Textual Modeling Scalability Studies

Máté Karácsony[1,2], Gábor Ferenc Kovács[1], Dávid János Németh[1], Boldizsár Németh[1], Zoltán Gera[1], Attila Ulbert[1], Tamás Kozsik[1], Gergely Dévai[1,2]

[1]ELTE-Soft Nonprofit Ltd.

[2]Ericsson

October, 2015

1 Contents

1 Introduction

The goal of this study is to evaluate text-based technologies and given software projects regarding their applicability for executable UML modelling in an Eclipse development environment.

1.1 Availability of used software

- Case studies for this study:
  - http://modelexecution.eltesoft.hu/20151030/
- txtUML: Textual, executable, translatable UML
  - GitHub: https://github.com/ELTE-Soft/txtUML
  - Webpage with documentation and releases: http://txtuml.inf.elte.hu/
- xUML-RT Model Executor:
  - Webpage with documentation and releases: http://modelexecution.eltesoft.hu
- Java--: a simpler version of Java aiming to teach programming
  - GitHub: https://github.com/LorenzoBettini/javamm
- Xtend: modernized Java
  - Webpage with documentation and releases: http://www.eclipse.org/xtend/

1.2 Short summary of the findings

The runtime scalability study shows that both the Model Executor and txtUML are ready for automated mass testing regarding runtime performance. For small number of objects, the Model Executor is faster, but txtUML scales considerably better for large number of objects.

The txtUML feature study shows that basic functionality of the custom syntax (XtxtUML) is ready, but there are many time consuming tasks ahead of us: extending validation, customizing the syntax even more, customizing content assist, adding quick fixes.

There are two big tasks: (1) adding support to split models in multiple files, (2) making the model export and diagram generation incremental.

The Xtext/Xbase scalability shows that the technology can be used in practice for small and medium sized files. There are no limitations for the number of files with cross-references.

Regarding the export of action code, the proposal is to transform the action code to standard UML2 activities, but into separate models on a per class basis. Papyrus does not load these models unless needed, which is important for scalability.

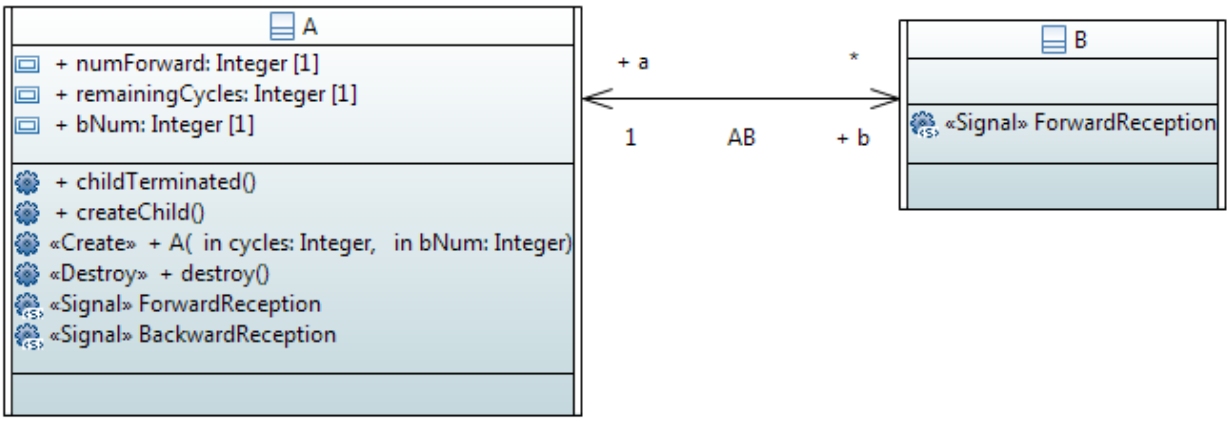
2 txtUML runtime scalability

This study evaluates the runtime performance of txtUML model execution different number of execution cycles and different number of objects created.

