|  |  |
| --- | --- |
| Annex 2 of D2.2 | The txtUML modeling tool |
| Access[[1]](#footnote-1): | **PU** |
| Type[[2]](#footnote-2): | **Report** |
| Version: | **1.1** |
| Due Dates[[3]](#footnote-3): | **M24** |
| *Open Cyber-Physical System Model-Driven Certified Development* | |
| **Executive summary[[4]](#footnote-4):** | |
| This document presents txtUML which is a textual modeling tool for software development according to the executable UML paradigm. It provides two textual notations for defining models which can then be executed, debugged, integrated into Java programs, visualized, animated, exported to a standard UML representation and translated to C++ code. The generated code can also be packaged to produce an FMU. | |

**Deliverable Contributors:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Name | Organisation | Primary role in project | Main Author(s)[[5]](#footnote-5) |
| Deliverable Leader[[6]](#footnote-6) | Jérémie TATIBOUET | CEA | Task Leader |  |
| Contributing Author(s)[[7]](#footnote-7) | Dávid János NEMETH | ELTE-Soft | Contributor | X |
| András NAGY | ELTE-Soft | Contributor | X |
| Zoltán Gera | ELTE-Soft | Contributor |  |
| Boldizsár NEMETH | ELTE-Soft | Contributor |  |
| Máté SZOKOLAI | ELTE-Soft | Contributor | X |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| Internal Reviewer(s)[[8]](#footnote-8) |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Document History:**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Reason for Change | Status[[9]](#footnote-9) |
| 0.1 | 10/11/2017 | Initial version of the deliverable | Draft |
| 1.0 | 17/11/2017 | Final version based on feedbacks | Final |
| 1.1 | 17/11/2017 | Apply remarks from Magnus | Final |

**Contents**

[**Contents** 3](#_Toc498932498)

[Abbreviations 3](#_Toc498932499)

[1 Overview 4](#_Toc498932500)

[2 Motivation 4](#_Toc498932501)

[3 Architecture 5](#_Toc498932502)

[3.1 Standalone syntax 7](#_Toc498932503)

[3.2 Embedded language 8](#_Toc498932504)

[3.2.1 Static validation of the embedded language 11](#_Toc498932505)

[3.3 Exporting UML2 models 11](#_Toc498932506)

[3.4 Diagram generation 12](#_Toc498932507)

[3.5 Execution, debugging and animation 13](#_Toc498932508)

[3.6 Compilation to C++ 14](#_Toc498932509)

[3.7 FMU export 15](#_Toc498932510)

[4 User guide 16](#_Toc498932511)

[4.1 Installation 16](#_Toc498932512)

[4.2 Sample models 16](#_Toc498932513)

[4.3 Creating own models 16](#_Toc498932514)

[4.4 Modeling language 17](#_Toc498932515)

[4.5 Generating diagrams 18](#_Toc498932516)

[4.6 Running and debugging models 18](#_Toc498932517)

[4.7 State machine animation 19](#_Toc498932518)

[4.8 Compilation to C++ 19](#_Toc498932519)

[4.9 FMU export 20](#_Toc498932520)

# Abbreviations

List of abbreviations/acronyms used in document:

**Abbreviation Definition**

API Application Programming Interface

CSS Cascading Style Sheets

DSL Domain-Specific Language

EMF Eclipse Modeling Framework

FMI Functional Mock-up Interface

FMU Functional Mock-up Unit

IDE Integrated Development Environment

JDT Java Development Tools

JVM Java Virtual Machine

UML Unified Modeling Language

# Overview

The name *txtUML* stands for textual, executable, translatable UML [1]. It is an Eclipse-based tool built on top of JDT [2], Xtext [3] / Xbase [4] and Papyrus UML [5]. The tool is designed for textual model editing. This makes storage, version control, compare and merge processes, editing and searching easier and more efficient.

The tool supports two textual syntaxes for modeling: the standalone syntax which is designed to be clean and short and alternatively, the txtUML Java API which can be used to define models as stan­dard Java programs. The tool supports the generation of graphical UML diagrams from the textual descriptions in the form of class and state machine diagrams. The layout of the diagrams can be controlled by a simple textual diagram layout language [6]. Models can be seamlessly integrated into Java programs, they can be executed and debugged. Generated state machine diagrams can be ani­mated during model execution to further enhance comprehension of model dynamics.

Compatibility with other tools is ensured by generating standard UML models in EMF-UML2 [7] for­mat. This representation is the input for our model compiler which generates C++ code, optionally packaged to produce an FMU.

