Novel architecture for executable UML tooling*

Gergely Dévai, Tibor Gregorics, Boldizsár Németh, Balázs Gregorics, Dávid János Németh, Gábor Ferenc Kovács, Zoltán Gera, András Dobreff, Máté Karácsony
{deva,gt,nboldi,grbtaai,nemdav94,kovacsgabor,gerazo,doauaai,kmate}@caesar.elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

Executable UML [Obj15b] models define both behavior and structure of software. These models can be executed, debugged and tested independently of the target platforms, providing early validation [DKN$^+$15]. Model compilers translate them to efficient, platform-specific target code.

Providing a practical toolchain for large scale executable UML modeling in industrial setup is challenging: version control, compare and merge functions, convenient editor, debugging support, high quality diagrams and model compilation need to be provided. On the other hand, the toolchain should be lightweight for scalability, stability and for low tool development costs.

In this paper we propose an architecture for executable UML modeling to achieve these goals using a text-based approach [GKR$^+$07]. We discuss how technologies like Xtext [Xteb] and Xbase [Xba], language embedding, JDT and Papyrus UML [Pap] can be integrated into a practical toolchain to design, debug, visualize [GGK$^+$15] and translate models. The proposal is based on a working implementation: *txtUML* [txt], which is now used in a pilot project by our industrial partner.

1 Introduction

Executable software modeling starts with a platform-independent model. Such a model is completely independent of the execution platform and implementation language, and can be executed, debugged and tested on model-level. This enables early functional validation of the software being developed. In order to test and deploy the product on the target platforms, model compilers are used to generate code in selected implementation languages. These code generators take additional information about the specifics of the targeted platform (in the form of platform-specific model or platform description).

The key point here is *model-level execution*, which enables the following two use cases:

- Interactive debugging: The execution of the model can be analyzed using the usual debugging features (breakpoints, stepping, variable view) and model specific features, such as the animation of state machines.

*This research is supported by Ericsson Hungary.

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

This use case requires the integration of the model execution engine with the user interface of the development environment.

- Automated mass testing: The model is exercised on a configured set of test cases as part of nightly testing or sanity checks before a commit. In this case command line compatible tooling is needed, which can be easily integrated into testing frameworks. Runtime performance of the model execution engine is important in this use case.

An executable software modeling environment must support many functionalities: a model editor with the graphical visualization of the model, tools for model compare and merge, a debugger with graphical animations, a model compiler, etc. Figure 1 depicts the many different use cases such a toolset is responsible for. Our experience shows that the available open source tools still need to evolve a lot to provide convenient, robust, scalable and stable solution for all these requirements.

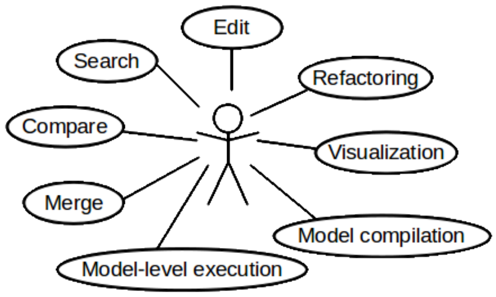

Figure 1: Use cases of executable modeling

Textual modeling solves many of these concerns: High-quality text editors with sophisticated editing and search-related features are available, and users can select from numerous compare and merge tools. It is also faster for experienced developers to edit models in text rather than to edit graphics, which is partly the consequence of the maturity of text editors compared to graphical model editors. However, merely defining a textual notation for modeling does not solve all the issues. Text editors need plugins to do syntax highlighting and auto-completion correctly for the new language. Moreover, graphical visualization of certain kinds of models, like UML for example, is essential: Understanding a model is much easier by looking at an expressive diagram than reading text. Thus the visualization of the textual model must be established. Executable modeling makes even more heavy-weight demands: interpreter and debugger are also required.

