oAW xText: A framework for textual DSLs

Version 1.1, September 22, 2006

Sven Efftinge, Flensburg, Germany, www.efftinge.de, sven@efftinge.de

Markus Völter, voelter – ingenieurbüro für softwaretechnologie, Heidenheim, Germany, www.voelter.de, voelter@acm.org

1. Introduction

Model-Driven Software Development [SV06] consists of two major aspects. The first one is processing models, i.e. checking their validity, transforming them into other models as well as generating code (and other textual artifacts) from models. The other aspect addresses the creation of models. Traditionally, the processing of models has received more attention from the MDSD community. Recently, the community has started addressing the model creation aspect (beyond using UML profiles…). In the Eclipse world, the Graphical Modeling Framework is a tool that allows developers to easily define graphical editors for EMF-based meta models. Graphical editors are not enough, though. Many problems are better described with textual concrete syntaxes.

As part of the Eclipse Modeling Project, there's a placeholder project called TMF (for Textual Modeling Framework) which will address exactly this challenge – defining "nice" textual syntaxes for EMF-based meta models.

In this paper we describe xText, a framework for building textual editors that is part of openArchitectureWare. We also want to put forward xText as a initial submission for the Eclipse TMF project. The oAW team will be glad to help writing the proposal.

After a short introduction of xText in general, we will introduce the tool based on a simple example.

2. What is xText

xText is part of the openArchitectureWare [oAW] project (which is in turn part of Eclipse GMT). oAW provides a set of tools to develop MDSD infrastructures; it helps with meta modeling, constraint checking, code generation and model transformation. More recently a framework has been developed that supports the creation of textual domain-specifc languages (DSL): xText

2.1 Features

Based on an EBNF-like notation, xText generates the following artifacts:

- a set of AST classes represented as an EMF-based metamodel
- a parser that can read the textual syntax and returns an EMF-based AST (model).
- a number of helper artifacts to embed the parser in an oAW workflow
- an Eclipse editor that provides syntax highlighting, code completion, code folding, a configurable outline view and static error checking for the given syntax (the constraints have to be formulated separately by the developer using oAW's OCL-like Checks language).

2.2 Parsing vs. Linking

Note that xText starts from a description of a textual syntax (the grammar) and derives an AST class model (the metamodel) from that concrete syntax definition. The linking of cross references within the same model or through different models, can be done separately from the textual syntax description. Linking can be a quite complicate process if you consider scopes, namespaces and visibility of elements We think that it is crucial for a textual language framework to allow the separation of parsing and linking.

The separation of these two concerns (parsing and linking) helps to implement more sophisticated linking logic independent of the concrete syntax. Additionally we can check the AST before doing additional linking and transformations. In some cases you even don't want to link references up-front, but want them to be looked up dynamically (late binding).

Linking in Xtext can be done in several ways. The easiest way is to make use of so called extensions. Extensions are operations that can be annotated to existing meta classes (see the following example). Another solution is to transform the AST to a "real" meta model. This has the additional advantage that the concrete syntax can be changed, or one can have several different concrete syntaxes (e.g. graphical and textual) for the same metamodel. The necessary transformation is relatively straight forward to define, because it is basically a one to one mapping with some additional linking logic An example can be found in [VKEH06]

3. Example

3.1 Introduction

In this example we will reimplement an exercise taken from the best-selling book "The Pragmatic Programmer" written by Andy Hunt and Dave Thomas [HT99]. In the third chapter "The Basic Tools" the authors motivate the reader to learn a "text manipulation language" (like Perl or Ruby) in order to be able to develop little code generators. The proposed exercise is very simple and code-centric and we will show how it can be implemented with oAW, specifically, the xText framework.

3.2 The exercise

The exercise consists of the following challenge:

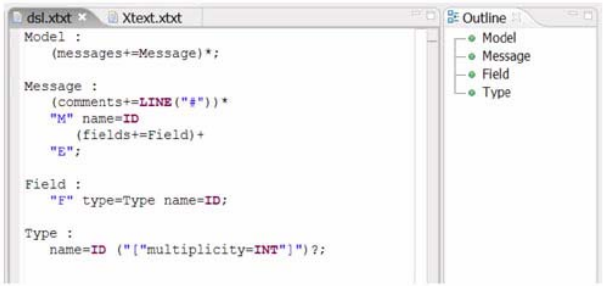
Write a code generator that takes the following input, and generates output in two languages of your choice. Try to make it easy to add new languages.

```
# Add a product
# to the 'on-order' list
M AddProduct
F id           int
F name         char[30]
F order_code int
E
```

In the original prag prog book, the solution is based on an ad hoc Perl program. In this paper we show the implementation based on xText.