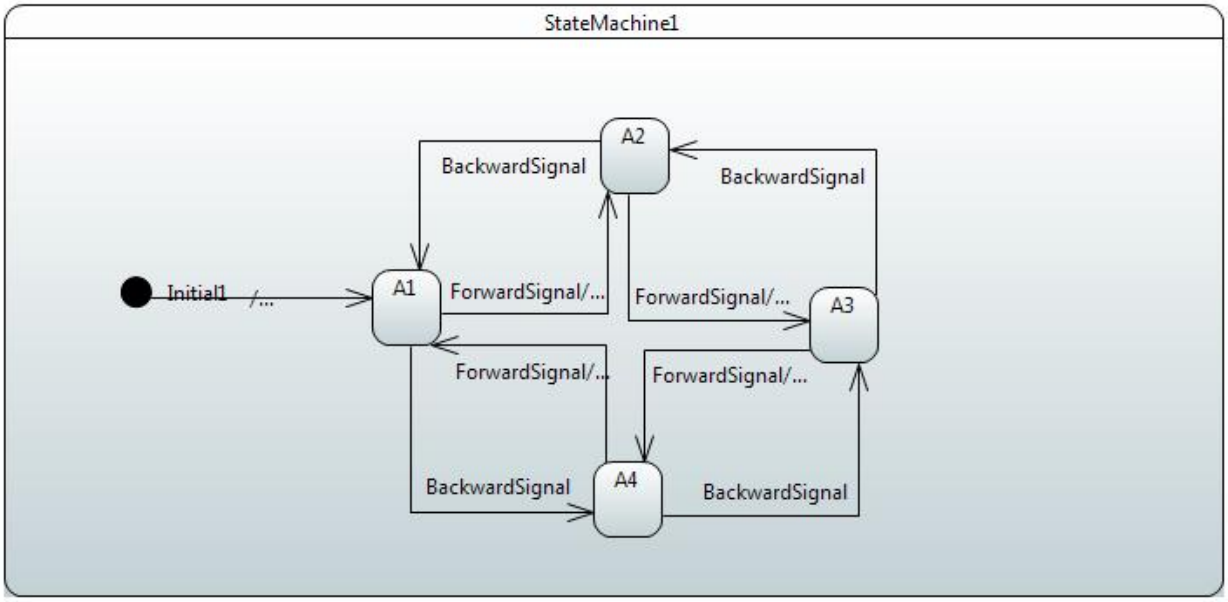
The model has 2 classes: A is a permanent singleton, while B-s are created and destroyed as the execution proceeds. A and B both have a state machine with 4 states. A's state machine is cyclical. The length of the measurement is given by the number of cycles A must perform. The instance of A sends signals to B-s to step on their state machines and also does backward movement when an object of B is destroyed.

The test model has two parameters: the number of object instances (N) of type B and the number of iterations (M). The model first creates N object instances and performs N link operations, then starts the iterations. Each iteration consists of 9 signal sending operations, the same number of state transitions, 1 object deletion, 1 object creation, 1 link and 1 unlink operations, 2 association navigations and the evaluation of 10 branch conditions. After the M iterations, all existing N objects of type B and their N links are deleted.

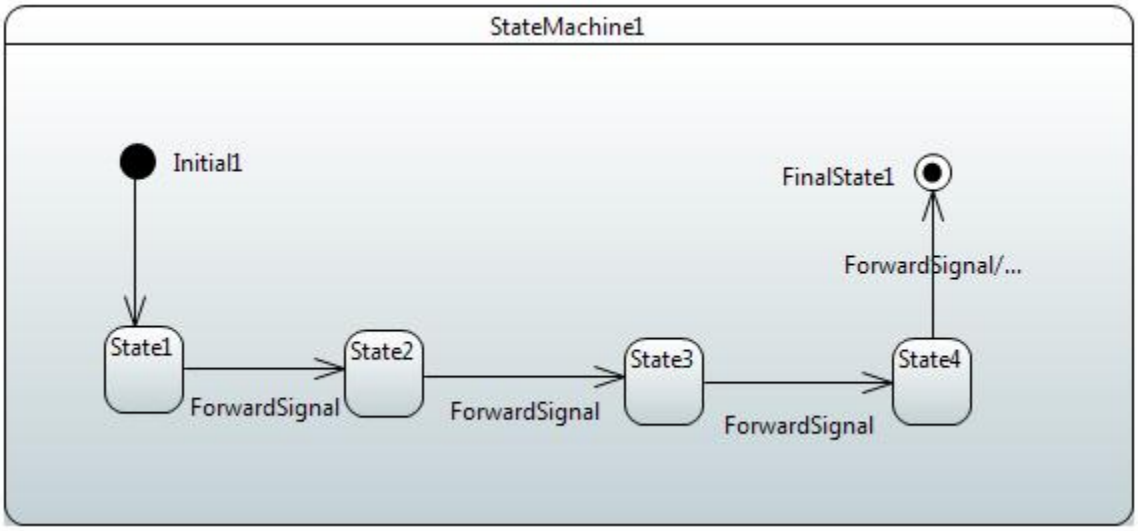
Class diagram of the model:



State machine of A:



State machine of B:



The same model is implemented both in Papyrus and in txtUML. The measurements were done on an Ericsson laptop: HP EliteBook notebook with Intel Core i5-3437U CPU @ 1.9 GHz and 8 GB RAM, running 64 bit Windows 7. The limit of each measurement was 11 minutes, the ">660" cells in the following tables denote longer (interrupted) experiments.

Model Executor results:

ME runtimes (sec)		Nr of objects			
		100	1000	10000	100000
Iterations	100	1	1	8	>660
	1000	1	2	19	>660
	10000	3	12	113	>660
	100000	9	67	>660	
	1000000	63	601	>660	
	10000000	>660	>660		

The results show that, up to 1000 instances and 1000 iterations, the model was executed within a couple of seconds. Up to 10000 objects and 10000 iterations, the execution time was under 2 minutes. This probably covers the volume of usual regression and nightly testing on model level within reasonable execution time. On the other hand, we expected better scalability when increasing the number of object instances with a fixed number of iterations. We plan to review the generated code and the runtime module to find the cause of the degradation.

txtUML results:

txtUML runtimes (sec)		Nr of objects				
		100	1000	10000	100000	1000000
Iterations	100	1	1	4	339	>660
	1000	2	2	5	404	>660
	10000	5	5	11	415	>660
	100000	20	25	38	>660	
	1000000	123	145	368	>660	
	10000000	>660	>660	>660		

For small number of objects (100), txtUML execution is approximately two times slower than the Model Executor, but is scales better for large number of objects: Already in case of 1000 objects it is faster than the ME and the difference is growing with the number of objects.

In summary, both tools are applicable for automated mass testing in practice.

3 txtUML feature study

Requirement	Status, effort needed
Custom expression syntax (eg. named parameters with '=>', navigation with '->', filter expressions)	Single step navigation is supported, easy to extend to navigation chains. Support for named parameters seems hard.
Custom type system (eg. collections, casts, primitive types)	Currently only one collection (with bag semantics) is supported, easy to extend to support other collections. Xbase implicit and explicit casts are supported in the frontend, but model export does not yet support any cast operations. Supported primitive types: integer, string, boolean. All Xbase/Java operations are available in the frontend, but model export supports just a few of these. Validation needs to be extended to limit the set of operations. Currently the syntax `int` and `boolean` is used instead of `Integer` and `Boolean`, expected to be easy to change.

Syntax highlight	Ready.
Content assist	Supported, but the lists are too long, needs to be customized. No technical difficulty expected.
Outline view	Ready, including custom icons.
Validation errors	Partial support. Technically easy, but time-consuming to complete.
Quick fixes	Not supported. Technically easy but time-consuming to implement all.
Breakpoint support	Ready.
Debug commands (run, step, pause etc.)	Ready.
Model export/import/transformation	Model export is supported into EMF-UML2. XtxtUML files are incrementally compiled to JtxtUML Java code. The syntax tree of this Java code is created and visited using JDT and the UML model is populated during the visiting process. Model import is currently not supported. There was an earlier case study to convert the structural part of EMF-UML2 models to JtxtUML. Conversion from JtxtUML to XtxtUML should be very easy to implement.
Incremental transformation	XtxtUML to JtxtUML and to Java bytecode transformations are incremental, therefore running XtxtUML models is like running Java programs in Eclipse. The EMF-UML2 export is not incremental. In order to have instant visual feedback during typing the code, this transformation has to be made incremental. Note that this is a major task and its priority needs to be carefully set compared to language feature implementation.
Support for graphical views	Papyrus diagram generation is implemented for class and state diagrams. The layout of class diagram can be defined in text. State diagrams have default (poor) layout. Making available the diagram descriptions for state diagrams is ongoing.