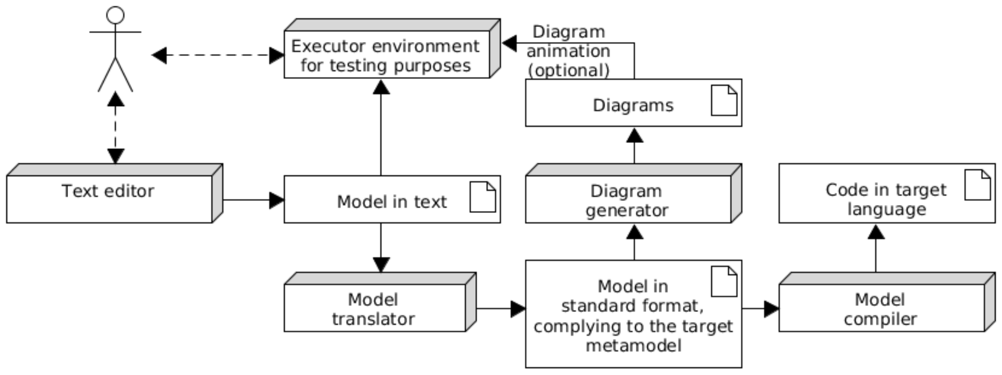
Executor environment for testing purposes
Diagram animation (optional)
Diagrams
Text editor
Model in text
Diagram generator
Code in target language
Model translator
Model in standard format, complying to the target metamodel
Model compiler

Figure 2: Architecture overview

This paper proposes a novel architecture, shown in figure 2, for text-based executable software modeling tools, taking into account the above mentioned use cases and challenges. Dashed lines on the diagram denote interaction between two modules, while continuous lines represent input and output. This architecture is validated by an open source prototype [txt], a set of Eclipse plugins created by the authors of this paper. It is called *txtUML*, which stands for *textual, executable, translatable UML*.

1.1 Overview

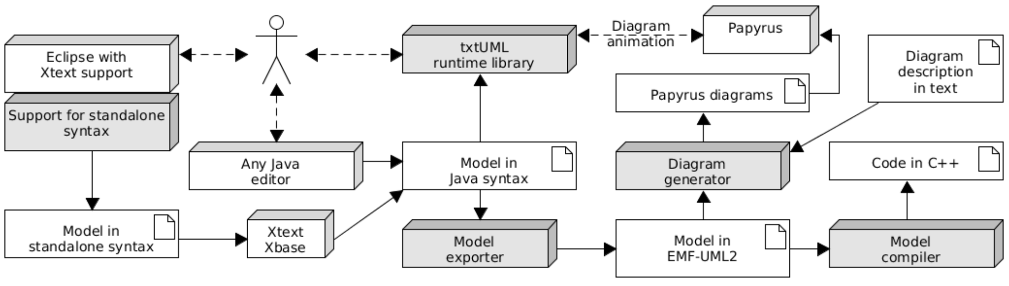

Figure 3: Overview of txtUML

Figure 3 gives an overview of the prototype implementation. Modules with gray background are developed in the txtUML project, while white ones are independent components we rely on.

Users define the UML models in text, and have two options regarding the syntax: Standalone syntax is clean and short, but users need to learn new syntactic elements. The other option is an embedded language in Java, which is realized by a Java API providing the necessary constructs to define models. This option is useful for Java programmers not willing to learn new syntax and opens up possibilities to edit, run and debug txtUML models in non-Eclipse Java development environments.

Models in standalone syntax are translated on-the-fly to the embedded Java syntax using *Xtext* and *Xbase* as underlying Eclipse technologies. See section 3.1 and 3.2 for details of the two syntaxes and the translation process. The resulting Java programs — on top of the txtUML runtime libraries — can be run and debugged in any Java environment. If Eclipse is used, Xtext and Xbase makes the standard debugging features (breakpoints, variable view, stepping in the code) available in standalone syntax as well.

The Java programs defining models can be translated to EMF-UML2 representation which is the de facto standard format of UML models in Eclipse environment. In order to help understanding and validating the models created in text, we generate UML diagrams compatible with the *Papyrus* open source UML framework (section 3.4). Currently class and state machine diagrams are supported. The txtUML runtime is able to communicate with the generated state machine diagrams and can animate them when the model is running or being debugged.

The toolchain is completed by a C++ code generator that uses the EMF-UML2 model as input (section 3.6). The toolchain can be extended by further project-specific code and document generators all working on the same, platform-independent EMF-UML2 representation.

The main novelty of this architecture is the multi-purpose Java syntax which is (1) a full-fledged language frontend, (2) the target of the translation from the standalone syntax and (3) the source of the UML model generation process at the same time. We summarize the most important advantages of this setup as follows:

- Running the models as Java programs provides *higher performance* than interpretation. This is important in automated testing scenarios.
- Learning new syntax and using its editor is not mandatory: The Java frontend is *standard Java with a smart API* and can be used in any Java development environment.
- The platform-independent, high abstraction level language allows the generation of *standard UML models with diagrams* and translation to *platform-specific implementation languages*.

