

Semantic Annotations For Xtext Languages

Markus Voelter, voelter@acm.org

Version 0.5

Overview

Many languages defined with Xtext have aspects in common. One way of exploiting this is to modularize language definitions and then composing specific languages from these modules. However, because of shortcomings of the current (4.3.1) version of Xtext, this is not necessarily a very viable approach.

This semantic annotation toolkit provides support for another approach. Languages are generated from scratch (i.e. there is no reuse between grammar fragments), but the semantics of various language building blocks are still reusable. By annotating grammar elements with semantic annotations, the necessary Xtext infrastructure can be generated to make those grammar elements behave in a given way.

Technically, this is implemented by generating extensions and checks as well as by model transformations and extensions of the meta- model.

This paper describes how to use the toolkit.

Note that this toolkit will of course not be able to address 100% of all cases. But it does address the 80% case, and the rest can be implemented manually, as usual.

General Approach

To use the semantic annotation toolkit, you basically use Xtext in the way you've used it before. Everything you've learned about Xtext is still valid. However a (hopefully growing in the future) number of repeated customizations is automated using this toolkit.

The semantic annotation toolkit comes with a textual DSL that is used to define certain semantics, constraints, and behaviors for languages defined via Xtext. The following example shows how to use the toolkit.

Defining an Xtext Language

To illustrate the general approach we use the well proven state machine example (take a look at the *org.openarchitectureware.seman.testSM* example from the repository at the Google Code project: <http://code.google.com/p/metamodelsemantics/>). In this project we have a grammar definition that looks as follows¹.

```
World:
  (statemachines+=Statemachine)*;

Statemachine:
  "statemachine" name=ID ("initial" initial=[State])? "{"
    ( states+=State |
      events +=Event )*
  ";

State:
  "state" name=ID "{"
    (transitions+=Transition)*
  ";

Event:
  "event" name=ID;

Transition:
  "transition" event=[Event] "->" target=[State];
```

To define the semantics, you should then create a *.seman* file right next to the grammar definition. Make sure that this file has the same name as the *.txt* file next to it (of course with the different extension).

Please add the following code to the file:

```
semantic annotation for
"platform:/resource/org.openarchitectureware.seman.testSM/src-
gen/org/openarchitectureware/seman/testSM/testsm.ecore" {
}
}
```

In the subsequent sections, we will add various semantic annotations to this file. The editor for the semantic annotations file is aware of the meta model of the annotated language and provides code completion and constraint checks.

To make sure the semantic annotations are processed, you also have to modify the workflow that generates the meta model and editor from the grammar. Here is how it should look like:

```
<workflow>
  <!-- OLD STUFF -->
  <property file='generate.properties' />
  <component file='org/openarchitectureware/xttext/Generator.oaw'
    inheritAll='true' />

  <!-- NEW STUFF -->
  <component file="org/openarchitectureware/seman/seman.oaw"
    inheritAll="true" />
</workflow>
```

As you can see, we call another workflow which processors the semantic annotations.

¹ I assume that you understand this code. If not, please go back and learn Xtext.

If you would now run the workflow, the additional generator will add checks to the *GenChecks.chk* file as well as extensions to the *GenExtensions.ext* files. It also generates a file called *GenSemanProcessor.ext*, but it is currently not used.

Specific Annotations

Unique Names

In many cases the elements in a collection must have unique names. To enforce this, you have to write a constraint. Alternatively you can use the following syntax in the semantic annotations file.

```
class Statemachine {
  uniqueNames events
  uniqueNames states
}
```

Qualified Names

In many cases, the structure of qualified names must be defined. The semantic annotations provide support here:

```
class Transition {
  parents
  namespace Statemachine State "transition" delimiter "."
}
```

You can specify a list of parent classes (here: *Statemachine*, *State*), literals (here: *transition*) and a delimiter. The generator builds a *namespace()* function as well as a *qname()* function. For each of the parent classes, it uses the *shortName()* to build the qualified name. The *shortName()* is the name of an element, or *<unnamed>* in case an element has no name.