# Motivation

Executable UML [8] models define both behavior and structure of software. These models can be exe­cuted, debugged and tested independently of the target platforms, providing early validation [9]. 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.

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 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.

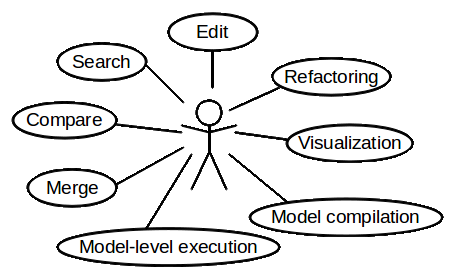
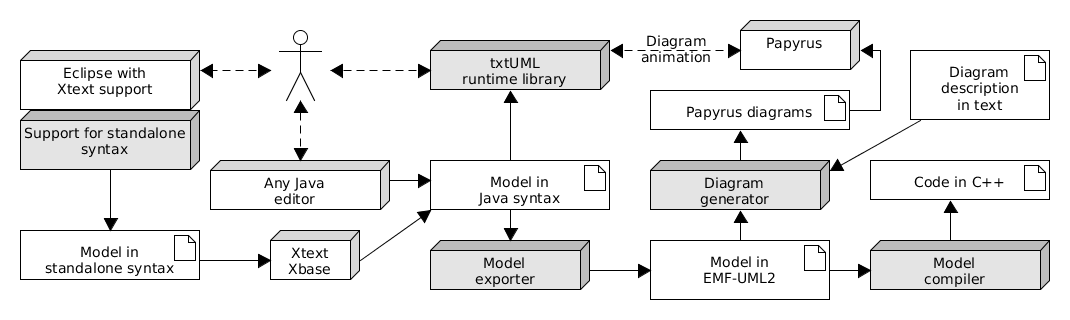
An executable software modeling environment must support many functionalities: a model editor with the graphical visualization of the model, tools for model compare and merge, a debugger with graphical animations, a model compiler, etc. Figure 1 depicts the many different use cases such a toolset is 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 UML modeling

Textual modeling [10] 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.

# Architecture

txtUML proposes a novel architecture for text-based executable software modeling, taking into ac­count the use cases and challenges discussed in Section 2. Figure 2 gives an overview of this archi­tecture. Dashed lines on the diagram denote interaction between two modules, while continuous lines represent input and output. Modules with gray background are developed in the txtUML project while white ones are independent components we rely on.

Figure 2: Overview of txtUML

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 Section 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 the 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, see 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, see 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.

## 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**->(SR.r).one();  
 }  
}  
**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** SR {  
 **hidden** 1 Sender s;  
 \* Receiver r;  
}

This simple model consists of two classes, Sender and Receiver which are connected by the asso­ciation SR. 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.

## 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.

**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(SR.r.**class**).one());  
 }  
}  
  
**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** SR **extends** Association {  
 **class** s **extends** HiddenEnd<One<Sender>> {}  
 **class** r **extends** End<Any<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 *in-out* 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.

### 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.

For this reason and to ensure the semantical correctness of the models as well, a validator is pro­vided which uses the Java Development Tools 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.

## 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 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 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.

## 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 or JointJS 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 Section 3.1 and Section 3.2:

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

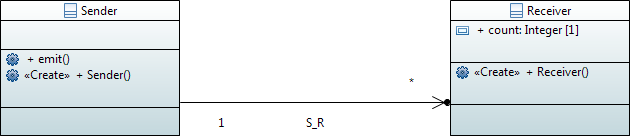
This description requests the Sender class to be the left neighbor of the Receiver class. The result­ing, generated Papyrus diagram is shown on Figure 3.

Figure 3: Generated Papyrus class diagram

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.

## 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 and modern web browsers. 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 Figure 4 and Figure 5).

|  |  |
| --- | --- |
| Figure 4: State machine is residing in a state | Figure 5: State machine is performing a transition |

## Compilation to C++

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*).

We separate the compiled code into the following parts:

* Code generated from the EMF-UML2 representation of models.
* Prewritten runtime with the following components:
  + Model executors, currently two is provided:
    - The single threaded executor which processes model messages synchronously in a central event loop and terminates when all of them are handled.
    - The multithreaded executor which deploys model objects to separate threads – ac­cording to the deployment configuration – and runs multiple event loops, not termi­nating even if there are currently no messages to process.
  + API based on JtxtUML action methods. It facilitates communication between models and the outside word to make integration with external applications easier.
  + State machine and event support with predefined base classes for model elements. This is to keep the amount of code needed to be generated to a minimum.
* Generated deployment configuration settings specifying thread usage – including object dis­tribution and scaling – and the type of model executor to be used.