2 Related Work

2.1 Executable UML

BridgePoint [One] is a commercial executable UML implementation originally based on the Shlaer-Mellor method [SM96]. The solution has been open sourced [Exe] in 2014. 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.

Unlike our solution, BridgePoint's model editor is diagram and form-based with textual action code snippets residing in operations and state machines. The model storage format is a series of SQL-like statements, therefore specialized model compare and merge tooling is needed. Another difference is that BridgePoint uses an interpreter for model execution which is a performance limitation in automated mass test execution use cases. The modeling language used by BridgePoint is an early fork of UML, therefore it diverges from the standard at certain points. In our case, it is a design decision to keep compatibility with the latest UML standard [Obj15b].

Foundational UML, or fUML for short [Obj13b] is a subset of UML containing a limited set of class modeling features and activities with formal execution semantics defined in an OMG standard. The goal of fUML is to be a basis for defining precise semantics for richer UML subsets: The *Precise Semantics of Composite Structures (PSCS)* [Obj15a] is an OMG standard along these lines concentrating on classes communicating through ports and interfaces. There is another OMG standard that defines textual syntax for fUML called *Alf* [Obj13a].

We have considered adopting Alf as the standalone syntax in our project but as of now, Alf (and fUML) does not support state machines. (This is expected to change when the Precise Semantics of State Machines (PSSM) standard will be accepted and Alf will be extended based on the basis of PSSM.) Using our own syntax also helps us to keep the standalone syntax and the Java frontend close to each other, making it easier for users to switch between the two.

There are fUML and Alf reference implementations available. The fUML reference implementation is an interpreter written in Java, and follows closely the formal semantics definition of the standard. In fact, the implementation was part of the standardization work, and therefore execution performance was not a design goal. The Alf reference implementation is integrated in the Eclipse environment using Xtext for parsing and OCL [Tec12] for semantic checks. Alf execution is provided by transformation back to fUML activities and leveraging the fUML implementation. Consequently, efficiency of model execution was ignored in these implementations.

Moka [Mok] is an extension module of the open source Papyrus UML editor [Pap]. Moka simulates UML activity diagrams and provides basic debugging support such as breakpoints on actions. It uses the diagrams of Papyrus as graphical frontend for simulation and debugging, and uses a modified version of the fUML reference implementation for the execution logic. Recent work on the integration of Alf code fragments into graphical Papyrus models [ST15] promise BridgePoint-style executable model editing but Alf source level debugging is not yet available.

Moliz [MLMK13] is another 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. This project also uses the fUML reference implementation.

All above mentioned solutions use interpretation for model execution. The *xUML-RT Model Executor* [DKN+15] — which is a Papyrus-based model execution toolchain for the merge of the xtUML and UML-RT languages — provides model execution and debugging features based on generated code. That toolchain leverages Papyrus as model editor and uses incremental compilation techniques to translate the model to Java code, and reimplements some of the Xtext infrastructure to connect the generated code with the model for debugging purposes. Another difference from this paper's architecture is that xUML-RT's generated Java code is hidden from the users, therefore its API does not form an embedded language.

2.2 Textual modeling and Xtext

Textual modeling has many advantages over editing models in graphics. We have highlighted some of these in the introduction of this paper but for a deeper insight, we refer the reader to [GKR+07].

Creating models in text and generating graphical diagrams is a well known and applied technique. *Umple* [FBL10] and *eTrice* [eTr] are examples of modeling environments using this principle. Unlike txtUML, models in these tools lack abstract action language: Umple allows modeling code mixed in Java, PHP, C++ and Ruby and its code generator emits code in these languages, while eTrice allows action code in Java or C written in the models as *string literals* which are propagated to the generated code. None of the approaches allow execution and debugging on the model level. Diagram generation methodology is also different: Umple and eTrice use autolayout algorithms (Graphviz and KIELER), while txtUML allows the user to define the layout of diagrams using a concise DSL (see section 3.4). On the other hand, Umple and eTrice allow hybrid model editing (models can be edited both in text end graphics), while txtUML's generated diagrams are used for visualization purposes only.

Xtext [Xteb] is a framework designed for the development of domain-specific languages. *Xbase* [Xba] is an expression language — provided as part of Xtext — which can be customized and reused in other Xtext-based languages. It has many advanced features like type inference, syntactic sugar for anonymous functions, extension methods and type guards in *switch* constructions. We extensively use Xtext and Xbase to implement the standalone syntax of txtUML and provide executability and debuggability for it.