Naming of Derived Resources

In almost all code generators based on DSLs you need to calculate the names of derived resources. You probably know extensions like the following:

```
baseClassFileName(Component this):
  fqBaseClassName().replaceAll("\\.", "/")+".java";
packageName( Component this ): system().packageName()+ "."+name.toFirstLower();
fqBaseClassName( Component this ): packageName()+ "."+baseClassName();
baseClassName( Component this ): name+"Base";
implClassName( Component this ): name;
fqImplClassName( Component this ): packageName()+ "."+implClassName();
```

These are extremely repetitive, but hard to modularize. The semantic annotations tool provides a solution (yes, ok, that's not semantics, but still quite useful ☺)

Here is an example specification:

```
semantic annotation for
"platform:resource/org.openarchitectureware.seman.test/src-
gen/org/openarchitectureware/seman/test/stdsl.ecore" {
```

```

derivedResourceCategory javaclass delimiter "." extension "java"

class Entity {
    derivedResource baseClass category javaclass suffix "Base"
        parts System "entities"
    derivedResource implClass category javaclass partsAsIn baseClass
}

```

You start out by defining a derived resource category that determines the delimited between parts of namespaces and the file extension for the derived resource.

You can then define derived resources for classes. Each derived resource has to have a name (*baseClass*) and a reference to a category (*javaclass*) that determines the delimiter and file extension. You can then define a suffix and a prefix for the derived resource name. And you define the structure of the qualified name, just as in the namespace definition. Finally, using the *partsAsIn* construct, you can make the nesting structure (but not prefix and suffix) similar to another derived resource.

To illustrate the exact meaning of the above example, I include here the generated extensions. This should make clear what is going on.

```

String baseClassName( Entity this ):
    name+"Base";
String baseClassNamespace( Entity this ):
    parentSystem().shortName()+"."+"entities";
String baseClassFqName( Entity this ):
    baseClassNamespace()+"."+baseClassName();
String baseClassFileName( Entity this ):
    parentSystem().shortName()+"/"+"entities"+"/"+baseClassName()+".java";

String implClassName( Entity this ):
    name;
String implClassNamespace( Entity this ):
    parentSystem().shortName()+"."+"entities";
String implClassFqName( Entity this ):
    implClassNamespace()+"."+implClassName();
String implClassFileName( Entity this ):
    parentSystem().shortName()+"/"+"entities"+"/"+implClassName()+".java";

```

Parents

Often, you will need to navigate to various levels of parents above an element. This involves a number of *eContainer* calls and downcasts. By using the following syntax you can automatically generate the respective extensions.

```

class Transition {
    parents
}

```

In case of our state machine example, the following extensions are generated.

```

State parentState( Transition this ):
    goUpTo( State );

Statemachine parentStatemachine( Transition this ):
    goUpTo( Statemachine );

World parentWorld( Transition this ):
    goUpTo( World );

```

Scoping

References always need to be scoped, since by default, all elements of the reference type in the model are shown, which is typically not the expected behavior. The semantic annotations toolkit contains a number of scoping schemes (which will be extended in the future). Here is the syntax.

```
class Transition {
  scope event allUnderParent Statemachine
  scope target allUnderParent Statemachine
}
```

The scope definition starts with the keyword *scope* and then the reference from the target class (here: *event* and *target*). We then specify the scoping scheme. *allUnderParent* means that starting at the current element (in this case a *Transition*) we climb up the containment hierarchy until we find an instance of the specified type (here: *Statemachine*) and then include all elements of the target type in the scope (in this case it would be all events or states below the containing state machine).

Here are examples of different scoping schemes.

```
class Statemachine {
  scope initial allInSiblingProperty states
  scope baseMachine allUnderParent World excludeSelf
}
```

The scope for the *initial* reference contains all elements in another property of the same element. In this case we propose all the *states* of the state machine.

The scope for *baseMachine* contains all elements under the *World* parent (i.e. all state machines in the model) but excludes the current state machine. *excludeSelf* can be added to most scoping schemes.

Take a look at the following example model. You can define nested scopes, recursively, and then declare types and variables. Variables reference the type.

```
scope s1 {
  type T1;
  var t1: T1;
  scope s2 {
    type T2;
    var t2: T1;
  }
  scope s3 {
    type T3;
    var t3: T3;
    scope s4 {
      type T4;
      var t4: T4;
    }
  }
}
```

The scope for this *type* reference should be defined as: all the types in the *types* collection of the scope in which the variable is declared, as well as all the types in the parent scopes, recursively. Here is what you need to specify:

```
class VarDecl {
  scope type allInParentProperty recursive Scope.types
}
```

The *recursive* keyword is optional. If not used, the scheme goes up to the first parent of the type specified (here: *Scope*) and the returns the contents of the specified property (here: *types*). If the *recursive* keyword is present, then the scheme recursively climbs the containment tree and looks for more instances of *Scope*, and also returns the contents of their *types* collection.

