation process consists of the following steps:

- Step 1: Definition. Instance code is written once for each meta-model element by the developer of the execution engine extension. The execution of models conforming to the meta-model does not require any code generation (automatic or manual).
- Step 2: Run-time. Instance logic may analyze the corresponding model elements at run-time. An instance implementation (instance implementing class) is chosen at run-time based on the corresponding model element type (meta-model element).

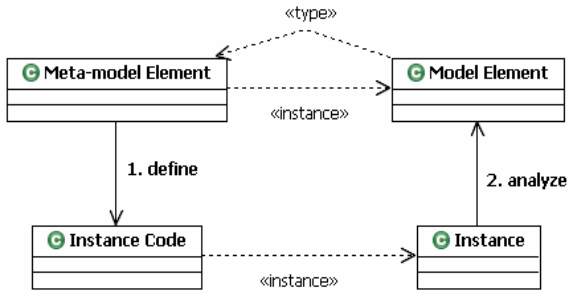

Figure 2: Interpreter

The compilation process consists of the following steps:

- Step 1: Definition. A compiler component is written once for each meta-model element by the developer of the execution engine extension.
- Step 2: Compilation. The compilation is performed once per specific model. Instance code is generated for all model elements.
- Step 3: Run-time. All model element logic is compiled in the Instance code. The relation between a model element and its Instance code is set up during the compilation step.

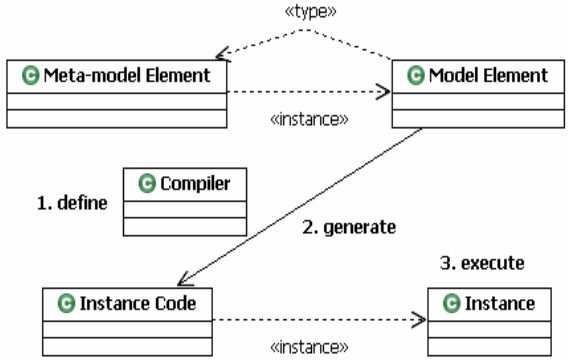

Figure 3: Compiler

The following example illustrates the difference between the two approaches:

Let's suppose we want to describe UML Activity logic and our model contains two activities, A and B, with different numbers of nodes and edges. We assume the activity's execution logic is comprised of the following steps:

- Initialize all nodes that have no incoming edges.
- Wait until all flows have finished or until one of the activity's final nodes was reached.
- Notify the invoker of the activity that it has completed.

Suppose a Java class named *ActivityInstance* contains the implementation of this logic.

In the interpreter approach, *ActivityInstance* will contain (among other things) the logic that determines which nodes belong to the activity at run-time by accessing the model and analyzing its structure. In this

case, both activities A and B in the model will be represented as instances of the same *ActivityInstance* Java class.

In the compiler approach the model will pass through a compiler component that generates two Java classes, *AActivityInstance* and *BActivityInstance* representing A and B, where each of these classes extends the *ActivityInstance* class. Each of the generated classes will contain code for the generation of nodes based on the analysis that was performed during the compilation. No run-time analysis of the model is required.

Another possibility is a combination of both methods, where instance code is generated for **some** model elements (see Figure 4) while others are interpreted at run-time. To execute such a model, the following steps are performed:

- Step 1: Definition. Instance code is written once for a subset, X of the meta-model elements. A compiler component is written for the remaining meta-model elements, X`.
- Step 2: Compilation. Instance code is generated for the model elements of types described in X`.
- Step 3: Run-time. Instance logic that was not generated in Step 2 analyzes the model elements of types in X.