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 another 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. Support for composite structures is released which is also make possible to model interprocess communication between components via ports.

## FMU export

Opposed to physical models usually described by differential equations, executable UML models are represented by classes and event-driven state machines. As the interface defined by the FMI standard is mostly suitable for models of the former type, transforming an UML model into an FMU is not a trivial task. An FMU wrapper has to be provided to manage exposed variables, life cycle and execution of UML models through FMI functions. An ideal solution should work for any exist­ing UML models without requiring additional modifications but unfortunately, in our current imple­mentation we had to make certain constraints on source models.

We have decided to explicitly represent the environment (or physical reality) with a predefined class even on model level. In each model a singleton object has to be specified which should connect it­self with the instance of the aforementioned environment – passed through its constructor – via an association. Simulation steps are controlled with preselected signals. At the start of a step, a signal containing the values of input variables in its attributes (the input signal) is sent to the model object. The simulation step ends when the model object sends a signal containing the values of output vari­ables in its attributes (the output signal) to the environment.

To keep the packaged FMU as lightweight as possible, we have decided not to use the XtxtUML/JtxtUML representation with our model executor written in Java as its base. Instead, we rely on the model compiler discussed in Section 3.6 to generate C++ code from the source model and then include the exported code in the FMU. This also makes implementing the necessary C functions of the FMI standard easier. In fact, the class representing the environment in the source model is only a stub which is handled differently during the code generation process, eventually be­ing transformed into the partially predefined implementation of our FMU wrapper.

Note that the wrapper is not entirely predefined: some parts are generated dynamically from the source model, e.g. which depend on input variables. In order to improve maintainability, we split the wrapper definition in half:

* An abstract base class containing attributes and functions which are required to ensure com­patibility with the C++ model execution runtime but are independent from FMI concepts.
* A subclass of the aforementioned base which implements FMI-specific features only. This is where FMI functions are defined according to the semantics specified above. For example, the fmi2DoStep function sends an input signal to the model object parameterized with the input variables stored in the wrapper. Similarly, when the wrapper processes an instance of the output signal it updates the output variables based on the received signal and calls the stepFinished callback to mark the simulation step as finished.

While this solution certainly works, it is unfortunate that UML models have to satisfy specific con­straints if we want to translate them into FMUs.

# User guide

This short user guide is intended to provide practical details about the usage of the tool. A more complete documentation can be found at the official website[[10]](#footnote-10) of the project. We also recommend watching our introductory video[[11]](#footnote-11). Alternatively, you can visit our GitHub repository[[12]](#footnote-12) as well.

## Installation

As txtUML is implemented as a set of Eclipse plugins, at first you have to install Java and Eclipse. Currently Java 8 and Eclipse 4.6.2 (Neon.2) is supported. For version 0.7.0, add the following up­date site in Eclipse (*Help > Install New Software... > Add...*):

<http://txtuml.inf.elte.hu/releases/txtuml-v070>

Select txtUML to be installed and let Eclipse guide you through the rest of the installation process. Restart Eclipse at the end of the installation process. In case of a successful installation, the txtUML menu appears in Eclipse's menu bar and there is a txtUML wizard category when selecting *File > New > Other...*.

## Sample models

For a quick start we recommend experimenting with the sample models[[13]](#footnote-13). Download and unzip the sample models. Import them into your Eclipse workspace (*File > Import... > General > Existing Projects into Workspace*). Clean and build the projects (*Project > Clean...*). Sample models are im­plemented using either XtxtUML syntax (see the source packages *<name of example>.x.model*) or JtxtUML syntax (see the source packages *<name of example>.j.model*), or both. In addition, the sample models are accompanied with diagram descriptions (see the Java classes inheriting from the ClassDiagram or StateMachineDiagram type).

