This article is known to apply to the following Eclipse projects:

- Eclipse Platform, release 3.3
- Eclipse Modeling Framework, release 2.3

Help us keep this information up-to-date: let us know if this information applies to other projects or releases.

To comment on this article, ask questions, or propose corrections, please see bug 234003.

Automating the embedding of Domain Specific Languages in Eclipse JDT

Summary

The Eclipse Java Development Tools (JDT) excels at supporting the editing and navigation of Java code, setting the bar for newer IDEs, including those for Domain Specific Languages (DSLs). Although IDE generation keeps making progress, entry barriers remain high, thus forcing many developers to rely on traditional ways to encapsulate new language abstractions: frameworks and XML dialects. We explore an alternative path, *Internal DSLs*, by automating the generation of the required APIs from Ecore models describing the abstract syntax of the DSLs in question. To evaluate the approach, we present a case study (statecharts) and discuss the pros and cons with respect to other approaches.

Most embedded DSLs, while offering a user-friendly syntax, are fragile in the sense that their expressions may not comply with the full static semantics of the DSL in question. Productivity studies recommend that errors should be reported while the frame of mind is still focused in the error location. To address this issue, we leverage the extension capability of Eclipse to detect at compile-time malformed DSLs expressions. The technique relies on mainstream components only: EMF, OCL, and JDT. We conclude by previewing ongoing work aimed at improving the support for embedded DSLs by performing language processing as a background task. The prototype described in this article (`DSL2JDT`) has been contributed to EMFT and is available from CVS as described in the Source Code section below.

By Miguel Garcia,
TUHH (Technische Universität Hamburg-Harburg, Germany)

May 25th, 2008

Table of Contents

1. Introduction
   - The internal DSL approach
   - Instructions for the impatient: how to use `DSL2JDT` in 10 seconds
   - Examples of existing internal DSLs
2. Statecharts example
3. Checking DSL well-formedness during editing
4. Sidenote: from EBNF grammar to Ecore model and back again
5. Processing DSL statements beyond well-formedness checking
   - Setting the stage: useful APIs for the task at hand
   - In-place translation
   - Statement-level annotations
   - DSL-specific views
6. Implementation of in-place translation
7. Related Work
   - DSL Embedding in Scala and Ruby
   - Static analysis of XML artifacts
   - Inspection and manipulation of Java ASTs
   - Competing approach: IDE generation
8. Conclusion
9. Acknowledgements
10. Source Code

Introduction

Nowadays, the development of software systems usually involves more than one language: SQL, BPEL, and JSP are popular examples, but the list can also be extended to include notations focused on certain aspects of system functionality (business rules, access control, databinding between GUI forms and underlying model objects, etc.)

Providing *integrated* IDEs for (combinations of) such Domain-Specific Languages (DSLs) has proven hard. A Java IDE aware of SQL would for example flag those embedded SQL statements that become invalid after refactoring the database schema. Supporting such scenarios is easier if both host and embedded languages are designed with cooperation in mind, as is the case with Microsoft's LINQ (Language INtegrated Query). Experience has also shown that any complex-enough DSL is doomed to re-invent constructs that are taken for granted in general-purpose languages (think of control-flow constructs in Oracle PL/SQL, in XSL, and in QVT-Operational), thus strengthening the case for integrated tool support.

The conventional wisdom around DSL tooling is that one may either:

1. provide minimal compile-time checking of DSLs. This is the path followed by XML practice, with errors being discovered at runtime when document instances are interpreted,
   or
2. invest effort in developing dedicated plugins for editing DSLs with custom syntax (be it textual or diagram-based), checking at compile time the Abstract-Syntax-Trees (ASTs) for all involved software artifacts (thus covering the refactoring scenario mentioned above).

The economics of the two alternatives are clear: the "dedicated IDE" approach is technically better but also justifiable only for DSLs with a large user base. Actually, most of the tooling cost for a DSL comes from supporting its concrete syntax. Most of the benefits of a DSL however result from the analyses and transformations performed on its abstract syntax. Given that this "back-end infrastructure" is common to all DSL implementation alternatives we take it as starting point for our generator of internal DSL APIs. Besides allowing for early feedback on the DSL being engineered, the resulting risk minimization is useful in another way: if the DSL proves successful enough to warrant development of a dedicated IDE, no development effort is thrown away. With `DSL2JDT` the *Internal DSL* code can still be used in such IDE, as it depends only on the abstract syntax of the DSL, which is independent from its concrete syntax.

