Textual, executable, translatable UML*

Gergely Dévai, Gábor Ferenc Kovács, and Ádám Ancsin

Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary,
{deva,koguaai,anauaai}@inf.elte.hu

Abstract. This paper advocates the application of language embedding for executable UML modeling. In particular, *txtUML* is presented, a Java API and library to build UML models using Java syntax, then run and debug them by reusing the Java runtime environment and existing debuggers. Models can be visualized using the Papyrus editor of the Eclipse Modeling Framework and compiled to implementation languages.
The paper motivates this solution, gives an overview of the language API, visualization and model compilation. Finally, implementation details involving Java threads, reflection and AspectJ are presented.

Keywords: executable modeling, language embedding, UML

1 Introduction

Executable modeling aims at models that are completely independent of the execution platform and implementation language, and can be executed on model-level. This way models can be tested and debugged in early stages of the development process, long before every piece is in place for building and executing the software on the real target platform.

Executable and platform-independent UML modeling changes the landscape of software development tools even more than mainstream modeling: graphical model editors instead of text editors, model compare and merge instead of line based compare and merge, debuggers with graphical animations instead of debuggers for textual programs, etc. Figure 1 depicts the many different use cases such a toolset is responsible for. Models are usually persisted in a format that is hard or impossible to edit directly, therefore any kind of access to the model is dependent on different features of the modeling toolset. Are the available tools ready to meet the requirements? Experience shows that they still need to evolve a lot. Another aspect is that, while graphical notations help understanding software more than textual representations, editing graphics is usually less productive than editing text [13].

Textual modeling languages solve many of these concerns. The many high-quality text editors with sophisticated editing and search-related features, as well as the numerous compare and merge tools serve as a solid basis. However, merely defining a textual notation for modeling does not solve all the issues. Text

* This work is supported by Ericsson Hungary and KMOP-2008-1.1.2.

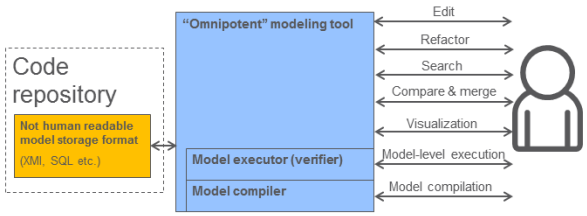

Fig. 1. Using an "omnipotent" modeling framework

editors need plugins to do syntax highlighting and autocompletion correctly for the new language. Executable modeling makes even heavier-weight demands: interpreter and debugger are also required.

Language embedding is a technique to implement a language in terms of an existing one. The implementation language is called the *host language*, while the implemented one is referred to as the *embedded language*. One way to embed a language is to create an API in the host language that offers the constructs of the embedded language for the developers. A program of the embedded language is therefore a host language program that heavily uses this API. The API can, on one hand, implement the execution semantics of the embedded language (making it executable in the runtime environment of the host language) and, on the other hand, can build an internal representation of the embedded program (to be used further for compilation, visualization, analysis etc.). The big advantage is that an embedded language can piggyback on the development and runtime environment of the host language.

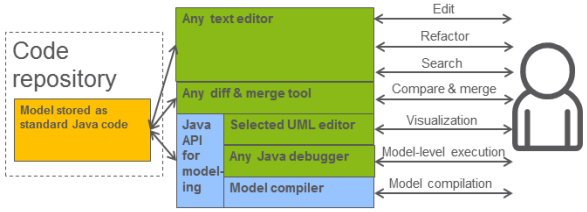

Fig. 2. Using an embedded language for executable modeling

This paper proposes using language embedding for executable UML modeling. As stated above, such an embedded language can be realized by an API that provides the constructs of executable UML. It is natural to use an object-oriented programming language as host language, because many of the UML modeling concepts, such as classes, inheritance, attributes, operations, visibility, instance creation etc. are already present in these languages. Other constructs, like components, associations, state machines, events etc. are usually not ex-

plicitly present, therefore these abstractions have to be provided by the API. The implementation of this API has to capture the runtime semantics of these constructs. A UML model can then be implemented by a program in the host language naturally reusing the modeling constructs that are part of the host language and using the API to build the rest of the model. Model execution, in this setup, is not more than running the resulting program in the execution environment of the host language. Figure 2 highlights how many elements of the "modeling environment" are provided for free by this approach. Editing, searching, basic refactoring can be performed in the usual development environment (smart text editor or IDE) of the host language. The debugger of the host language can be reused to observe model execution in detail. Visualization of the model can be achieved via existing UML editors or viewers.