## Creating own models

* New txtUML project: txtUML models should be placed in txtUML projects. A new txtUML project can be created by selecting *File > New > Project... > txtUML > txtUML Project* and setting the project name. By default, the project will be created in the current workspace. To override this, uncheck the *Use default location* checkbox and select a location for the new project.
* New txtUML model: Select *File > New > Other... > txtUML > txtUML Model*. Select a *Source folder* from an existing project for the new model. Select an existing *Package* from that folder or type a new package name. Type a *Name* for the new model. Select the *syntax* of the new model: XtxtUML for custom modeling syntax or JtxtUML for Java syntax. Both XtxtUML and JtxtUML models can be connected with Java code, can be run and debugged and used as a source for Papyrus UML model generation. A txtUML model is a package with either a *package-info.java* file (in case of JtxtUML) where the package has an annota­tion of the form @Model(“ModelName”) or a *model-info.xtxtuml* file (in case of XtxtUML) which has a model declaration of the form model-package example.x.model as “Mod­elName”;. All files in this package (and its subpackages) are part of the model. The wizard described above creates one of these files depending on the XtxtUML/JtxtUML selection.
* New model elements: For XtxtUML syntax, select *File > New > Other... > txtUML > Xtx­tUML File*. Fill in the source folder and package to place the new source file in, then enter a file name. You can also choose between the two possible extensions: *.xtxtuml* or *.txtuml*. For JtxtUML syntax, select *File > New > Class* to create a new Java class.

## Modeling language

The txtUML language covers a subset of UML. We summarize the supported elements below:

* Class modeling: classes with attributes and operations; simple binary associations; composi­tions; (single) inheritance.
* State modeling: simple and composite states; transitions triggered by signals; guards; choice states.
* Behavior modeling: action code can be written in operations of classes, entry and exit ac­tions of states and effects of transitions. Supported base types are int, double, boolean and String with the usual arithmetic and logic expressions, variables and assignment. Control structures (loops, branches), attribute access and operation calls are supported. UML-spe­cific actions: creation and deletion of objects; linkage and unlinkage via associations and connectors; reading links; sending signals; accessing signal data in entries, exits and effects.
* Component modeling: interfaces containing signals; ports; connectors.

The design of the two language variants follows a pattern: kinds of the model elements are shown by keywords (signal, class, transition) in XtxtUML while the Java version uses inheritance from Signal, ModelClass and Transition. These classes are provided by the txtUML Java API. Properties of the transitions are expressed by Java annotations (e.g. @From) in Java, while attribute-like syntax with keywords (e.g. from) is used in the standalone version. See the demo models or the Language Guide[[14]](#footnote-14) to study the txtUML language both in XtxtUML/JtxtUML. In case of JtxtUML, the JavaDoc[[15]](#footnote-15) of the API can also be used.

## Generating diagrams

It is possible to generate EMF-UML2 models together with Papyrus or JointJS diagrams from txtUML mod­els. Currently class and state machine diagrams can be generated. Content and layout of the class dia­grams and flat state machine diagrams can be defined by textual diagram descriptions.

The fol­lowing simple example assumes classes A, B, C and D in the model. We create a class diagram where classes A, B and C are in a row, and class D is below B. Diagram definitions can be written us­ing a Java API. See the Diagram Language Guide[[16]](#footnote-16) for a detailed description.

**public** **class** ExampleDiagram **extends** ClassDiagram {  
 @Row({A.**class**, B.**class**, C.**class**})  
 @Below(val = D.**class**, from = B.**class**)  
 **class** ExampleLayout **extends** Layout {}  
}

To generate diagrams, select *txtUML > Generate Papyrus diagrams from txtUML* or *txtUML > Generate JavaScript diagrams from txtUML* from the menu bar. Diagram descriptions are grouped by projects. You can select several descriptions – if the de­scriptions are related to different models, a separate Papyrus project will be generated for each indi­vidual model. Diagrams can be generated from a context menu as well, either in the *Project Ex­plorer* or in the *Package Explorer*. Simply right click on a diagram description file (you can select several descriptions too) and choose *Visualize as Papyrus diagram* or *Visualize as JavaScript diagram*.

## Running and debugging models

Models in txtUML can be run as Java applications with the help of a model executor which is part of the txtUML modeling API. While it is possible to write custom model executors, the default one will be sufficient in most cases. Model executors can be managed through the ModelExecutor in­terface. This interface has two static create methods to instantiate the default executor. The first is without parameters while the second one takes a single String argument as an optional name for the new executor instance. This name will appear in the automatic logs.

In the simplest case, a main Java class that solely executes a model would look like this:

**public** **class** Tester {  
 **public** **static** **void** main(String[] args) {  
 ModelExecutor.create().run(() -> {  
 MyClass instance = Action.create(MyClass.**class**);  
 Action.start(instance);  
 Action.send(**new** MySignal(), instance);  
 // ...  
 });  
 }  
}

The run method takes a Runnable instance, the initialization of the model execution which should create, link, start and send signals to the model objects which are required at the beginning of the model execution. This initialization code will run as part of the model, so any action that is allowed in the model is also allowed here.