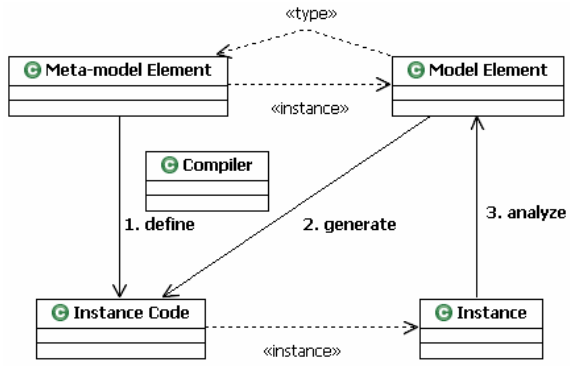

Figure 4: Interpreter and Compiler Combined

In our architecture, we want the flexibility to adopt either the interpreter or compiler approaches, depending on the specific meta-model.

UML SIMULATION

Using the architecture described in the previous section, it is possible to define an environment for any meta-model. In this section we briefly explain how a UML environment can be constructed.

For every element of the UML meta-model, we define an *Instance* describing its behaviour. Our architecture is implemented in Java so an *Instance* is a Java class. We can define a number of *Instances* for a single meta-model element to support semantic variation points. A configuration file is used to select the appropriate semantic variation. Moreover, one can add support for any UML profile by defining *Instances* for stereotyped model elements. To improve the performance of our debugger we generate code of specific *Instances* for some model elements.

Since *Instances* are Java classes, memory management, garbage collection, and primitive data types are handled by the Java Virtual Machine. This implementation allows us to support Java as an action language with minimal effort.

We can provide visualization facilities and other capabilities related to the execution of UML behaviours using the connection between *Instances* and model elements.

Figure 5 illustrates an example of a running UML 2.0 activity.

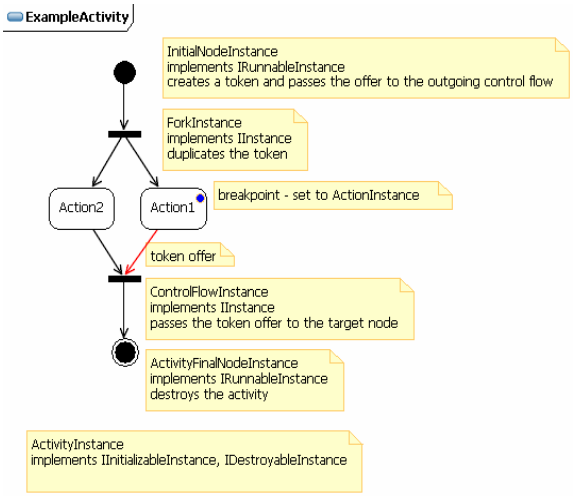

Figure 5: Example of Running Activity

UML SIMULATOR PROTOTYPE

The model execution engine provides mechanisms for the realization of behavioural semantics and the execution and observation of model behaviour. We used this generic execution engine to implement a UML Model Simulator. It is designed as an extension to Rational Software Architect (RSA), adding execution and simulation capabilities.

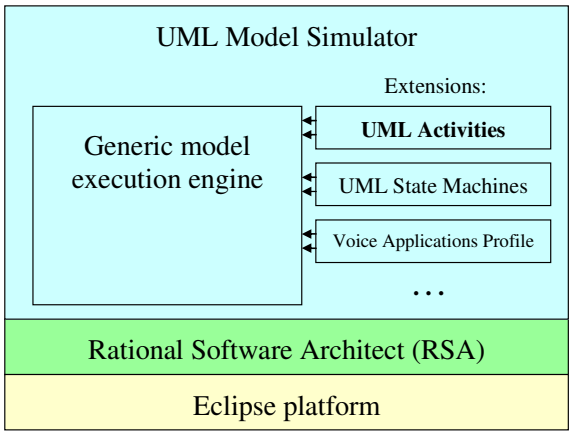

Figure 6: UML Simulator

The first version of the tool supports UML classes and primitive data types, and focuses on the execution of activities. It also supports Java as an action language. The behaviour of a UML element can be changed using profiles by applying stereotypes to UML elements. The tool can be extended to support the execution of models that conform to a specific profile.