3.3 Generating a parser

We will use the grammar language provided by xText. The following screen shot shows how the syntax is described for the DSL shown in the exercise above[1]. In fact language and tooling used for describing DSL syntax is bootstrapped, i.e. it is implemented using the xText framework itself. Bootstrapping is a common technique in the field of language and compiler development. If you can bootstrap your language and tools, this proves a certain level of maturity of the tools.



3.4 What's in a grammar?

An xText grammar consists of a number of rules (`Model, Message, Field` and `Type`). A rule is described using sequences of tokens. A token is either a reference to another rule or one of the built- in tokens (`STRING, ID, LINE, INT`). xText automatically derives the meta model from the grammar (remember that the meta model is basically a data structure whose instances represent the structure of sentences in the language). A rule results in a meta type, the tokens used in the rule are mapped to properties of that type (`comments, name, fields`). There are different assignment operators used in our example. The equals sign (`'='`) just assigns the value returned from the token to the respective property (the property will have the type of the token) and `'+='` adds the value to the property (the property will have the type *List<tokenType>*).

The first rule in the file (`Model`) is called the entry rule. It acts as a container for all the `Messages` contained in our input. The property that holds the list of messages is called `messages`. A `Message` starts with any number (note the quantifier '*') of comments, while each comment in turn starts with a hash '#' and ends with the end of the current line. This is described with the built-in token type `LINE(startpattern)`. Each comment is added (hence the use of `+=`) to the `comments` property of the current `Message`. The message's body starts with the keyword "`M`" (nice keyword, isn't it? ;-)) followed by an `ID` which acts as the name of the message.

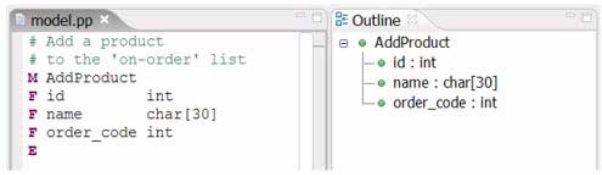
An `ID` is also a built-in token, which must correspond to the regular expression '[azAZ_]+[azAZ_09]*', which means that it is basically a sequence of characters and digits, as well as the underscore. Additionally it must not be a keyword, so '`M`' and '`F`' are not allowed as a message's name (note that this is automatically enforced by *xText*).

A `Message` contains at least one (quantifier '`+`') field, each `Field` starting with the keyword '`F`' followed by a type (described in it's own rule) and the field's name.

A `Type` can either be a simple type, or optionally (quantifier '?') an array. If this is the case the property `multiplicity` will be set to the specified number.

3.5 What do we get?

We get a nice Eclipse editor.



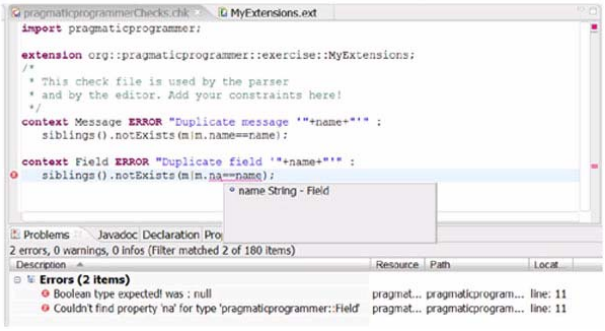
So we have described the DSL with a few lines of code. And, yes, that's all! In contrast to the Perl solution we do not have a simple event-driven parser, but a DOM-like parser that creates an actual AST (an object graph in memory) for our model.

In addition, the xText tooling (which basically consists of invoking an oAW-based generator to build the necessary artifacts) has generated a textual editor plug-in for Eclipse, which offers features such as syntax highlighting, an outline view and real time validation. Let's have a look at it.

3.6 Static checking

The Perl solution from the book doesn't have any support for validating models written in our DSL. So errors won't be found until the parser runs, or even worse, when the data is used in some way. However, what we really want is that the program is checked for syntactically and semantic correctness before it is processed any further, because it makes no sense to process an invalid program. It would also be really helpful if we could get feedback about invalid models/programs while editing them - just as we're used to when editing e.g Java code. openArchitectureWare comes with a language called *Check* which is specialized for model validation. *Check* is very similar to OCL [OCL], but uses oAW's expressions sub-language. If you want the parser and editor generated by xText to validate your models, all you have to do is add the respective constraints to a *Check* file. Here it is:

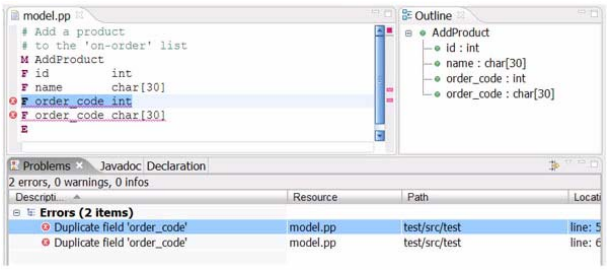
[1] We use a screen shot because we want to illustrate that xText also offers tool support for it's grammar language.