Xbase has also been used to create *Xtend* [Xtea] which is intended to improve expressiveness of Java by extending it with useful capabilities like type multi-dispatch method calls and operator overloading. *Java--*, created by Lorenzo Bettini et al. [Jav] is a simplified, educational dialect of Java. It uses a customized version of Xbase expressions to create a Java-like procedural language. We have studied the Xtend and Java-- implementations in order to find appropriate customization points of Xbase during the development of the standalone syntax of txtUML.

3 Architecture

3.1 Standalone Syntax

From now on, *XtxtUML* will stand for the standalone syntax variant (as an abbreviation of Xtext-based txtUML), whereas we will refer to the Java-embedded alternative as *JtxtUML* (for Java-based txtUML) which will be discussed in detail throughout section 3.2.

Essentially, the XtxtUML syntax can be considered syntactic sugar on top of JtxtUML as we map the elements of the former back the latter one. The base of this mapping is Xtext's built-in *JVM types* Ecore metamodel which is a sophisticated internal representation of the Java type system covering structural concepts such as class attributes and methods as well. As the compilation of its constructs to Java is predefined, in most of the cases merely specifying the connection between elements of our syntax and the JVM metamodel was sufficient to provide automatic code generation. The mapping itself is defined with the help of the framework's *JVM model inferrer* API.

One of the main advantages of using Xtext for implementing the standalone syntax variant is that, in this way, highly customizable Eclipse IDE support such as syntax highlighting, hyperlinking and reference lookup is provided out of the box by the framework. Validation for language elements can also be defined in a declarative manner. The aforementioned mapping makes it possible to use XtxtUML entities and their generated JtxtUML equivalents interchangeably across other XtxtUML or even Java sources.

Based on Xtext, not only structural but also behavioral parts of the new language can be implemented. For the latter, significantly more challenging task we heavily modified Xtext's reusable expression language — Xbase — both in its grammar and semantics to suit our needs. Due to the overall customization-oriented nature of the framework, it was even possible to extend Xbase with new expressions — e.g. signal sending and association navigation — by defining their syntax, type computation, compilation to Java and optional validation.

For a brief insight into XtxtUML, see the following example.

```
package examples.counter;

signal S;

class Sender {
    public void emit() {
        send new S() to this->(S_R.r).selectAny();
    }
}

class Receiver {
    private int count;

    initial Init;
    state Accepting;
    transition Initialize {
        from Init;
        to Accepting;
    }

    transition Accept {
        from Accepting;
        to Accepting;
        trigger S;
        effect { count++; }
    }
}

association S_R {
    hidden 1 Sender s;
    * Receiver r;
}
```

This simple model consists of two classes, `Sender` and `Receiver`, which are connected by the association `S_R`. When the `emit()` method of a `Sender` instance is called, it sends a new instance of signal `S` to one of the `Receiver` instances which are accessed by the aforementioned association. The arrival of the signal triggers a reflexive transition in the receiver which — as its effect — increments the counter containing the number of received signals.

One of the main design concepts of XtxtUML was to provide a clean and intuitive, Java-like syntax both for structural entities and action code. We believe that using this approach, not only is it easier for Java developers to become familiar with the language but the mapping to JtxtUML can be defined in a more straightforward way as well.

The next two subsections present interesting extensions of Xtext which have been used to implement our standalone syntax.

3.1.1 Handling package annotations

JtxtUML requires a `@Model` annotation on the root package of each model. As the Java specification allows only one annotated package declaration for a given package, this is conventionally done in a separate `package-info.java` source file. In the standalone syntax, we use a `model-info.xtxtuml` for the same purposes, that only contains a model declaration. Unfortunately — as of version 2.8.4 — Xtext does not support the mapping of language constructs into Java packages, so the model declaration could not be associated directly with a package info file. Instead, we infer a Java type from the model declaration and mark it with special meta-information. A modified version of Xtext's code generator handles this case to generate an annotated package description instead of a regular class.

3.1.2 Propagation of custom markers

Xtext is able to propagate errors and warnings from the generated Java code back to the original source element in the domain-specific language. As JtxtUML comes with its own validation rules and engine, it would be beneficial to use it also for validation of XtxtUML models. By default, Xtext propagates back only Java error and warning markers but not JtxtUML ones. Therefore, in XtxtUML a modified version of this mechanism was implemented by overriding the original marker propagation behavior.