4 Xtext+Xbase editor feature scalability

Xtext is a framework to create editor and compiler support for textual languages in Eclipse. It provides means to define the language syntax as a context free grammar (with restrictions of LL parsing) and customize editor features.

Xbase is an implementation of a Java like expression language (with many extensions) to serve as a (customizable) default implementation for the expression layer of DSLs. It provides tight integration with Java and support for standard debugging features via incremental translation to Java.

Customization of Xbase is possible by deriving from its grammar, and by overriding and extending its default type computation and compilation mechanisms. The debugging support is provided directly by the expression compiler, by saving location information into trace files and source maps during code generation. The structural, non-Xbase parts of a DSL are compiled to Java by associating a Java construct for each AST node type of the DSL. The association process is often referred as inferring. This process is called by a special, replaceable Generator class, which is a part of Xbase. For each structural element, it infers the associated Java construct then compiles it. For each Xbase expression it calls the Xbase compiler that will compile Java code with location information. By replacing this generator, we are able to replace the whole Xbase infrastructure with a custom implementation which also provides debugging support, but in this case we have to reimplement the whole expression compiler with type computation.

The following scalability measurements were done on an HP EliteBook Folio 9480m laptop with 8GB memory and an Intel Core i5-4310U CPU, running 64-bit Windows 7. The tests were run under Eclipse Luna SR2, using Xtext version 2.8.4. Eclipse was configured to have 2GB heap memory, using the "-Xmx2048m" Java switch in eclipse.ini.

In the following tables the column "Opening" refers to opening only a single source file, not every file from the project. Column "Compilation" contains the full compilation time of the whole project into Java class files. The row headers are encoded in the format "*n*F*m*L", where *n* is the number of individual source files, and *m* is the number of lines in each source file. When there are multiple source files, they have random cross-references between each other.

The typing experience was always smooth, independently of the used language, file count or the number of lines. Syntax highlight was also instantaneous in case of keywords. However, semantic highlighting of Java built-in types took 1 or 2 seconds in all cases. Navigation inside a single source file was also immediate. Opening a cross reference in a different source file only took the same amount of time as opening the target source file without navigation.

From data below we can assume that most of the analyzed features are only depending on the length of the current file, not on the number of files. There are no significant differences between languages regarding to opening files, creating the outline, navigating between references and compilation to Java. There were several very small differences in the performance of context assist and validation. Currently XtxtUML has a slower scoping implementation than chosen reference languages. However, it

outperforms the others in validation speed, as many of the possible validation rules are not implemented yet.

4.1 XtxtUML

XtxtUML is a custom DSL syntax for txtUML, implemented via Xtext and Xbase. The experiment was done using branch “xbase-measure” of the corresponding GitHub repository.

	Opening	Outline	Context assist	Validation errors	Compilation
1F10L	1-2 sec	1 sec	2-3 sec	<1 sec	1-2 sec
1F100L	1-2 sec	1 sec	2-3 sec	<1 sec	1-2 sec
1F500L	1-2 sec	1-2 sec	2-3 sec	1-2 sec	1-2 sec
1F1000L	3-4 sec	2 sec	2-3 sec	1-2 sec	2-3 sec
10F100L	1-2 sec	1 sec	2-3 sec	<1 sec	3-4 sec
100F100L	1-2 sec	1 sec	2-3 sec	<1 sec	18-20 sec
500F100L	1-2 sec	1 sec	2-3 sec	<1 sec	1-2 min
1000F100L	1-2 sec	1 sec	2-3 sec	<1 sec	3 min

4.2 Java—

Java-- is a simpler version of Java aiming to teach programming. It customizes Xbase expressions and type system to make them look like and behave exactly as Java expressions and statements.

	Opening	Outline	Context assist	Validation errors	Compilation
1F10L	1-2 sec	1 sec	instant	1-2 sec	1-2 sec
1F100L	2-3 sec	1 sec	instant	1-2 sec	1-2 sec
1F500L	2-3 sec	1 sec	1 sec	1-2 sec	1-2 sec
1F1000L	3-4 sec	2 sec	2 sec	3-4 sec	3-4 sec
10F100L	1-2 sec	<1 sec	instant	1-2 sec	2-3 sec
100F100L	1-2 sec	<1 sec	instant	1-2 sec	8-9 sec
500F100L	1-2 sec	<1 sec	instant	1-2 sec	1-2 min
1000F100L	1-2 sec	<1 sec	instant	1-2 sec	2-3 min

4.3 Xtend

Xtend is a flexible and expressive dialect of Java, which is implemented over Xbase. Most of Xbase and Xtext itself is now written in Xtend.