The UML Model Simulator provides a wide range of capabilities. It supports the most commonly used execution modes such as step-wise execution and run to break-point. It allows run-time state observation, including the visualization of the current execution state, token offers, and parameter values. The debugger also allows dynamic object creation and user initiated invocation of behaviours and operations. The most useful feature is the visualization of the execution of model behaviour.

We extended RSA with a Model Debugging perspective that contains various views:

Model Explorer view. This is the corresponding RSA view with the addition of two items to the popup-menu: *Debug Model* used to start a model simulation session, and *Add Breakpoint,* which can be applied to any runnable element of the model.

Debug view. This view is responsible for the control of the execution process (starting, stopping, and step mode execution), object creation and destruction, observing the values of objects' attributes, and invoking the operations on the objects.

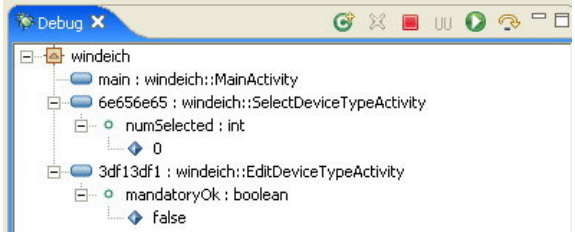

Figure 7: Debug View

Call Stack view. This view shows all running behaviours organized according to the call history.

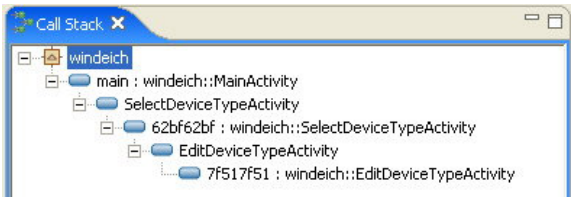

Figure 8: Call Stack View

I/O view. This view, is used to observe outputs sent from the model and to send or broadcast signals to model behaviours.

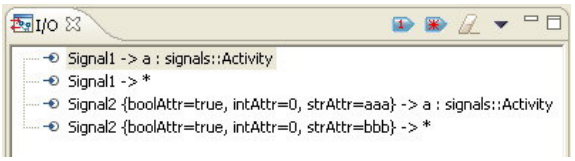

Figure 9: I/O View

Ready view. This view shows all elements that are ready for execution and all elements that have reached breakpoints. The user can select the next element for execution.

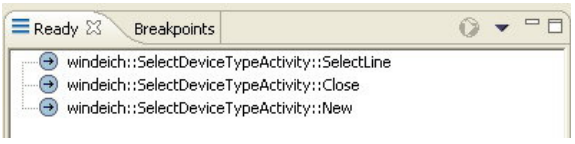

Figure 10: Ready View

Breakpoints view. This view lists all active breakpoints.

Diagram visualization. This is the most powerful part of the tool. It illustrates the execution of a UML behaviour model. The user can see which nodes are ready for execution (green), which edges pass tokens (blue), and which node provides the token (magenta):

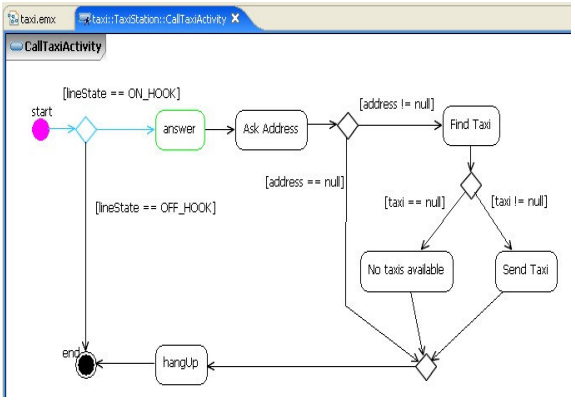

Figure 11: Diagram Visualization

We developed a UML profile for model simulation. The profile is used to specify:

- Elements containing Java code (Java as an action language)
- External classpath for linking with existing code

- Initial system configuration – objects that are created at the beginning of the simulation session
- System boundaries – classes that are visible to the user
- Predefined values for parameters
- Predefined decisions for activity decision nodes

We have completed a prototype version of the tool and we continue to extend the capabilities of the tool and improve its maturity.