Authors of this paper have created a prototype implementation of a modeling language along these lines [7]. Its name, *txtUML* stands for *textual, executable, translatable UML*. Our host language is Java. Papyrus, the primary UML editor of the Eclipse Modeling Framework, is selected to provide model visualization.

The paper is organized as follows. The next section reviews related work and compares them to our approach. Section 3 presents the language API, while sections 4, 5 and 6 talks about execution, visualization and model compilation. More details on the implementation of these aspects are given in Section 7. The last section summarizes the paper and highlights its most important contributions.

2 Related Work

The UML standard does not define precise execution semantics, but the *Foundational Subset for Executable UML (fUML)* [18] is an OMG standard doing that for a subset of UML, including classes and actions, but excluding components and state machines. There is another OMG standard in preparation to define precise semantics of UML composite structures [20].

BridgePoint [15] is an executable modeling tool originally based on the *Shlaer-Mellor methodology* [24]. Its metamodel is a non-standard fork of UML, including components, classes, state machines (with graphical-only editor) and action language (textual with custom syntax). BridgePoint includes open-sourced [2] editor, and commercial model executor with graphical debugging features and model compiler. A similar, commercial solution is *iUML*, a tool based on the *xUML* methodology [23]. A UML simulator based on a generic model execution engine [14] was developed by IBM Haifa Research Lab and integrated into *Rational Software Architect*. *Topcased* [22] is an open-source joint industry-academia project to create an open-source engineering toolkit. It is implemented over the Eclipse platform and supports UML model execution with graphical feedback. The project is currently under migration to *PolarSys* [8]. The *Eclipse Modeling Framework (EMF)* [25] is surely the most active open source modeling community nowadays. *Moka* [4] is one of the EMF-based tools, and it provides model execution with graphical animations for fUML actions. The *Yakindu Statechart Tools* [10] is an open source toolkit for editing and simulating statecharts and

compiling them to C, C++ and Java. It supports statecharts with semantics slightly different from UML state machines and a limited textual action language. All these tools come with graphical-only editors for most layers of their modeling language and are subject to the concerns described in Section 1.

Turning to textual UML modeling, *Action Language for Foundational UML (Alf)* [17] needs to be mentioned. This is an OMG standard that defines textual syntax for fUML. Although there are prototype tools [11] to parse and run Alf code, the support for this language is not mature enough yet. Furthermore, fUML and Alf are based on a limited subset of UML, they miss concepts like components and state machines that are essential for executable modeling. The *Umple* project [12, 9] shows a very neat way to integrate textual modeling with traditional coding. It provides custom syntax for associations, state machines that can be mixed directly into Java, C++, Php or Ruby sources. The Umple compiler preprocesses these files and translates the Umple code fragments to the language they are mixed in. Umple relates to our work because both approaches are textual and try to make modeling more lightweight. On the other hand, their goals are different: Umple provides easy mix-in of model abstractions into traditional programming projects, while txtUML aims at completely platform and implementation language independent, executable modeling.

Section 1 discussed the importance and difficulties of *model management*. The Epsilon [21, 1] project provides a number of textual languages based on a common expression language for different model management tasks like transformation, comparison, merge etc. In particular, not human readable model storage format is one aspect of the model management problem, which is addressed by the OMG standard *UML Human-Usable Textual Notation (HUTN)* [16]. In this paper, in contrast, we explore the possibilities of reusing the management toolset of a mainstream programming language.

Regarding the visualization aspect of our project, *MetaUML* [3] (a LaTeX package to create UML diagrams) and *TextUML* [6] (providing custom textual notation to build UML diagrams) are related projects. Furthermore, there are UML tools (Topcased java2uml importer, ObjectAid, UML-lab, Class Visualizer, MoDisco etc.) that can create UML diagrams (usually class diagrams) based on Java code. The important difference compared to our approach is that these tools start from arbitrary Java code and produce a small part of a model on a best effort basis, while our library only accepts a subset of Java (the txtUML embedded language) and turns the full model definition (including state machines, actions etc.) to a model for visualization or compilation.