The internal DSL approach

One of the techniques covered by Martin Fowler in the online draft of his upcoming book on DSLs is *Internal DSLs*, which allow embedding DSL expressions in Java code. For example, the jMock testing framework allows writing code like:

```
mainframe.expects(once()).method("finish").after("start");
```

(reproduced from the paper Evolving an Embedded Domain-Specific Language in Java)

In effect, the Content Assist feature of the JDT and the type system of Java 5 are leveraged to enforce some of the well-formedness rules of the embedded DSL (jMock) when expressing ASTs for it in the host language (Java 5). Additionally, method chaining facilitates editing when used in conjunction with so called *progressive interfaces*: whenever the DSL grammar calls for a mandatory construct, the preceding method in the chain returns an interface with a single method declared in it (standing for the successor in lexical order in the underlying DSL grammar) so that the IDE offers a single choice. Using again the terminology described in more detail by Fowler, the resulting API is a Fluent Interface. Together with an Expression Builder they form the building blocks of an internal DSL API.

In terms of the familiar EMF Library example, the automatically generated Fluent Interface allows typing code snippets like the one depicted below. The example also shows that the technique is applicable to any Ecore model, although in the rest of this article we focus on language metamodels only.

Figure 1: Fluent Interface for a (non-DSL) Ecore model: the Library example

Fluent interfaces, by themselves, do not capture all relevant well-formedness rules (WFRs) of any but the simplest DSLs. For example, most imperative languages demand that: (a) "each variable usage must appear in scope of its single previous declaration", and (b) "duplicate names are to be avoided in the same namespace". As for modeling languages, two representative WFRs can be drawn from UML: (c) in class diagrams, cyclic inheritance is not allowed, and (d) in statecharts, a composite state consists of one or more regions, all of whose states must be uniquely named.

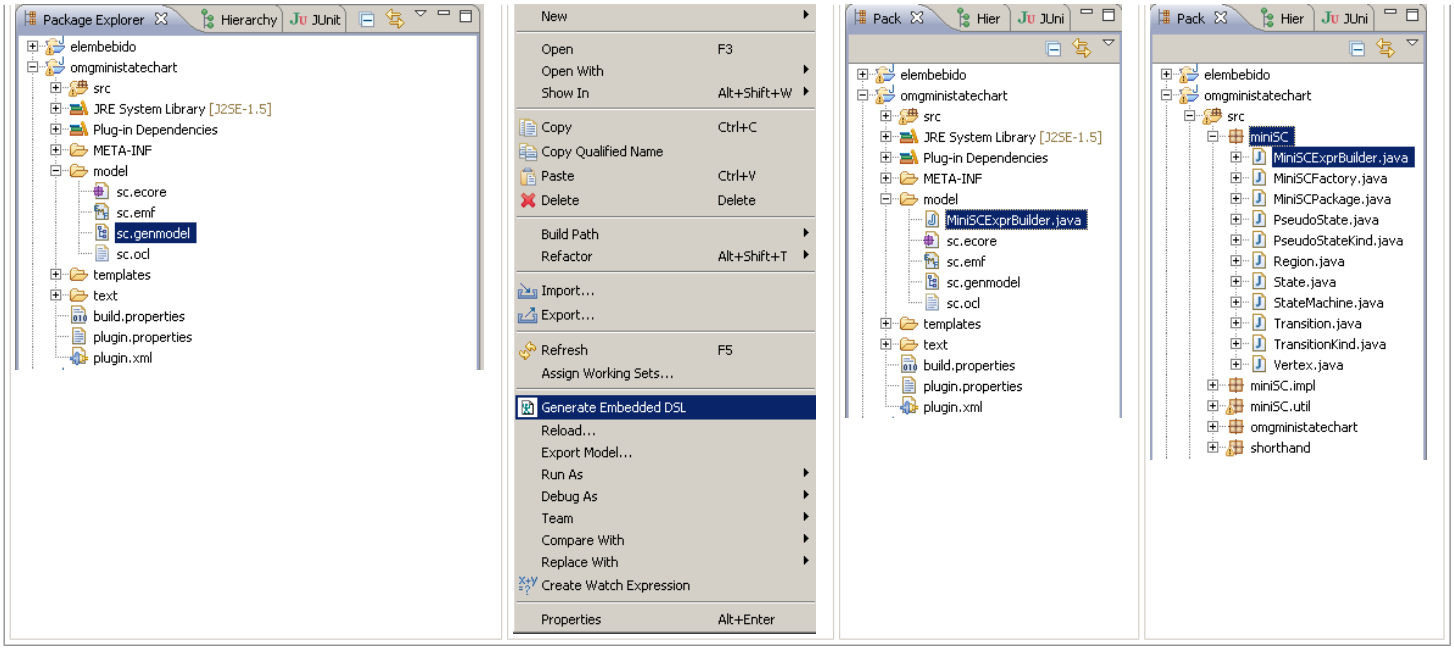
Our approach towards DSL embedding allows evaluating at compile-time such constraints, provided they can be discovered by the EMF Validation Framework using reflection. Christian W. Damus covers in the article Implementing Model Integrity in EMF with MDT OCL how to annotate an `.ecore` model with constraints. For simplicity, OCL may be left out initially and the validation methods completed manually. Examples are given later showcasing both alternatives for the statechart DSL.

