Migrate from oAW to TMF Xtext

Chapter 1. From oAW to TMF

TMF Xtext is a complete rewrite of the Xtext framework previously released with openArchitectureWare 4.3.1 (oAW). We refer to the version from oAW as oAW Xtext whereas the current Xtext version that is hosted at Eclipse.org will be called TMF Xtext to avoid confusion. oAW Xtext has been around for about 2 years before TMF Xtext was released in June 2009 and has been used by many people to develop little languages and corresponding Eclipse-based IDE support.

TMF Xtext has been improved in many aspects compared to the former version. While it integrates far better into EMF, it offers new fundamental features as well. The overhauled architecture leads to better performance when working with large models and since the whole framework is wired via dependency injection it is highly customizable. Last not least a test coverage of more than 2.000 unit tests provide confidence in the overall quality of TMF Xtext. We have been using the framework in production environments since one of the earlier milestones.

In this document we want to share the experience we made when migrating existing Xtext projects. The document describes the differences between oAW Xtext and TMF Xtext and is intended to be used as a guide to migrate from oAW Xtext to TMF Xtext. For people already familiar with the concepts of oAW Xtext it should also serve as a shortcut to learn TMF Xtext.

1. Why a rewrite?

The first thing you might wonder about is why we decided to reimplement the framework from scratch as opposed to use the existing code base and enhance it further on. We decided so because we had learned a lot of lessons from oAW Xtext. Although we wanted to stick with many proven concepts we found the implementation was lacking a solid foundation (the author of these lines is the original author of that non-solid code btw. :-)). The first version of oAW Xtext was basically a proof of concept which was so well received that it had been extended with all kinds of features (some were good, some were bad). Unfortunately code quality, clean and orthogonal concepts and test coverage did not receive the necessary focus.

In addition to this aspects of quality, oAW Xtext suffers from some severe performance problems. The extensive and naive use of Xtend (see next section) prevented many users to use oAW Xtext for growing real-world models.

2. Migration overview

Although a couple of things have changed we tried to keep good ideas and left many things unchanged. At the same time we wanted to clean up poor concepts and solve the main problems we and you had with oAW Xtext. From a bird's eye view if you want to migrate an existing oAW Xtext project to TMF Xtext, you mainly just need to rename the old grammar from *.xtxt to *.xtext and add two lines to the beginning of that document (see below for details). You might also have to change a few keywords, but all in all this is pretty easy and we've migrated a couple of oAW Xtext projects this way without problems. The other aspect where lots of code might have been written for is validation. In oAW Xtext we used Xpand's Check language to define constraints on the meta model. Even though this has been one major reason for the lack of scalability in Xtext we decided to keep the Check language as an option for compatibility reasons (see 'Differences in Validation'). Therefore, you do not need to translate your existing checks to a different language. Even better, you can overcome some performance issues by leveraging the newly introduced hooks to control the time of validation (while you type, on save, or on triggering an explicit action). Anyway, if you want to provide a slick user experience validation should run fast while you type. Therefore, we strongly encourage you to implement validation using our declarative Java approach (TODO: ref).

We've developed and reviewed a lot of oAW Xtext projects and saw that most of the work was done in the grammar and in the validation view point. Other aspects such as outline view, label provider or content assist have been customized too, but they usually do not contain complicated Xtend code. In some projects the exception was linking and content assist which in oAW Xtext usually forces one to write a lot of duplicated code. While working on this we came up with a new concept called "scopes" that not only streamlines implementation in terms of redundancy. Scopes also increase the overall performance of Xtext. But since the concept of scopes was not carved out in oAW Xtext one usually implemented a cluttered and duplicated poor copy through linking and content assist. For obvious reasons, we didn't manage to come up with a good compatibility layer. So this is where most of the migration effort will go into if implemented customized linking. But we think the notion of scopes is such a valuable addition that it is worth the refactoring. Also, when looking at existing oAW Xtext projects we found that most projects either didn't change the default linking that much or they came up with their own linking framework anyway.

However, if we have completely misunderstood the situation and your oAW Xtext project cannot be migrated in a reasonable amount of time, please tell us. We want to help you!

3. Where are the Xtend-based APIs?

One of the nice things with oAW Xtext was the use of Xtend to allow customizing different aspects of the generated language infrastructure. Xtend is a part of the template language Xpand, which is shipped with oAW (and now is included in M2T Xpand). It provides a nicer expression syntax than Java. Especially the existence of higher-order functions for collections is extremely handy when working with models. In addition to the nice syntax, it provides dynamic polymorphic dispatch, which means that declaring e.g. label computation for a meta model is very convenient and type safe at the same time. In Java one usually has to write instanceof and cast orgies.