The main problem with this approach is that due to Xtext's code generation policies, XtxtUML source files and their generated Java equivalents are mainly out of sync during editing, thus preventing the JtxtUML validator from issuing errors and warnings instantly. It is still an open question whether we should continue providing real-time XtxtUML validation at the expense of keeping two separate validators synchronized, or settle for the less user-friendly backpropagation solution.

3.2 Embedded Language

JtxtUML, our second syntax, is embedded in pure Java without any extensions or modifications to the host language, enabling users to write their models using only well-known language constructs and our API. The current implementation is based on Java SE 8, the newest version of Java, as we aimed to provide a convenient, fast, easy-to-read syntax and for this reason, we were ready to take advantage of any features that are provided by the Java SE.

As it was mentioned in previous sections, a JtxtUML model is also a runnable Java program in itself, therefore speed is indeed an important aspect here. Although creating a user-friendly API sometimes requires slight compromises on runtime performance, our experience so far is that the achieved performance is more than good enough for testing and debugging purposes.

The following short example is the same that is shown in section 3.1 but this time in JtxtUML.

```java
package examples.counter;

import hu.elte.txtuml.api.model.*;

class S extends Signal {}

class Sender extends ModelClass {
    public void emit() {
        Action.send(new S(), this.assoc(S_R.r.class).selectAny());
    }
}

class Receiver extends ModelClass {
    private int count;

    class Init extends Initial {}
    class Accepting extends State {}

    @From(Init.class) @To(Accepting.class)
    class Initialize extends Transition {}

    @From(Accepting.class) @To(Accepting.class) @Trigger(S.class)
    class Accept extends Transition {
        @Override
        public void effect() {
            count++;
        }
    }
}

class S_R extends Association {
    class s extends HiddenOne<Sender> {}
    class r extends Many<Receiver> {}
}
```

As it can easily be noticed, JtxtUML is more verbose than its counterpart, the XtxtUML syntax, but being an embedded language it has many advantages that make it a reasonable option to choose, like the aforementioned familiarity of Java developers or the off-the-shelf massive language support.

The example also shows the similarity of JtxtUML and XtxtUML which was an aim of our project as they are only syntactic variants of the same language with the ability to switch from one to the other, learning only minimal extra information.

In case of becoming familiar with JtxtUML, this extra information is mainly about the Java language elements we use to represent those UML features that are not present in Java.

To describe the structure of a model, no mutable language constructs (like variables) are used to prevent accidental modification of the model structure at runtime. This approach resulted in the fact that almost all model elements are represented by a Java type — a Java class, in most cases — with a special super type to show the kind of the particular element and also to inherit behavior which becomes important when executing models. To keep JtxtUML code free from string literals referencing model elements by name — making refactoring really hard —, we take advantage of Java reflection which let us refer to a type at runtime through its associated `java.lang.Class` object.

Annotations and generics (type arguments) are widely used as well to write static information in JtxtUML models. Annotations are suitable for adding data that is not always required (e.g. the trigger of a transition), explicitly naming properties (e.g. the `@From` and `@To` annotations) or containing primitive values (e.g. the `@Min` and `@Max` annotations which are used to write custom association end multiplicities; this feature is not presented in the above example). Generics can help to reference types when this information is also required at compile time, like in the case of association ends, as the `this.assoc` call has to return a collection of the desired type. These type parameters are retrievable at runtime as well because they are set in the declaration of a type and that can be inspected with Java reflection.

Despite these powerful features of Java, some limitations of the language proved extremely hard to overcome. Type erasure, to begin with, deprived us of many possibilities to write things in a simpler way. The lack of value types forced us to use immutable classes which can be inconvenient for the users too, as they also have to manually implement custom value types in an immutable and therefore verbose way. Garbage collection gives us no opportunity to force the deletion of objects from the heap or at least to check whether the user's code holds any references to them which would be helpful to effectively implement and dynamically validate model object deletion. The parameter passing rules of Java will make it challenging to implement UML's *out* and *inout* parameter passing modes. However, the greatest limitation seemed to be the single inheritance of Java, which made us unable to introduce multiple inheritance between model classes, which is allowed in UML. The default Java solution for this problem, the usage of interfaces, could not be applied here because Java interfaces are too limited in features to be used instead of classes and it would be very inconvenient for a user to create both the interface and the implementing class for a single model class.