The combination of Fluent Interface and build-time well-formedness checking surpasses the "DSL in XML" approach in terms of usability and safety, moreover relying on mainstream technologies: Eclipse Ecore, Eclipse OCL, and Eclipse JDT. Additional techniques (in-place translation, statement-level annotations, and DSL-specific views) may be optionally adopted to further increase the usability of embedded DSLs. We report about our progress so far around them in Sec. *Processing DSL statements*. But first, more examples of existing internal DSLs are given.

Instructions for the impatient: how to use `DSL2JDT` in 10 seconds

If you just can't wait to start using `DSL2JDT`, follow these steps:

1. checkout the two plugin projects that make up `DSL2JDT` from CVS as explained in the Source Code section
2. start a second Eclipse instance, launching it with the two plugins above enabled
3. create a plugin project, create your `.ecore` metamodel in it and generate its corresponding `.genmodel`.
4. Open the `.genmodel` file with its editor (you may want to set the `Base Package` property of the root package) and generate Model code (at the very least, more if you like).
5. right-click on `.genmodel`, choose "Generate Embedded DSL".
6. a text file named `<rootPackageName>ExprBuilder.java` is created in the same folder where the `.genmodel` is located. Move this Java file to the root Java package generated from the `.genmodel`.



tip If you followed steps 1-6 above, you'll have a project similar to what `omgministatechart.zip` delivers out of the box!

Examples of existing internal DSLs

As far as we know, the APIs of all existing internal DSLs have been developed manually. The code snippets in this subsection (from the Guice, Jequel, and KodKod projects) illustrate some frequent idioms. Basically, repetition of enclosing lexical contexts is avoided, thus reducing syntactic noise.

Listing 1: Guice, a lightweight dependency injection framework for Java 5 and above, http://code.google.com/p/google-guice/

```
public class MyModule implements Module {
   public void configure(Binder binder) {
     binder.bind(Service.class)
       .to(ServiceImpl.class)
       .in(Scopes.SINGLETON);
   }
 }
```

Listing 2: Jequel, Embedding SQL in Java, http://www.jequel.de

```
public class JEQUEL {

 interface ArticleBean {
       int getArticleNo();
       String getName();
   }

   public void testParameterExample() {
       final Sql sql = Select(ARTICLE.NAME, ARTICLE.ARTICLE_NO)
               .from(ARTICLE)
               .where(ARTICLE.OID.in(named("article_oid"))).toSql();

       final Collection articleDesc = sql.executeOn(dataSource)
               .withParams("article_oid", Arrays.asList(10, 11, 12))
               .mapBeans(new BeanRowMapper() {
                   public String mapBean(final ArticleBean bean) {
                       return bean.getArticleNo() + "/" + bean.getName();
                   }
               });
       assertEquals(1, articleDesc.size());
       assertEquals("12345/Foobar", articleDesc.iterator().next());
   }

}
```