	Opening	Outline	Context assist	Validation errors	Compilation
1F10L	1-2 sec	1 sec	instant	1 sec	1-2 sec
1F100L	1-2 sec	1 sec	instant	1 sec	1-2 sec

1F500L	1-2 sec	1 sec	<1 sec	1 sec	2-3 sec
1F1000L	1-2 sec	1-2 sec	1-2 sec	2-3 sec	2-3 sec
10F100L	1-2 sec	1 sec	instant	1 sec	1-2 sec
100F100L	1-2 sec	1 sec	instant	1 sec	14-15 sec
500F100L	1-2 sec	1 sec	instant	1 sec	1-2 min
1000F100L	1-2 sec	1 sec	instant	1 sec	2-3 min

5 Action code representation

This section discusses different ways to communicate action code from a textual model to a code generator module, whose basic input is an EMF-UML2 model. We will assume that the basic communication interface between the textual model editing frontend and the code generator is EMF-UML2 representation, because this architecture is standards compliant and opens up the possibility to pull in third party UML tools into the tool-chain in the long run. (If this is not a requirement, then any kind of “shortcuts” can be used to feed the code generator from the textual input.)

5.1 String in opaque behavior

When generating the EMF-UML2 from a textual model representation, it is possible to “cut” the action code fragments from the source and “paste” them into the UML model into opaque behaviors.

Pro:

- This is probably the most compact representation and the UML model generation is probably the fastest this way.

Con:

- The code generator will need an extra component to parse the opaque behaviors, type check them and connect the parsed elements to the rest of the UML model. This is waste of code and time, because the same capability and information is present on the text editor / model generator side.
- If the action code syntax is not standard, the model with opaque behaviors will not be fully standards based and only its structural part would be processed by third party UML tools.

5.2 UML activities

It is possible to translate the action code to UML activities. However, UML activities provide a verbose representation which results in large UML models, as shown by the following experiment.

The models of the experiment contain different number of classes (10, 50 and 100), and one association per two classes. Each class owns 10 attributes of different basic UML types and 10 operations with 14 lines of action code each (local variable, object creation, assignment operations, if, while). The same models were also generated with empty operation bodies for comparison.

Technically, the models were generated in XtxtUML syntax by a Java program then translated to EMF-UML2 models by the txtUML framework.

	10 classes		**50 classes**		**100 classes**	
	Size (Kb)	**Generation time (sec)**	**Size (Kb)**	**Generation time (sec)**	**Size (Kb)**	**Generation time (sec)**
With complete operations	6495	3	32475	18	64953	35
With empty operations	169	<1	844	1	1687	2

In case of 50 classes and above with complete operation bodies, the default 512 MB memory limit of Eclipse was not enough, and it had to be raised to 1024 MBs to make the model export possible.

Pro:

- Fully standards-based.

Con:

- Does not scale.

5.3 UML activities in separate models

It is possible to fragment an EMF-UML2 model. Activities of operation bodies and of state machine entries/exits/effects can also be separated, as the following example shows:

example_structure.uml, storing the structure of a model (two classes, an association and minimalistic state machine):

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001"
xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML" xmi:id="_ipySwHyQEeWMxelKhvei4w"
name="example_structure">
  <packagedElement xmi:type="uml:Class" xmi:id="_7K-JkHyQEeWMxelKhvei4w" name="A">
    <ownedAttribute xmi:type="uml:Property" xmi:id="_E-ZLcHyREeWMxelKhvei4w" name="a">
      <type xmi:type="uml:PrimitiveType"
href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#Integer"/>
    </ownedAttribute>
```