In case of the action code, both our opportunities and requirements proved to be much less than in the case of the model structure. It is simple Java action code with the extension of public and protected methods of API types, most importantly, the class `Action`, whose static methods implement basic operations of JtxtUML, like sending signals, linking associations or deleting model objects.

3.2.1 Static Validation of the Embedded Language

Enhancing Java with the required UML features is only part of the task when defining an embedded language like JtxtUML as Java provides many tools that cannot be translated to UML at all or only if they are used with certain restrictions. Examples include casting, threading and synchronization, local and anonymous classes; not to mention the various features of the standard library or any other libraries written in plain Java which may only be accessed from JtxtUML in a well-controlled way, through external classes[1].

For this reason and to ensure the semantical correctness of the models as well, a validator is provided which uses the Java Development Tools [JDT] Eclipse plugin to parse and check JtxtUML models. The use of JDT instead of standard Java reflection is an unfortunate necessity which is further explained in the next section as we first faced the decision between these two options during the implementation of the model exporter.

3.3 Exporting UML2 models

For visualizing and compiling the models we decided to export them into standard UML model format. The generated UML models are used as an intermediate representation for compilation to other programming languages and they can also be processed by external tools.

The export process is currently implemented as a batch operation converting the whole model at once. It parses all Java source files and outputs an EMF-UML2 model. We tried two approaches for extracting information from txtUML models:

- *Java reflection and AspectJ*: This solution uses standard Java reflection to analyze the structural elements (for example classes, method signatures) of the code. However, Java reflection cannot provide information on method internals. Therefore we experimented with AspectJ to export operations. AspectJ can inject aspects (additional method calls) to predefined points in the Java code, and these aspects can collect the necessary information to complete the export.
- *Parsing*: In this case we parse the Java code using JDT [JDT] and walk through the abstract syntax tree in order to translate the txtUML model to an EMF-UML2 model.

We have found out some drawbacks of the first solution: In that case the system has to run methods to analyze their body, and each time a method call has parameters, dummy values need to be produced, which complicates implementation and makes it fragile. Furthermore, AspectJ caused inconveniences while running the Java debugger on the model, and interfered with debugging features provided by Xtext/Xbase for the XtxtUML syntax. As these problems became unmanageable, we switched to the second, JDT-based solution.

Another dilemma is about the representation of action code in the UML model. One possibility to encode behavior in UML is using *opaque behaviors*: These are just strings labeled with the name of the language they are written in. We decided not to use opaque behaviors for two reasons: Polluting the UML model with action code in XtxtUML or JtxtUML syntax would introduce non-standard elements, limiting the compatibility with third party tools. Also, a model compiler would have to parse and type check these opaque behaviors and do reference resolution, which introduces a lot of complexity. Therefore we have chosen the other possibility, namely UML activities. This provides standard and language-independent action code format. On the other hand, it

[1]External classes are not explained in detail in this paper for brevity.

requires nontrivial translation logic both in the exporter module and in the model compiler. It is also a threat that UML activities are extremely verbose, and this might lead to scalability problems in case of large models with much action code.

3.4 Diagram Generation

While textual modeling is beneficial in several aspects, we consider graphical diagrams extremely important for understanding models. For this reason we included a diagram generation module into the toolchain, which produces Papyrus diagrams on top of the exported EMF-UML2 models. As of now, class diagrams and state machine diagrams are supported.

The most important question of visualization is the layout. A popular solution is the application of autolayout algorithms. However, that is not ideal if users want to control the layout, possibly partially, and would like to store layout information under version control. To solve this problem we have created a small DSL, embedded in Java, to define diagram layout concisely. The following example shows a layout definition for the model presented earlier in sections 3.1 and 3.2:

```
public class ExampleDiagram extends ClassDiagram {
    @Left(val = Sender.class, from = Receiver.class)
    public class MyLayout extends Layout {}
}
```

This description requests the `Sender` class to be the left neighbor of the `Receiver` class. The resulting, generated Papyrus diagram is shown on image 4.

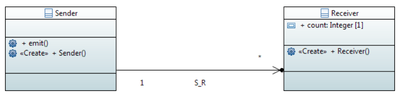

Figure 4: Generated class diagram

We have published the layout definition DSL and the diagram generation algorithm in [GGK$^+$15]. Here we give a short overview only: The language includes constructs to define the relative positioning of boxes on the diagram. The constraints are transformed to a linear inequality system of special form, that can be solved by the *Bellman-Ford* graph algorithm. Once the boxes are placed on the diagram, the links are laid out on a grid using the *A** algorithm with a cost function that minimizes length, number of crosses and turns.