Listing 3: Relational calculus expressions for the KodKod relational engine, http://web.mit.edu/emina/www/kodkod.html

```
public class KodKod {

 /**
   * Returns a formula stating that all vertices
   * have at least one color, and that no two adjacent
   * vertices have intersecting colors.
   * @return a formula stating that all vertices
   * have at least one color, and that no two adjacent
   * vertices have intersecting colors.
   */
 public Formula coloring() {
   final Variable n = Variable.unary("n");
   final Formula f0 = n.join(color).intersection(Color).some();
```

```
    final Formula f1 = n.join(color).intersection(n.join(graph).join(color)).no();
    return (f0.and(f1)).forAll(n.oneOf(Node));
  }

}
```

Statecharts example

Statecharts often serve as examples in discussions on model-driven tooling and this article follows that tradition. Being a graphical formalism, any usability points that their embedding can score should be welcomed with appreciation: a basic statechart metamodel (Figure 2) *devoid of any annotation for concrete syntax* is given as sole input to `DSL2JDT` (by right-clicking on the `.genmodel` file and choosing "Generate Embedded DSL"). The screen capture in Figure 3 shows the resulting Expression Builder API being used to instantiate the telephone statechart from Figure 4.

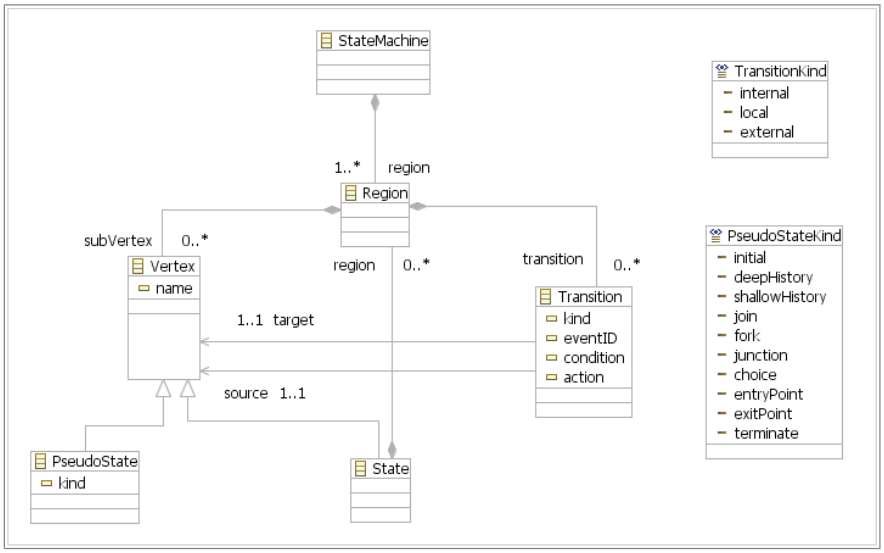
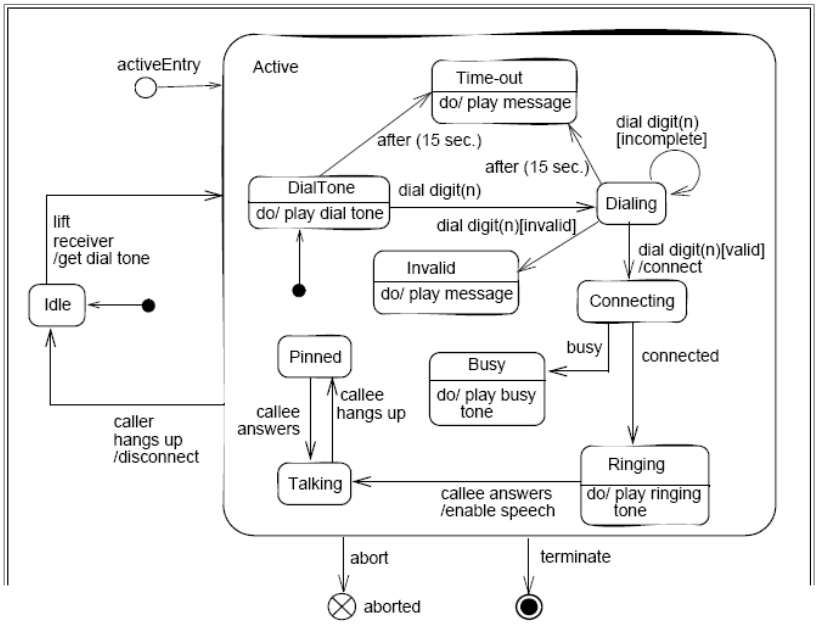