3 Language Overview

Our implementation currently supports classes, state machines and action code. To make clear distinction between plain Java classes and those that are part of the UML model description, all model classes have to inherit (directly or

indirectly) from `ModelClass`, which is provided by our API[1]. Associations are also represented by class definitions, inheriting from `Association`. Multiplicities are described as Java annotations like `@One`, `@Many` etc. For example, a class diagram talking about *users* using *machines* can be described this way:

```
class Machine extends ModelClass { /* ... */ }
class User extends ModelClass { /* ... */ }
class Usage extends Association {
  @One Machine usedMachine;
  @Many User userOfMachine;
}
```

Signals are defined by classes that inherit from `Signal`. Let us define a signal that users can send to machines by pressing the power button:

```
class ButtonPress extends Signal {}
```

State machines are described by nested classes inside model classes, inheriting from `State`, `InitialState` or `Transition`. The annotations `@From`, `@To` and `@Trigger` complete the definition of transitions.

```
class Off extends State { /* ... */ }
class On extends State  {
  public void entry() { /* ... */ }
  public void exit() { /* ... */ }
}

@From(Off.class) @To(On.class) @Trigger(ButtonPress.class)
class SwitchOn extends Transition {
  public void effect() { /* ... */ }
}
```

Operations of model classes, entry and exit actions of states and effects of transitions contain action code. The `doWork` operation of the `User` model class, for example, log a message to the console, selects the machine that it is associated with, then presses its power button:

```
void doWork() {
  Action.log("User: starting to work...");
  Machine myMachine = Action.selectOne(this, Usage.class, "usedMachine");
  Action.send(myMachine, new ButtonPress());
}
```

Action language constructs include instance related operations (eg. creation, deletion, access to operations and attributes), association related ones (link, unlink, different traversals), signal sending, logging and basic operations of elementary types like integers and strings.

It is important to note that only a restricted subset of Java is allowed in model definitions. For example, adding a `Vector<User>` attribute into the `Machine`

[1] Inheritance between two model classes encodes UML generalization. Java does not support multiple inheritance, which is currently a limitation of txtUML.

class instead of the explicit definition of the association is invalid: Selecting the concrete collection type is an optimization issue and should not be part of the abstract model. The techniques to be discussed in Section 7.2 can rule out the violations of these rules, including constraints specified in the UML standard.

Using an embedded language in Java makes model descriptions a bit more verbose than tailor-made custom syntax. This is a moderate price for familiar syntax and the possibility of model execution and debugging with existing tools.

4 Execution

A model, defined using the API introduced above, can be compiled, run and debugged as any Java application. Figure 3 shows a debugging session executing the example used in Section 3. A breakpoint, set on the signal sending instruction of the `doWork` method, is being hit. It is possible to inspect the threads that belong to the active object instances, the attributes of these instances, in particular the current state of the machine instance. When stepping over the send instruction in the debugger, the developer can verify that the state changes from *off* to *on*.

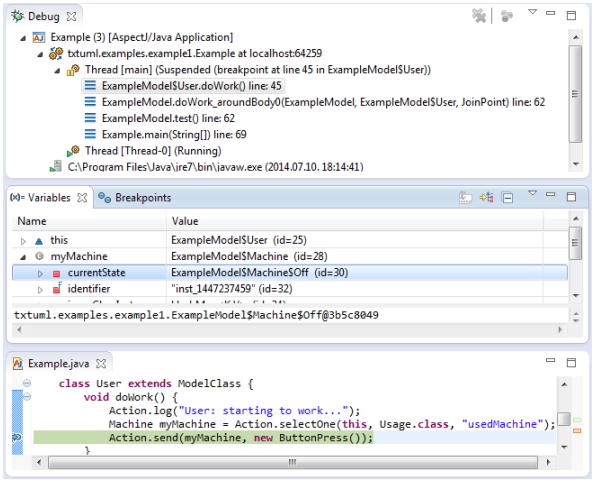

Fig. 3. Eclipse debugging session executing the example of Section 3

Discussing the execution semantics behind our implementation is far beyond the scope of this paper. We summarize our approach as follows: We start from the constructs in the xtUML modeling language [2], port them to standard UML2 [19] and gradually add more UML2 abstractions. In case of abstractions that are covered by other standards, like fUML [18] and PSCS [20], we adopt the standard semantics. If the execution semantics of the considered abstractions are

not standardized (eg. state machines), we closely follow the informal semantics given in the UML2 standard, examine different possibilities and select one based on usability and implementation considerations.