RELATED WORK

Prior work on model execution can be divided into two streams, the academic and the industrial. The academic study of model execution was initiated by Riehle et. al. in their paper (Riehle at al. 2001). They describe a UML virtual machine focusing almost entirely on the instruction set and memory model. Much of their work has been subsumed by developments in the definitions of UML 2.0 (Selic) and the MOF (MOF).

In the industrial arena, several model execution frameworks have been defined. One of the earliest tools capable of executing models was Rational Rose Real Time, now known as Rational Rose Technical Developer (Rational Rose Technical Developer). Other tools that use a similar approach are the Kennedy Carter tools for xUML (Raistrick at al. 2004), the Kabira tools (Kabira Technologies), which implement Mellor and Balcer's version of executable UML, and the I-Logix (I-Logix) tool Rhapsody, which implements Harel's Statecharts (Harel and Politi 1998). Other commercial tools that claim to implement model execution include those from (Interactive Objects), (Compuware), (Telelogic), and (Artisan Software).

All of these tools rely almost exclusively on state diagrams to describe behaviour in a generic way. They only support one behavioural semantic each – and no two of them support precisely the same semantics. Each tool limits itself to a particular application domain and adopts the most appropriate conventions for that domain. Many of them compile the state diagrams into executable code, losing the observability property of our framework. Our framework allows for incomplete models to be executed at the level of abstraction appropriate to the modelling context since we support user/tool interaction at run-time to supply missing information. We are also taking part in the OMG's standardization effort for executable UML.

FURTHER WORK

This work is taking place under the auspices of the ModelWare project (ModelWare) as part of the task of building an infrastructure for industrial strength model driven development (MDD). ModelWare is a European project involving a consortium of 19 academic and industrial partners whose objectives are to provide the tools, processes, and methodologies supporting MDD, and to promote the industrial adoption of these work products.

Our model execution engine is being used as the basis for a UML simulator and model-based test generation tools. These tools are being used in several distinct industrial contexts including large-scale distributed systems for air traffic management, voice-based application dialogs, retail business applications, and others. In order to satisfy these diverse modelling requirements, we are developing industry specific profiles for executable models with the appropriate interfaces to our model execution framework.

ACKNOWLEDGEMENTS

MODELWARE is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006). Information included in this document reflects only the author's views. The European Community is not liable for any use that may be made of the information contained herein.

REFERENCES

Accelerated Technology. http://www.acceleratedtechnology.com

Artisan Software. http://www.artisansw.com

Brown A.W. 2004. "Model-Driven Architecture: Principles and Practice." Journal of Systems and Software Modeling, Volume 3, Number 4, pp314-327, Springer Verlag.

Compuware. http://www.compuware.com

Harel D. and M. Politi. 1998 "Modeling Reactive Systems with Statecharts: The STATEMATE Approach." McGraw-Hill (258 pp.).

I-Logix. http://www.ilogix.com

Interactive Objects. http://www.io-software.com

Kabira Technologies. http://www.kabira.com

Kennedy-Carter http://www.kc.com

Mellor S.J. and M.J. Balcer. 2002. "Executable UML: A Foundation for Model Driven Architecture." Addison-Wesley.

ModelWare Project web-site, http://www.modelware-ist.org

Raistrick C., P. Francis, and J. Wright. 2004. "Model Driven Architecture with Executable UML." Cambridge University Press.

Rational Rose Technical Developer, http://www-306.ibm.com/software/awdtools/developer/technical

Riehle D., S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. 2001. "The Architecture of a UML Virtual Machine." In Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01). ACM Press. Page 327-341, http://www.riehle.org/computer-science/research/2001/oopsla-2001.html

Selic B.V. "On the Semantic Foundations of Standard UML 2.0." http://www-128.ibm.com/developerworks/rational/library/05/317_semantic

Telelogic. http://www.telelogic.com

The Meta Object Facility. http://www.omg.org/technology/documents/formal/mof.htm