Figure 2: Metamodel for the Mini Statechart DSL

```
public class C {

    public static StateMachine telephoneExample() {
        State stateIdle = S.state().name("idle").region().toAST();
        PseudoState stateTopInitial = S.pseudoState().name("start").kindInitial().toAST();
        /*
         * here's where the sub-machine of state Active would be embedded.
         * That's a total of nine states, one pseudostate, and twelve
         * transitions. Given that the usage of the expression builder is clear
         * from the rest of this method, all that is elided.
         */
        Region regionActive = S.region().subVertex().transition().toAST();
        State stateActive = S.state().name("active").region(regionActive).toAST();
        PseudoState topTerminate = S.pseudoState().name("terminate").kindTerminate().toAST();
        Transition liftReceiver = S.transition().source(stateIdle).target(stateActive).eventID("liftReceiver").action("getDialTone")
                .toAST();
        Transition callerHangsUp = S.transition().source(s)
        Region topRegion = S.region().subVertex(stateIdle                                    topTerminate).transition().toAST();
        StateMachine telephone = S.stateMachine().region(                                    ;

        return telephone;
    }

    private static class S extends MiniSCExprBuilder {
    }

}
```

Vertex arg
stateActive
stateTopInitial
stateIdle

Figure 3: Embedded DSL statements for our Mini Statechart DSL



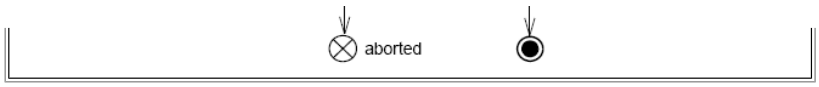


Figure 4: Statechart of a simple telephone
(reproduced as-is from OMG UML 2.1.1 Superstructure Specification)

tip When using a Fluent Interface, Content Assist suggestions contain by default the methods declared in `java.lang.Object`, which are distracting. They can be filtered away with the **Java** > **Type Filters** preference page (that will elide them also in the Open Type dialog, quick fix and organize imports, but will not affect the Package Explorer and Type Hierarchy views).

What does the expression builder API for the statechart DSL look like? Consider for example class `Region` containing zero o more `Vertex` and zero o more `Transition`. At edit time, Content Assist should offer first `subVertex(...)` as completion proposal (only). After accepting that suggestion, the next method in the chain should be `transition(...)` (only). And that's just two structural features. Well, the fragment of the expression builder defining such API is reproduced below:

```
public class MiniSCExprBuilder {

1    // start of the method chain for class Region
      public static RegionBeingBuilt0 region() {
              return new RegionBeingBuilt(miniSC.MiniSCFactory.eINSTANCE.createRegion());
      }

2    // steps of the method chain
      public interface RegionBeingBuilt0 {
              public RegionBeingBuilt1 subVertex(miniSC.Vertex... items);
      }
      public interface RegionBeingBuilt1 {
              public RegionBeingBuilt2 transition(miniSC.Transition... items);
      }
      public interface RegionBeingBuilt2 {
              public miniSC.Region toAST();
      }

3    // the class holding state between method invocations in a chain
      public static class RegionBeingBuilt implements
             RegionBeingBuilt0, RegionBeingBuilt1, RegionBeingBuilt2 {
              private final miniSC.Region myExpr;

              RegionBeingBuilt(miniSC.Region arg) {
                      this.myExpr = arg;
              }

              public RegionBeingBuilt1 subVertex(miniSC.Vertex... items) {
                      this.myExpr.getSubVertex().clear();
                      this.myExpr.getSubVertex().addAll(java.util.Arrays.asList(items));
                      return this;
              }

              public RegionBeingBuilt2 transition(miniSC.Transition... items) {
                      this.myExpr.getTransition().clear();
                      this.myExpr.getTransition().addAll(java.util.Arrays.asList(items));
                      return this;
              }

              public miniSC.Region toAST() {
                      return this.myExpr;
              }
      }

      // ...
```