3.5 Execution, Debugging and Animation

The architecture presented in this paper provides model-level execution for models written in any of the two syntactic variants. In case of the embedded language, the models are Java programs using the txtUML API and runtime library, therefore these can be executed and debugged in any Java development environment. For models in the standalone syntax, execution and debugging controls are provided by Xtext, based on the transformation to the embedded language. This includes breakpoint support, variable view, and session control functions like step over, step into, step out, resume and stop.

The model execution runtime library adds two useful features: runtime validation and state machine diagram animation. Runtime validation generates warnings, for example, when multiplicity constraints are violated or signals are dropped. This feedback helps the modelers to find bugs early in the development process, even without generating code and deploying it on a target platform.

The runtime behind txtUML API does have sophisticated tracing capabilities. These are switched off by default when the program is run as a plain Java application. If the extra functionality is switched on, the Eclipse-side plugin makes a connection towards the runtime in order to receive trace information. Data is provided even about individual object states and are fully kept track of.

The trace data is on one hand used to provide the user with sophisticated warnings, and, on the other hand to animate the generated state machine diagrams. This is achieved by the CSS capabilities of Papyrus. Because normal debugging features still work in this mode, breaking or reaching a breakpoint gives the possibility to examine various model states on the paused animation (see on Fig. 5, 6).

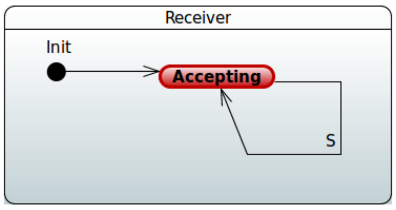

Figure 5: State machine is residing in a state

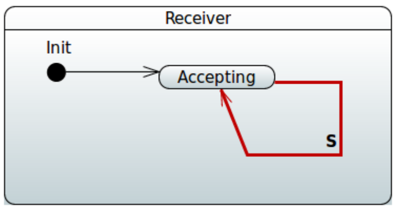

Figure 6: State machine is taking a transition

3.6 Compilation

We made a significant design choice by compiling from EMF-UML2 instead of using the original XtxtUML/JtxtUML code. The EMF-UML2 representation created from XtxtUML/JtxtUML code is a de facto standard and gives enormous flexibility for our tool. By using EMF-UML2 directly, compilation fits into the general exporting framework and can use all the benefits and generality of other export methods. Support for a new language, a new tool or even a graphical representation can be easily added the same way because the exporting mechanism does not rely on any specifics of the Java code. This gives a true independence between model execution/testing and the compiled code which makes the development more robust. Currently we support compilation to standard C++11 (tested on gcc, clang, msvc).

Despite the generated code being standard compliant, there are target deployment scenarios where the code generation has to be adjusted to certain needs. This includes inter-process/inter-machine set up, intra-process/thread pooling settings and also target platform capabilities. We have separated the compiled code into 3 different parts to support these:

- Code generated from models
- Support runtime
- Generated deployment configuration settings

By keeping these isolated, the generated code is practically easier to integrate. Deployment settings reside in a few specific files and do not pollute the model code which stays clean this way. Settings can be easily changed in the configuration files without recompilation. The support runtime can be freely interchanged by an other one.

We plan to support the versatile usage of the compiled code by spending efforts on adding more runtimes and developing a rich deployment configuration for multiple target platforms.

4 Conclusion

Figure 7 shows the main features of the proposed toolchain and the way they are connected to each other. The core functionality is the txtUML Java API and the underlying runtime library. This feature can be used in any Java development environment to build and execute models. Static validation rules check the soundness of the model structure and give instant feedback to the user. This feature is already dependent on the Eclipse framework. The same holds for the standalone syntax of the language and its editor, based on Xtext and Xbase. Export to EMF-UML2 representation and code generator uses certain Eclipse modules like JDT and the EMF-UML2 API, but it is possible to package these features into a command line tool. The diagram generation feature also builds on the generated EMF-UML2 model and it requires Eclipse and Papyrus.