The model executor writes log messages to the console and to a log file. Runtime errors and warn­ings are always logged but there is an optional trace logging which reports all important events dur­ing the model execution, for example, when a state machine of a model object leaves or enters a state. This trace logging is switched off by default but can be switched on with the setTraceLog­ging method. It is important to call this method before starting the model execution with the run method.

Models in txtUML can be debugged as well. Switch to Java or Debug perspective and create a new run/debug configuration of type *Java Application*. Breakpoints can be created and managed the same way as for Java programs. The standard debug controls (stop, pause, resume, step, step-into) work as usual. The variable view can show the current signal, current state, associations and the at­tribute values of the actual object.

## State machine animation

State machine diagrams generated by txtUML can be animated. Create a new run/debug configura­tion of type *txtUML Application*. Open the generated Papyrus or JointJS diagram and start the model either in run or in debug mode. For a JointJS diagram, the port displayed by the running model must be copied into the appropriate field in the browser then the switch must be turned ‘on’. The current state and currently executed transition gets highlighted. For each state machine diagram, the state changes of the first activated object of the corresponding type will be highlighted. An expected later improvement will make it possible to select the object to be ani­mated during the debug session.

## Compilation to C++

The C++ model compiler can be reached by selecting the *txtUML > Generate C++ code from tx­tUML menu*. To generate code, a txtUML deployment configuration must be specified. The runtime library contains only prewritten *.cpp* files so they can be used for other generated models too. A de­ployment configuration is a description of how the object instances will be distributed into different threads. This is a special class which is derived from the Configuration base class. The model classes can be grouped together and the events that arrive to classes belonging to the same group will be served by a configured thread pool.

For example, consider the following configuration:

@Group(contains = {A.**class**, B.**class**},

**class** DefaultConfiguration **extends** Configuration {}

This means that A and B will be served by the same thread pool and the remint classes will be grouped in the default group. A more complex example:

@Group(contains = {A.**class**, B.**class**}, max = 10, constant = 2, gradient = 0.5)  
@Group(contains = {C.**class**})  
**class** ExampleConfiguration **extends** Configuration {}

This means that instances of classes A and B are served by the same thread pool which contains two constant threads plus one for every 2 A or B instances created, but no more than 10. Instances of class C are served by another thread pool which contains only one thread (according to the default values).

The generated C++ code is saved in the *cpp-gen* folder of the selected project. Note that you might have to refresh the folder so that the newly generated files become visible in Eclipse. You can com­pile the generated files with any C++ compiler manually but we suggest using the generated *CMakeLists* file to create native “make files” that can be used in the compiler environment of your choice. It is recommended to create a new folder next to the generated files, where the build envi­ronment should be created. The compilation can be performed by the following command:

cmake -G <environment> -D CMAKE\_BUILD\_TYPE=<type> <path>

Where the parameters mean the following:

* <environment>: The chosen build environment. You can use the cmake --help command to list the possible build environments.
* <type>: The type of the build. Can be Debug or Release.
* <path>: The relative path to the generated *CMakeLists* file.

A concrete example:

cmake -G "MinGW Makefiles" -D CMAKE\_BUILD\_TYPE=Release ..

After translating a txtUML model to C++, you might want to create a *main.cpp* for testing purposes where you instantiate model objects and send signals to them. This can be achieved with our C++ runtime API. For more information, see the *main.cpp* files included with the demo models (look for the source packages *<name of example>.j/x.cpp* where they exist).

## FMU export

The FMU export functionality is not yet released but it can still be tested using a custom txtUML build. From our GitHub repository download tag *synchronized-fmu[[17]](#footnote-17)* and set up a development Eclipse according to our GitHub wiki page titled *Installation of the Development Environment[[18]](#footnote-18)*. Us­ing the *Launch Runtime Eclipse* run configuration you can start an Eclipse instance with FMU ex­port capabilities.

In this Eclipse instance import the *MoonLander* project from the *examples/tests* folder of the down­loaded branch into your workspace (*File > Import... > General > Existing Projects into Workspace*). Select *txtUML > Generate C++ code from txtUML*, specify the MoonLanderConfigu­ration class as deployment configuration, check the *Generate FMU* option and select the Moon­LanderFMIConfiguration class as FMI configuration. After choosing *Finish*, the *MoonLander* model gets exported as an FMU. The generated files are placed under the *cpp-gen* folder of the project and they are automatically complied under the *fmu\_build* folder. A *<modelname>.fmu* file is also placed here which is possible to run by the OMSimulator.