3.1. Xtend is hard to debug

While the aforementioned features allow the convenient specification of label and icon providers, outline views, content assist and linking, Xtend is interpreted and therefore hard to debug. Because of that Xpand is shipped with a special debugger facility. Unfortunately, this debugger cannot be used in the context of Xtext since it implies that the Xtend functions have to be called from a workflow. This is not and cannot be the case for Xtext Editors. As a result one has to debug his way through the interpreter, which is hard and inconvenient (even for us, who have written that interpreter).

3.2. Xtend is slow

But the problematic debugging in the context of Xtext was not the main reason why there are no Xtend-based APIs beside Check in TMF Xtext. The main reason is that Xtend is too slow to be evaluated “inside” the editor again and again while you type. While Xtend’s performance is sufficient when run in a code generator, it is just too slow to be executed on keystroke (or 500ms after the last keystroke, which is when the reconciler reparses, links and validates the model). Xtend is relatively slow, because it supports polymorphic dispatch (the cool feature mentioned above), which means that for each invocation it matches at runtime which function is the best match and it has to do so on each call. Also Xtend supports a pluggable typesystem, where you can adapt to other existing type systems such as JavaBeans or Ecore. This is very powerful and flexible but introduces another indirection layer. Last but not least the code is interpreted and not compiled. The price we pay for all these nice features is reduced performance.

In addition to these scalability problems we have designed some core APIs (e.g. scopes) around the idea of Iterables, which allows for lazy resolution of elements. As Xtend does not know the concept of Iterators you would have to work with lists all the time. Copying collections over and over again is far more expensive than chaining Iterables through filters and transformers like we do with Google Collections in TMF Xtext.

3.3. Convenient Java

To summarize the dilemma we had to find a way to allow for convenient, scalable and debuggable APIs. Ultimately we wanted to provide neat DSLs for every view point, which provide all these things. However, we had to prioritize our efforts with the available resources in mind. As a result we found ways and means to tweak Java as good as possible to allow for relatively convenient, high performing implementations.

Java is fast and can easily be debugged but ranks behind Xtend regarding convenience. We address this with different approaches to make Java development in the context of Xtext as comfortable as possible.

Most of the APIs in TMF Xtext use polymorphic dispatching, which mimics the behavior known from Xtend. Another valuable feature of Xtend while working with oAW Xtext is static type checking while working with the inferred Ecore model whereas in Java the work with dynamic Ecore classes was rather cumbersome. Since TMF Xtext generates static Ecore classes per default you get static typing in Java as well. Additionally, the use of Google Collections reduces the pain when navigating over your model to extract information.

With these techniques an ILabelProvider that handles your own EClasses `Property` and `Entity` can be written like this:

```
public class DomainModelLabelProvider extends DefaultLabelProvider {

  String label(Entity e) {
```

```
    return e.getName();
  }

  String image(Property p) {
    return p.isMultiValue() ? "complexProperty.gif": "simpleProperty.gif";
  }

  String image(Entity e) {
    return "entity.gif";
  }

}
```

As you can see this is very similar to the way one describes labels and icons in oAW Xtext, but has the advantage that it is easier to test and to debug, faster and can be used everywhere an ILabelProvider is expected in Eclipse.

3.4. Conclusion

Just to get it right, Xtend is a very powerful language and we still use it for its main purpose: code generation and model transformation. The whole generator in TMF Xtext is written in Xpand and Xtend and its performance is at least in our experience sufficient for that use case. Actually we were able to increase the runtime performance of Xpand by about 60% for the Galileo release of M2T Xpand. But still, live execution in the IDE and on typing is very critical and one has to think about every millisecond in this area.

As an alternative to the Java APIs we also considered other JVM languages. We like static typing and think it is especially important when processing typed models (which evolve heavily). That's why Groovy or JRuby were no alternatives. Using Scala would have been a very good match, but we didn't want to require knowledge of Scala so we didn't use it and stuck to Java. Also without changing the EMF generator it is inconvenient to use EMF classes in Scala, since Scala uses a different convention for getter and setter methods.

4. Differences

In this section differences between oAW Xtext and TMF Xtext are outlined and explained. We'll start from the APIs such as the grammar language and the validation and finish with the different hooks for customizing linking and several UI aspects, such as outline view and content assist. We'll also try to map some of the oAW Xtext concepts to their counterparts in TMF Xtext.

4.1. Differences in the grammar language

When looking at a TMF Xtext grammar the first time it looks like one has to provide additional information which was not necessary in oAW Xtext. In oAW Xtext *.xtxt files started with the first production rule where in TMF Xtext one has to declare the name of the language followed by declaration of one or more used/generated meta models:

TMF Xtext heading information

```
grammar my.namespace.Language with org.eclipse.xtext.common.Terminals
generate myDsl "http://www.namespace.my/2009/MyDSL"

FirstRule : ...
```

In oAW Xtext this information was provided through the generator (actually it is contained in the *.properties file) but we found that these things are very important for a complete description of a grammar. Therefore we made that information becoming a part of the grammar language in order to have self-describing grammars and allow for sophisticated static analysis, etc..