```
   <ownedBehavior xmi:type="uml:StateMachine" xmi:id="_KWkLMHyrEeWABrIzManC4A"
name="StateMachine1">
    <region xmi:type="uml:Region" xmi:id="_RvMloHyrEeWABrIzManC4A" name="Region1">
     <transition xmi:type="uml:Transition" xmi:id="_ZBkAAHyrEeWABrIzManC4A"
source="_Xje4cHyrEeWABrIzManC4A" target="_YAtnAHyrEeWABrIzManC4A"/>
     <subvertex xmi:type="uml:Pseudostate" xmi:id="_Xje4cHyrEeWABrIzManC4A" name="Initial1"/>
     <subvertex xmi:type="uml:State" xmi:id="_YAtnAHyrEeWABrIzManC4A" name="State1">
      <entry xmi:type="uml:Activity" href="example_activity.uml#_kDgDcHyrEeWABrIzManC4A"/>
     </subvertex>
    </region>
   </ownedBehavior>
   <ownedOperation xmi:type="uml:Operation" xmi:id="_KI0B4HyREeWMxelKhvei4w" name="f">
    <method xmi:type="uml:Activity" href="example_activity.uml#_l1Bi4HyUEeWMxelKhvei4w"/>
   </ownedOperation>
  </packagedElement>
  <packagedElement xmi:type="uml:Class" xmi:id="_HwUcUHyYEeWMxelKhvei4w" name="B"/>
  <packagedElement xmi:type="uml:Association" xmi:id="_JCIpgHyYEeWMxelKhvei4w" name="AB"
memberEnd="_JCIpgXyYEeWMxelKhvei4w _JCICcHyYEeWMxelKhvei4w"
navigableOwnedEnd="_JCICcHyYEeWMxelKhvei4w _JCIpgXyYEeWMxelKhvei4w">
   <ownedEnd xmi:type="uml:Property" xmi:id="_JCIpgXyYEeWMxelKhvei4w" name="a" type="_7K-
JkHyQEeWMxelKhvei4w" association="_JCIpgHyYEeWMxelKhvei4w">
    <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_JCIpgnyYEeWMxelKhvei4w" value="1"/>
    <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_JCIpg3yYEeWMxelKhvei4w"
value="1"/>
   </ownedEnd>
   <ownedEnd xmi:type="uml:Property" xmi:id="_JCICcHyYEeWMxelKhvei4w" name="b"
type="_HwUcUHyYEeWMxelKhvei4w" association="_JCIpgHyYEeWMxelKhvei4w">
    <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_JCICcXyYEeWMxelKhvei4w" value="1"/>
    <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_JCICcnyYEeWMxelKhvei4w"
value="1"/>
   </ownedEnd>
  </packagedElement>
</uml:Model>
```

example_activity.uml, storing the two activities referenced from the structure above:

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001"
xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML" xmi:id="_d8H-4HySEeWMxelKhvei4w"
name="example_activity">
  <packagedElement xmi:type="uml:Activity" xmi:id="_l1Bi4HyUEeWMxelKhvei4w" name="Activity1">
   <specification xmi:type="uml:Operation"
href="example_structure.uml#_KI0B4HyREeWMxelKhvei4w"/>
  </packagedElement>
  <packagedElement xmi:type="uml:Activity" xmi:id="_kDgDcHyrEeWABrIzManC4A"
name="EntryActivity">
  </packagedElement>
```

```
</uml:Model>
```

The syntax of referencing an entity stored in another model is highlighted in red.

When opening the first model in Papyrus, the second model is not loaded automatically, only if the user navigates to a model entity in the referenced model. We have verified this lazy loading behavior by removing read permission from the second model and experimenting in Papyrus to see which action triggers the read error.

Considering the results of this and the previous section, the proposal is to keep the activities in separate models, one such model per class. The structure of the model can be kept in one model (except for really huge models where separation of the structure could be done on package basis). The models with the activities are not even needed for visualization purposes and to navigate in the structure of the visual model. These heavy-weight models should only be generated if the resulting UML model will be used as input to the code generator.

Pro:

- Fully standards-based.
- Scales better. (To be confirmed by changing the txtUML model export function to work this way.)
- No need for parsing on the code generator side.

Con:

- UML activities are verbose representation.

5.4 Custom metamodel for activities

It is also possible to create a custom metamodel for activities. When generating the EMF-UML2 model from the textual model, this metamodel can be populated to store the action code snippets of the model. These instance models can also reference elements from the EMF-UML2 model.

This solution is a variant of the previous one, but uses custom metamodel for action code instead of UML activities.

Pro:

- Custom metamodel can be less verbose.
- No need for parsing on the code generator side.

Con:

- Not fully standards based.