It is also possible to run the compiled FMU with our debugger tool. This takes a text file as input where each line represents a simulation step and contains values of input variables. The debugger instantiates the FMU (fmi2Instantiate) and for each line of the input file it sets input model vari­ables (fmi2Set), requests a simulation step (fmi2DoStep) and finally queries (fmi2Get) and prints output variables to the console. The debugger is automatically compiled along model files as the fmudebug executable to the same folder where other C++ files are built. Assuming that this direc­tory is *<project root>/cpp-gen/<model package>* and an input file named *input.txt* is defined in *<project root>*, from the folder of the generated files the debugger can be started with the following command:

./fmudebug ../../input.txt

Such an input file is provided for the *MoonLander* example as well. Feel free to experiment with its contents and see how the FMU reacts.

References

|  |  |
| --- | --- |
| [1] | OMG, "Unified Modeling Language (UML)," 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/. |
| [2] | Eclipse Foundation, "Java Development Tools," [Online]. Available: https://www.eclipse.org/jdt/. |
| [3] | Eclipse Foundation, "Xtext," [Online]. Available: https://www.eclipse.org/Xtext/. |
| [4] | Eclipse Foundation, "Xbase," [Online]. Available: https://wiki.eclipse.org/Xbase. |
| [5] | Eclipse Foundation, "Papyrus," [Online]. Available: https://www.eclipse.org/papyrus/. |
| [6] | B. Gregorics, T. Gregorics, G. F. Kovács, A. Dobreff and G. Dévai, „Textual Diagram Layout Language and Visualization Algorithm,” in *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, Ottawa, Canada, 2015. |
| [7] | Eclipse Foundation, "EMF-UML2," [Online]. Available: http://www.eclipse.org/modeling/mdt/?project=uml2. |
| [8] | S. J. Mellor and M. . J. Balcer, Executable UML: A Foundation for Model-Driven Architecture, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. |
| [9] | G. Dévai, M. Karácsony, B. Németh, R. Kitlei and T. Kozsik, "Execution via Code Generation," in *1st International Workshop on Executable Modeling*, Ottawa, Canada, 2015. |
| [10] | H. Grönniger, H. Krahn, B. Rumpe, M. Schindler and S. Völkel, „Textbased modeling,” in *4th International Workshop on Software Language Engineering*, Nashville, TN, USA, 2007. |

1. Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP. [↑](#footnote-ref-1)
2. Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report. [↑](#footnote-ref-2)
3. Due month(s) according to FPP. [↑](#footnote-ref-3)
4. It is mandatory to provide an executive summary for each deliverable. [↑](#footnote-ref-4)
5. Indicate Main Author(s) with an “X” in this column. [↑](#footnote-ref-5)
6. Deliverable leader according to FPP, role definition in PCA. [↑](#footnote-ref-6)
7. Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP. [↑](#footnote-ref-7)
8. Typically person(s) with appropriate expertise to assess deliverable structure and quality. [↑](#footnote-ref-8)
9. Status = “Draft”, “In Review”, “Released”. [↑](#footnote-ref-9)
10. [http://txtuml.inf.elte.hu](http://txtuml.inf.elte.hu/) [↑](#footnote-ref-10)
11. <https://youtu.be/LhcdAiFcRzw> [↑](#footnote-ref-11)
12. <https://github.com/ELTE-Soft/txtUML> [↑](#footnote-ref-12)
13. <http://txtuml.inf.elte.hu/wiki/lib/exe/fetch.php?media=v070:demo.zip> [↑](#footnote-ref-13)
14. <http://txtuml.inf.elte.hu/wiki/doku.php?id=v070:language> [↑](#footnote-ref-14)
15. <http://txtuml.inf.elte.hu/releases/txtuml-v070/api/java/> [↑](#footnote-ref-15)
16. <http://txtuml.inf.elte.hu/wiki/doku.php?id=v070:layout> [↑](#footnote-ref-16)
17. <https://github.com/ELTE-Soft/txtUML/releases/tag/synchronized-fmu> [↑](#footnote-ref-17)
18. <https://github.com/ELTE-Soft/txtUML/wiki/Installation-of-the-Development-Environment> [↑](#footnote-ref-18)