The screen shot also shows the tool support for the *Check* language. As *Check* is statically typed, the editor supports static type checking as well as code completion proposals. But let's talk about the language itself. Basically a check file consists of a number of constraints. A constraint starts with the keyword `context` followed by the name of the type for which this constraint must hold.

The severity is expressed using one of the keywords `ERROR` or `WARNING`. The message (e.g. "`Duplicate message '"+name+"'`") can be any valid expression as long as it returns a String. The constraint expression itself is defined after the colon. This expression must evaluate to a boolean. All constraints must be *true* in order for the complete model to be valid. Finally, each constrain is terminated by a semicolon. There is an implicit *this* variable pointing to the current element to be checked. As in OO programming, the `this` can be omitted, so we can just write `name` instead of `this.name` to access the `name` property of the element.

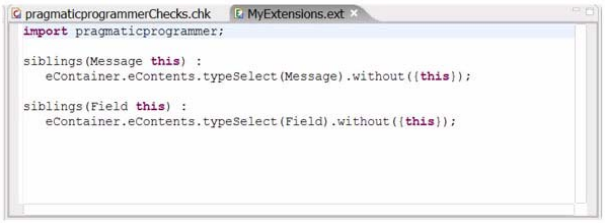
The first constraint makes sure that the name of each message is unique within the model (in our case, the text file). There is an extension (more about extensions in a minute) returning all siblings of the current message. We then ensure that messages have unique names by checking whether there is a message with the same name among its siblings. The way we do this is by invoking `notExists(m|m.name==name)` on the list of siblings. `notExists` is one of several higher-order functions for collections, it takes a predicate and evaluates it for each element of the list. The current element is temporarily bound to the local variable `m`. `notExists` returns true, if the predicate returns `false` for each of the elements it iterates over. Here is a screen shot of our generated editor having found constraint violations:



3.7 Extensions

Extensions are used to add behaviour, derived properties or other aspects to your meta types without touching the original definition of the meta type. They are defined using the xTend language which is also based on oAW's expressions language. xTend also has features for model-to-model transformations, but since we don't need them here we won't explain them.

To be able to use extensions in constraints, you have to import the extension file that defines the required extensions into the Checks file. This is done using the extension keyword. The extensions used in our checks look as follows:



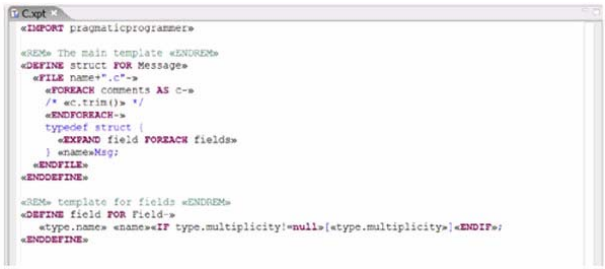
The siblings extension simply selects all the children (*eContents*) of the current element's parent (*eContainer*). From that list, we remove the element on which we've called the siblings function - we only want the element's siblings, not the element itself. Removing elements from a set is done using the without operation, the expression {*this*} creates a list containing just one element: this.

3.8 Generating Code

Code generation is not directly related to the xText framework – you can generate code from models independent of how they have been created. We'll nonetheless show code generation in this paper in order to show how the generated parser is integrated into a complete MDSD tool chain.

Also, earlier we talked about first transforming the AST-based model into a "real" domain model using model-to-model transformations. For non-trivial scenarios this is what is typically done. However, in our simple example here we'll generate code directly from the AST-based model.

Since our parser creates an object graph representation of our model (the abstract syntax tree), our generator is not written against events but against meta types from which the AST is built. To generate the actual code, we use xPand, a template language also based on openArchitectureWare's expression language. Therefore you can use the meta model, expressions and extensions we have seen in the previous section. Here are the templates:



An xPand template file consists of one or more templates declared using the *DEFINE* keyword. A template has a name (such as *struct*) and is bound to a specific meta type (for example, *Message*). This binding is important, since, whenever you execute the template, there's an implicit this variable that points to an instance of that meta type, the element, for which the template is currently executed. Inside the main template struct there is a *FILE* block which opens a file with the specified

name (*name+".c"*). All the text that is generated inside that *FILE...ENDFILE* block is written to that file. The usual control structures such as *FOREACH* and *IF* are available. You can call other templates using the *EXPAND* statement, passing the context element along (an instance of the type for which the target template is defined).