Apart from the first two lines the grammar languages of both versions are more or less compatible. The syntax for all the different EBNF concepts (alternatives, groups, cardinalities) is similar. Also assignments are syntactically and semantically identical in both versions. However in TMF Xtext some concepts have been generalized and improved:

4.1.1. String rules become Datatype rules

The very handy String rules are still present in TMF Xtext but we generalized them so that you don't need to write the 'String' keyword in front of them and at the same time these rules can not only produce EStrings but (as the name suggests) any kind of EDatatype. Every parser rule that does neither include assignments nor calls any that does returns an EDataType containing the consumed data. Per default this is an EString but you can now simply create a parser rule returning other EDatatype as well (see TODO-REF).

```
Float returns ecore::EDouble : INT ('.' INT)?;
```

4.1.2. Enum rules

Enum rules have not changed significantly. The keyword has changed to be all lower case ('enum' instead of 'Enum'). Also the right-hand side of the assignment is now optional. That is in oAW Xtext:

```
Enum MyEnum : foo='foo' | bar='bar';
```

becomes

```
enum MyEnum : foo='foo' | bar='bar';
```

and because the name of the literal equals the literal value one can omit the right-hand side in this case and write:

```
enum MyEnum : foo | bar;
```

4.1.3. Native rules

Another improvement is that we could replace the blackbox native rules with full-blown EBNF syntax. That is native rules become terminal rules and are no longer written as a string literal containing ANTLR syntax.

Example :

```
Native FOO : "'f' 'o' 'o'";
```

becomes

```
terminal FOO : 'f' 'o' 'o';
```

See the reference documentation for all the different expressions possible in terminal rules (TODO-REF).

4.1.4. No URI terminal rule anymore

We decided to remove the URI terminal. The only reason for the existence was to mark the model somehow so that the framework knows what information to use in order to load referenced models. Instead we decided to solve this similar to how we imply other defaults: by convention.

So instead of using a special token which is syntactically a STRING token, the default import mechanism now looks for EAttributes of type EString with the name 'importedURI'. That is if you've used the URI token like this:

```
Import : 'import' myReference=URI;
```

you'll have to rewrite it that way

```
Import : 'import' importedURI=STRING;
```

Although this changes your meta model, one usually never used this reference explicitly as it was only there to be used by the default import mechanism. So we assume and hope that changing the reference is not a big deal for you.

4.1.5. Return types

The syntax to explicitly declare the return type of a rule has changed. In oAW Xtext (where this was marked as 'experimental') the syntax was:

```
MyRule [MyType] : foo=ID;
```

in TMF Xtext we have a keyword for this :

```
MyRule returns MyType : foo=ID;
```

This is a bit more verbose, but at the same time more readable. And as you don't have to write the return type in most situations, it's good to have a more explicit, readable syntax.

4.2. Differences in Validation

TMF Xtext still supports implementing validation using Xpand's Check. However it is no longer using the EMF meta model (typesystem) but now uses the so called JavaBeansMetamodel. This means that the types in Check and Xtend correspond to Java types. This requires minor changes (namely the namespaces to be imported are now the qualified name of the generated Java classes). Example: If your Check file looked like this in oAW Xtext :

```
import myMetamodel;
context Type ERROR "foo" : name!=null;
```

it becomes something like the following in TMF Xtext :

```
import my::pack::to::myMetaModel;
context Type ERROR "foo" : name!=null;
```

Where `my::pack::to::myMetaModel` refers to the package the generated EClasses are in. In other words there's a Java class `my.pack.to.myMetaModel.Type`. We changed this in order to allow use of any Java API from within Check and Xtend.

4.3. Differences in Linking

The linking has been completely redesigned. In oAW Xtext linking was done in a very naive way: To find an element one queries a list of all 'visible' EObjects, then filters out what is not needed and tries to find a match by comparing the text written for the crosslink with the value returned by the `id()` extension. As a side-effect of `link_feature()` the reference is set.

The code about selecting and filtering `allElements()` usually has been duplicated in the corresponding content assist function, so that linking and content assist are semantically in sync. If you're good (we usually were not) you externalized that piece of code and reused the same extension in content assist and linking.

To put it bluntly this approach could be summarized in two steps:

1. Give me the whole universe including every unregarded object in the uncharted backwaters of the unfashionable end of the western spiral arm of the galaxy and squeeze it into an Arraylist
2. From this, select the one I need

This was not only very expensive but also lacks an important abstraction: the notion of scopes.

4.3.1. The idea of scopes

In TMF Xtext we've introduced scopes and scope providers that are responsible for creating scopes. A scope is basically a set of name->value pairs. Scopes are implemented upon Iterables and are nested to build a hierarchy. With scopes we declare "visible" objects in a lazy and cost-saving way where the linker only navigates as far as necessary to find matching objects. The content assist reuses this set of visible objects to offer only reachable objects.