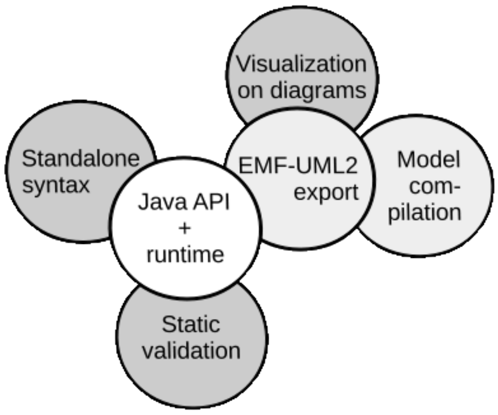

Figure 7: Features and their dependencies

This modularity of the architecture allows cherry-picking of the needed features for users and opens up replacement possibilities for the developers of the toolchain. For example, Papyrus, which is a graphical UML editor, is currently used only for visualization purposes. It would be simpler to use a UML library that generates read-only images: In fact, using the Papyrus APIs involves technical challenges. On the other hand, the emerging community around Papyrus and a future possibility to enable round-trip-editing are good reasons to use the UML editor for visualization.

A first step towards round-trip-editing, i.e. changes in text or on diagrams are seamlessly propagated to the other format, would be to make the generation of the EMF-UML2 model and the Papyrus diagrams *incremental* instead of the current batch transformation. This is one important future work.

Another open question is related to the two alternative text frontends we provide: Will users prefer one of those over the other one? If there is need only for the embedded Java syntax, then we can simplify the toolchain by removing the standalone syntax and its editor. On the other hand, if only the standalone syntax is preferred, then we can make the Java API internal, and gain much more freedom to simplify its implementation and boost performance. The third option is mixed usage or keeping the Java frontend as fall-back option. In this case a reverse transformation from embedded to standalone syntax will be useful.

After analyzing the available open source modeling tools described in the related work section, and experimenting with other possible architectures for executable modeling [DKN+15], we came to the conclusion that the solution proposed in this paper is lightweight and modular enough to meet industrial needs. The first pilot projects are now ongoing in collaboration with our industrial partner. By making the toolchain open source [txt] we intend to make a useful contribution for the modeling community.

References

[DKN+15] Gergely Dévai, Máté Karácsony, Boldizsár Németh, Róbert Kitlei, and Tamás Kozsik. UML Model Execution via Code Generation. In *1st International Workshop on Executable Modeling*, 2015.

[eTr] eTrice. `http://www.eclipse.org/etrice/`.

[Exe] Executable Translatable UML Open Source Editor. `https://www.xtuml.org`.

[FBL10] Andrew Forward, Omar Badreddin, and Timothy C Lethbridge. Umple: Towards combining model driven with prototype driven system development. In *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*, pages 1–7. IEEE, 2010.

[GGK+15] Balázs Gregorics, Tibor Gregorics, Gábor Ferenc Kovács, András Dobreff, and Gergely Dévai. Textual Diagram Layout Language and Visualization Algorithm. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 196–205. IEEE, 2015.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased modeling. In *4th International Workshop on Software Language Engineering*, 2007.

[Jav] Java--. `http://javamm.sourceforge.net`.

[JDT] Java Development Tools. `http://www.eclipse.org/jdt/`.

[MLMK13] 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.

[Mok] Moka. `http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`.

[Obj13a] Object Management Group. Action Language for Foundational UML (ALF), standard, version 1.0.1. `http://www.omg.org/spec/ALF/`, 2013.

[Obj13b] 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.

[Obj15a] Object Management Group. Precise Semantics of UML Composite Structures (PSCS), standard, version 1.0.
`http://www.omg.org/spec/PSCS/1.0/`, 2015.

[Obj15b] Object Management Group. Unified Modeling Language (UML), standard, version 2.5. `http://www.omg.org/spec/UML/2.5/`, 2015.

[One] OneFact. BridgePoint xtUML tool. `http://onefact.net`.

[Pap] Papyrus. `http://wiki.eclipse.org/Papyrus`.

[SM96] Sally Shlaer and Stephen J. Mellor. The Shlaer-Mellor method. *Project Technology white paper*, 1996.

[ST15] Ed Seidewitz and Jérémie Tatibouet. Tool paper: Combining alf and uml in modeling tools–an example with papyrus–. In *OCL 2015–15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings*, page 105, 2015.

[Tec12] 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.

[txt] txtUML: Textual Executable Translatable UML – Open source repository.
`https://github.com/ELTE-Soft/txtUML`.

[Xba] Xbase. `https://wiki.eclipse.org/Xbase`.

[Xtea] Xtend. `http://www.eclipse.org/xtend/`.

[Xteb] Xtext. `http://www.eclipse.org/Xtext/`.