5 Visualization

Even if our proposal is a textual modeling language, we consider visualization an extremely important aspect. Graphical representation of system architecture and behavior makes it easier to understand large systems. The advantage of our approach is that we have the possibility to reuse any UML editor or viewer tool for model visualization by generating the persistency format of the selected tool. We use the Ecore based implementation of UML2 as our internal model representation that gives us Papyrus-compatible [5] model persistency for free. Papyrus provides autolayout for diagrams.

Note, however, that automatically laid out diagrams are far from perfect, because the position of model elements and their relative distance in a hand-drawn diagram also carries information. For example, the class representing the most important concept is usually placed in the middle of a class diagram. Closely related classes are drawn close to each other, while marginal or technical associations can be long broken lines. Our plan is to extend the embedded language with constructs to define the relative placement of model elements. For example, one could specify that the `User` class has to be displayed next to the class `Machine`, on its right hand side. This is definitely an important future work in our project.

Figure 4 shows the Papyrus editor showing our example model. The tree-view of the model in the *Model Explorer* on the left hand side is automatically loaded when opening the model persisted by our API. The class diagram on the right-hand side can be created by manually dragging the classes from the Model Explorer onto the diagram.

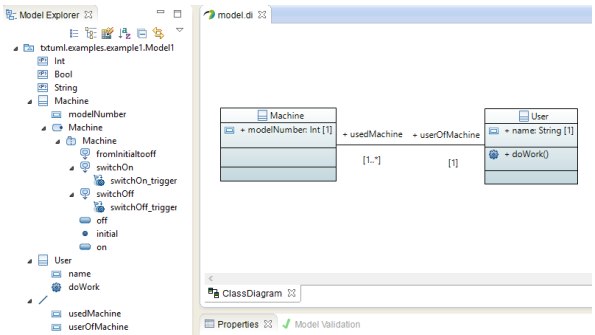

Fig. 4. The Papyrus editor showing the example model of Section 3

6 Translation

The toolchain of executable modeling is completed by the *model compiler* that turns a model into a program in a given implementation language (C++, Java, etc.) optimized towards a selected platform and runtime libraries. Our implementation includes a prototype model compiler generating C++. For example, let us show a fragment of the generated code for the example used throughout the paper:

```
struct Machine
{
  std::vector<User*> userOfMachine;
  enum state { state_Init, state_Off, state_On };
  state current_state;
  /* ... */
};
```

The association between the machine and its users results in the member called `userOfMachine` in the machine class. Since the multiplicity of this end of the association is `@Many`, a vector of pointers to user objects is generated. Note, that the compiler is free to choose a different collection type for optimization reasons. In particular, this member can be omitted if the association is never used to navigate from machines to users. The states of the machine are represented by the enumeration type `state` and the `current_state` member is responsible for storing the actual state in runtime.

Section 5 revealed that we use the Ecore-based UML2 implementation as the internal representation of the models. The model compiler's task is to query the model via the UML2 API and turn its elements C++ to code.

7 Implementation

7.1 Implementation of the execution semantics

Execution semantics of the model are captured by the implementation of our API, introduced in Section 3. The interesting bit of this implementation is the asynchronous communication between object instances. Any modeled class has to inherit from `ModelClass`. Its constructor creates a Java thread for all the instances[2]. Each thread has a mailbox of type `LinkedBlockingQueue<Signal>` that other instances can put signals in asynchronously. The threads run the following loop:

```
while (true) {
    Signal signal = mailbox.take();
    parent.processSignal(signal);
}
```

[2] This is the reason to use inheritance to mark classes belonging to the model.