The generator description is not necessarily shorter, but it's much more readable because you can see what the resulted code actually will look like. In real life, code generators are often more complex than this one so I really don't want to see complex ones written in Perl the way the authors suggest. Note, that again we have rich tool support (static checking, code completion, etc.) for xPand templates.

3.9 Binding things together

In order to run the example, we need to define a process that

- first invokes the generated parser to instantiate an AST from our model (text file)
- and then passes that AST to the xPand engine in order to generate the target code.

Again, oAW offers a specialized tool for that purpose: the workflow engine. Generator workflows are described in XML. The language is actually a mixture of the Spring configuration language and ant. This is how our workflow looks like:

```
<workflow>
    <property file="gen.properties"/>
    <component file="org/pragmaticprogrammer/exercise/parser/Parser.oaw">
       <metaModel id="mm" class="oaw.type.emf.EmfMetaModel">
          <metaModelFile value="pragmaticprogrammer.ecore"/>
       </metaModel>
       <modelFile value="${model}"/>
       <outputSlot value="m"/>
    </component>

    <component class="oaw.xpand2.Generator">
       <metaModel idRef="mm"/>
       <expand value="gen::C::struct FOREACH m.messages"/>
       <outlet path="${gendir}"/>
    </component>
</workflow>
```

A workflow is a sequence of so-called workflow components. We can also define properties (just like in ant). The workflow engine is a dependency injection container, so it is responsible for creating the components and wiring them together. Therefore we have to specify the concrete class that should be instantiated as the component (it has to posses a default constructor). This is done using the class attribute. Alternatively we can refer to another workflow description using the file attribute.

In the workflow shown above we first import a properties file that contains two properties:

```
model=model.pp
gendir=src-gen/
```

Then we define two components. The first one is the parser. As you can see we refer to another workflow file (*.../Parser.oaw*). It has been generated by xText and includes the parser and the check component (checking the very same constraints that are also checked interactively in the editor). So we don't have to care about the internal details, we just have to use the provided black box (we call such a black box a cartridge). The cartridge needs a number of parameters. The first parameter passed to the parser cartridge is called metaModel. It has a class attribute itself, so the workflow engine takes care of creating a respective object and injecting the nested property (*metaModelFile*) through a respective setter method (*setMetaModelFile(java.lang.String)*). The other two parameters for the parser cartridge are *modelFile* and *outputSlot*. The *modelFile* points to the text file that contains the model we want to parse.

The second one requires slightly more explanation. Components communicate with each other using so called slots. In our example the parser reads a model, creates an object graph and stores it in a slot called m. Slots are a lot like variables. So when subsequently the xPand generator is invoked we access the model in the slot using the slot name as a variable in the expression that starts the generation process: *gen::C::struct FOREACH m.messages*.

3.10 Running the Workflow

There are different ways to execute a workflow. If you use Eclipse, the simplest way is probably to use the launch short cut provided by the openArchitectureWare plug-ins. Alternatively you can use the good old command line or an ant-task included in oAW. A suitable Maven2 target is on it's way.

4. Conclusion & Future Work

The current implementation of xText, especially when integrated with the rest of oAW, provides a workable and useful solution for textual modelling languages that has been used in several industry projects. However, it also has a number of shortcomings that will be addressed in future versions[2]:

- The current implementation of the code completion feature is not sufficient. A more context-specific and customizable implementation is necessary.
- In order to be able to split a big model into several text files (or any other kind of model file, such as GMF or EMF models) some mechanism of referencing or including other model files is necessary.
- It is currently not possible to include expressions in xText editors, thereby limiting the kinds of things that can be modeled using an xText editor. Adding expression support is thus an important todo.

5. References

[HT99] Hunt, Thomas, The Pragmatic Programmer, Addison-Wesley, 1999

[oAW] http://eclipse.org/gmt/oaw

[OCL] http://www.klasse.nl/ocl/

[SV06] Stahl, Voelter, Model-Driven Software Development, Wiley, 2006

[VKEH06] Voelter, Kolb, Efftinge, Haase : From Frontend To Code http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html

[2] If and how these things will be implemented also depends on how the Eclipse TMF project proceeds and the role xText plays in it