It is also possible to support imports. Consider the following extended example, note the *import* statement in *s5*:

```
scope s1 {
  type T1;
  var t1: T1;
  scope s2 {
    type T2;
    var t2: T1;
  }
  scope s3 {
    type T3;
    var t3: T3;
    scope s4 {
      type T4;
      var t4: T4;
    }
    scope s5 {
      import s4
      var x: T4;
    }
  }
}
```

Obviously, this makes the *s4* scope visible in *s5*, although it is not a parent scope. This is a classical scope import. You mark it up using the *extendAlong* keyword, specifying the name of the property of the parent class (here: *Scope*) that contains the imports.

```
class VarDecl {
  scope type allInParentProperty recursive Scope.types extendAlong imports
}
```

The type in the property (*imports*) can be arbitrary, but it has to have exactly one (not many!) reference (not containment!) to the same type (here: *Scope*).

There's also a scoping scheme that finds all instances of a type, even in imported files:

```
class VarDecl {
  scope type global
}
```

You can also use the *excludeSelf* modifier to make sure the current element is not found.

Finally, if none of the available scoping schemes work for you, you can specify a *custom* scope:

```
class Transition {
  scope event custom
}
```

To manually implement this scope you have to override the following extension by re-implementing it in the *Extensions.ext* file. It has to return a collection of model elements.

```
scope_Transition_event( Transition this ): // whatever
```

Link Order

The order in which the various references of a class are linked is non-deterministic in Xtext. However, sometimes it is important to link in a specific order. For example, when the scope of one reference A depends on the value of another reference B, it is important to first link B and then A. It is possible to specify this ordering constraint:

```
class Transition {
  linkorder event target
}
```

Note that currently it works only for single-valued references. Also, in the example above where the scope of reference A depends on reference B, you will have to use the *custom* scoping scheme for the scope of B.

Helper functions

Sometimes it is useful to have additional functions, a similar function for each meta class, for example, an *eval()* function in a expression language. You can specify this in the semantic annotations and have the function generated with an implementation that throws a *RuntimeException*, so you cannot forget to override the function and provide a sensible implementation.

```
semantic annotation for "platform:/resource/org...est/stdsl.ecore" {
  helperfunction myhelper::eval( Object ctx ) includeSubtypes
}
```

What is generated is a file *GenMyhelper.ext* with a function *eval(<Metaclass> this, Object ctx)*. You are expected to implement a file *Myhelper.ext* that includes and reexports *GenMyhelper.ext*, overriding the functions with a sensible implementation.

If you specify *includeSubtypes*, there's a separate function for each subtype of an abstract class (abstract rule). If not, you get a function only for the abstract class.

Templates

Consider the following example. *Configurations* contain two parameters *p1* and *p2*. The configuration can also reference another configuration through its *templateRef* reference, denoted by that *template* keyword. The expected behavior is that if you ask for example *C2* for the value of *p1* then, since it is not defined in *C2*, it delegates to its template and returns, in this case, *C1P1*. If a config specifies the value (like *C3*) then the value defined by itself is returned.

```
system S1 {
  config C1 {
    p1 C1P1;
    p2 thing C1Thing;
  }
  config C2 template C1 {
```

```

}

config C3 template C1 {
    p1 C3P1;
    p2 thing C3Thing;
}
}

```

Here is the piece of semantic annotation you have to write to get this behavior:

```

class Configuration {
    template templateRef {
        properties p1 p3;
        mode accessorFunctions;
    }
}

```

The *template* construct references a reference in the target class (here: *templateRef* in the class *Configuration*). It also lists a number of properties (here: *p1* and *p3*) for which the if-null-then-delegate behavior should apply.

From this specification, the semantic annotation processor generates extensions that have the same name as the properties (here: *p1*(*Configuraton this*) and *p3*(*Configuration this*)) that can be called by for example code generator authors to access the properties with this behavior.

The fact that accessor functions are generated is governed by the *mode accessorFunctions*; declaration. Currently, this is the only possible way. In a later implementation, a model transformation would be an alternative solution.

The default behavior, as shown above, is that the values in the template replace the values in the element you query. This means:

Single-Valued Attribute	Null	return template's value
Single-Valued Attribute	Not null	return target's value
Multi-Valued Attribute	Empty	return template's value
Multi-Valued Attribute	Not Empty	return target's value

It is also possible to use the *join* keyword, like so:

```

class Configuration {
    template templateRef {
        properties p1 join(p3);
        mode accessorFunctions;
    }
}

```

In this case, the semantics of what's returned are different:

Single-Valued Attribute	Null	return a collection with only the template's value in it
Single-Valued Attribute	Not null	return a collection with target's value and template's value
Multi-Valued Attribute	Empty or Not	return union of values from target and template

Further Work

The semantic annotation toolkit will continue to be developed over time. If you want to help or if you have suggestions please contact Markus Voelter at voelter@acm.org.