That is, they are polling their mailboxes for incoming signals and propagate those to their object instances (`parent`) for processing. The `processSignal` method implements the runtime semantics of state machines. Note that the `take` method blocks the thread if the mailbox is empty, therefore no busy-waiting is involved. When the object instance is deleted, its thread is interrupted, leading to an `InterruptedException` that stops the loop above.

7.2 Building the internal representation of models

Both visualization and model compilation requires a representation of models that can be queried and transformed to graphics or text. In our case this representation is the Ecore-based implementation of UML2.

Using Java reflection, it is possible to get information about the static structure of Java code. For example, in case of the `Machine` class, we iterate through its attributes and member functions to find out the attributes and operations to be added to the model representation via the UML2 API. Then, the enclosed classes are examined, if they inherit from `State` or `Transition` in order to add the state machine of this class to the model.

Java reflection is unable to look into method bodies, but it is possible to call methods via the reflection API. However, simply calling the methods found eg. in states or transitions would not help in building model representation, since the instructions inside these methods implement the runtime semantics of the model. For this reason, we use AspectJ, an aspect-oriented extension to Java. For example, the following AspectJ code fragment replaces all method calls on instances of `ModelClass` with code that adds the corresponding call operation element to the model:

```
Object around(ModelClass target): call(* *(..)) && target(target) {
  return Importer.call(target, thisJoinPoint.getSignature().getName(),
                       thisJoinPoint.getArgs());
}
```

8 Summary

In this paper we presented *txtUML*, which is an executable UML language embedded in Java. Our main contributions are the following:

- A novel textual UML modeling method that makes model-level execution and debugging possible by reusing the Java runtime environment and Java debuggers.
- Application of Java reflection and AspectJ for creating model representation that can be used for visualization and model compilation.
- Prototype implementation of the framework.

Using a mainstream programming language provides easy integration to existing toolchains, while reusing its execution environment and debuggers reduces the implementation effort needed to create a framework for executable UML.

References

1. Epsilon project. http://www.eclipse.org/epsilon/
2. Executable Translatable UML Open Source Editor. https://www.xtuml.org/
3. MetaUML project. http://metauml.sourceforge.net/old/
4. Moka. http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
5. Papyrus. http://wiki.eclipse.org/Papyrus
6. TextUML project. https://github.com/abstratt/textuml
7. The txtUML project. http://txtuml.inf.elte.hu/
8. Topcased migrates to PolarSys. http://polarsys.org/topcased-migrates-polarsys
9. Umple project. http://cruise.eecs.uottawa.ca/umple/
10. Yakindu Statechart Tools. http://statecharts.org/
11. Alf Open Source Implementation. http://modeldriven.org/alf/ (2013)
12. Forward, A., Badreddin, O., Lethbridge, T.C.: Umple: Towards combining model driven with prototype driven system development. In: Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on. pp. 1–7. IEEE (2010)
13. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Textbased modeling. In: 4th International Workshop on Software Language Engineering (2007)
14. Kirshin, A., Dotan, D., Hartman, A.: A uml simulator based on a generic model execution engine. In: MoDELS Workshops. vol. 4364, pp. 324–326 (2006)
15. MentorGaphics: BridgePoint xtUML tool. http://www.mentor.com/products/sm/model_development/bridgepoint/
16. Object Management Group: UML Human-Usable Textual Notation (HUTN), standard, version 1.0. http://www.omg.org/spec/HUTN/ (2004)
17. Object Management Group: Action Language for Foundational UML (ALF), standard, version 1.0.1. http://www.omg.org/spec/ALF/ (2013)
18. 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)
19. Object Management Group: Unified Modeling Language (UML), standard in preparation, version 2.5 beta 2. http://www.omg.org/spec/UML/2.5/Beta2/ (2013)
20. 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)
21. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on. pp. 162–171. IEEE (2009)
22. Pontisso, N., Chemouil, D.: Topcased combining formal methods with model-driven engineering. In: Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on. pp. 359–360. IEEE (2006)
23. Raistrick, C.: Model driven architecture with executable UML, vol. 1. Cambridge University Press (2004)
24. Shlaer, S., Mellor, S.J.: The Shlaer-Mellor method. Project Technology white paper (1996)
25. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)