When the linking has to be customized scoping is where most of the semantics typically goes into. By implementing an IScopeProvider for your language linking and content assist will automatically be kept in sync since both use the scope provider.

The provided default implementation is semantically mostly identical to how the default linking worked in oAW Xtext:

1. Elements which have an attribute 'name' will be made visible by their name
2. Referenced resources will be put on the (outer) scope by using the 'importedURI'- naming convention (TODO-REF) and will only be loaded if necessary

3. The available elements are filtered by the expected type (i.e. the type of the reference to be linked)

4.3.2. Migration

We expect the migration of linking to be very simple if you've not changed the default semantics that much. We've already migrated a couple of projects and it wasn't too hard to do so. If you have changed linking (and also content assist) a lot, you'll have to translate the semantics to the IScopeProvider concept. This might be a bit of work, but it is worth the effort as this will clean up your code base and better separate concerns.

4.4. Differences in UI customizing

In oAW Xtext several UI services such as content assist, outline view or the label provider have been customized using Xtend. In TMF Xtext there is no Xtend API for these aspects. Extensive model computations for the content assist is most probably not necessary anymore- it reuses scopes. And since we provide a declarative Java API that mimics the polymorphic dispatch and relies on static Ecore classes you will gain nearly the same expressiveness as before while increasing maintainability and performance.

Beside the API change in favor of Java we have to mention that in TMF Xtext the outline view does not support multiple view points so far. This is just because we didn't manage to get this included. We don't think that view points are a bad idea in general, but we decided that other things were more important.

5. New Features

This section provides an overview of new possibilities with TMF Xtext compared to oAW Xtext. Please note that this list is neither complete nor does it explain every aspect in detail to keep this document tight.

5.1. Dependency Injection with Google Guice

Beyond the mentioned architectural overhaul that carve out separate concerns in a meaningful way these different classes of TMF Xtext are wired using Google Guice and can easily replaced by or combined with your own implementation. We could have foreseen some common needs for adaption but with this mechanism you can virtually change every aspect of Xtext without duplicating an unmanageable amount of code.

5.2. Improvements on Grammar Level

The Xtext grammar language introduces some new features, too. Read the chapter 'grammar language' to understand the details about all the improvements that have been implemented.

Grammar mixins allow you to extend existing languages and change their concrete and abstract syntax. However the abstract syntax (i.e. the Ecore model) can only be extended. This allows you to reuse existing validations, code generators, interpreters or other code which has been written against those types.

In oAW Xtext common terminals like ID, INT, STRING, ML_COMMENT, SL_SOMMENT and WS (whitespace) were hard coded into the grammar language and couldn't be removed and hardly overridden. In TMF Xtext these terminals are imported through the newly introduced grammar mixin mechanism from the shipped grammar `org.eclipse.xtext.common.Terminals` per default. This means that they are still there but reside in a libraries now. You don't have to use them and you can come up with your own set of reusable rules.

Reusing existing Ecore models in oAW Xtext didn't work well and we communicated this by flagging this feature as 'experimental'. In TMF Xtext importing existing Ecore models is fully supported. Moreover, it is possible to import a couple of different EPackages and generate some at the same time, so that the generated Ecore models extend or refer to the existing Ecore models.

5.3. Fine-grained control for validation

In order to make more expensive validations possible without slowing down the editor, TMF Xtext supports three different validation hooks.

1. FAST constraints are triggered by the reconciler (i.e. 500 ms after the last keystroke) and on save.
2. NORMAL constraints are executed on save only.
3. EXPENSIVE constraints are executed through an action which is available through the context menu.

Please note that when using Xtext models for code generation the checks of all three categories will be performed.

Beside this it is now possible to add information about the feature which is validated.

```
context Entity#name ERROR "Name should start with a capital "+this.name+"." :
  this.name.toFirstUpper() == this.name;
```

If you add the name of a feature prepended by a hash (‘#’) in Check, the editor will only mark the value of the feature (name), not the whole object (Entity). Both concepts, control over validation time as well as pointing to a specific feature, complement one another in Check and Java based validation.

6. Migration Support

In this document we tried to explain why we decided to change some aspects of Xtext’s architecture. We consider most changes as minor but when it comes to scopes you will face a conceptual enhancement that did not exist in oAW Xtext. We tried to explain why it is not easily possible to come up with an adapter for scoping.

That said you might not have the time to do the migration and wished to have advice for migrating, especially from oAW linking to TMF linking. You’re welcome to ask any questions in the newsgroup and we’ll try to help you as much as possible in order to get your projects migrated. Also, if you don’t want to do the migration yourself we (itemis AG) can do the work for you or help you with that.