As can be seen, three parts are generated for each concrete class: 1 a factory method that simply wallpapers over a factory invocation. The freshly instantiated `EObject` is not directly returned but wrapped first in a decorator (class 3 `RegionBeingBuilt` in this case) which selectively discloses update methods on the wrapped `EObject`. Such update methods are grouped into 2 batches (three in this case, from `RegionBeingBuilt0` to `RegionBeingBuilt2`).

The choices offered by a progressive interface are not as linear as the example above might suggest. `DSL2JDT` performs the following optimizations:

- optional fields (i.e., EMF structural features having `lowerBound` of 0) are packed together in a batch of options, and thus may be omitted.
- alternatives (e.g. due to an `EEnum`) are similarly offered in a single batch by Content Assist. Antoher idiom consists in having a single update method (taking an enum literal as argument) in the generated Expression Builder, by specifying a (GenModel or Ecore) annotation with source `Gymnast` and key-value pair `("terminal2method", "false")`
- for boolean fields so called *yes/no methods* can be specified, by means of a GenModel or Ecore annotation with source `Gymnast` and two key-value pairs: `("yes", "methodNameToSetTrue")` and `("no", "methodNameToSetFalse")`.
- owned classes having primitive fields only (`EInt`, `EString`, etc.) are instantiated with a single method invocation, its parameters being assigned to such fields.
- *progressive interfaces* can be disabled (on a class or package basis) so that all update methods are offered in a single batch by Content Assist. For disabling, an annotation with source `Gymnast` and key-value pair `("progressiveInterface ", "false")` should be specified (this was in fact used way back in Figure 1).

Besides relying on JDT Content Assist, another potential venue for speeding up typing of embedded DSL statements are fill-in-the-blanks templates, a capability that `DSL2JDT` as of now does not exploit (but feel free to extend our source code to generate them from the input `.genmodel`).

Checking DSL well-formedness during editing

As stated in the introduction, we want to engage the IDE in checking the static semantics of DSL expressions. Two ways are feasible, which we dub *The Pragmatic Way* and *The Grand Plan Way*. We cover the former in this section and leave the latter for Sec. *Processing DSL statements* (that section is much longer). In a nutshell, the infrastrucure required for the second alternative is overkill for well-formedness checking, however it enables other use cases (in-place translation, statement-level annotations, and DSL-specific views).

The pragmatic approach simply leverages existing JUnit support in JDT:

1. Each group of embedded DSL statements (making up a DSL expression) is encapsulated in a dedicated Java method that returns the self-contained AST (obtained with

`toAST())`

2. a JUnit test is created for each method above, invoking the default EMF validation on the AST root node. That way, the particular WFRs of all the nodes in the tree will be evaluated, without having to enumerate them explicitly (EMF determines all the applicable validators using reflection).
3. The following utility function encapsulates the invocation to EMF validation, from JUnit's `assertTrue()`. Although not shown here, debugging the unit tests with an exception breakpoint of `AssertionError` allows inspecting detailed diagnostic messages for each malformed AST node.

```
public class MyEcoreUtil {
     public static boolean isWellFormed(EObject root) {
        Diagnostician diagnostician = new Diagnostician();
        final Diagnostic diagnostic = diagnostician.validate(root);
        boolean res = diagnostic.getSeverity() == Diagnostic.OK;
        return res;
     }
     // ...
}
```

For example, the static semantics for the telephone example from Figure 4 can be checked with:

```
public class TestTelephone extends junit.framework.TestCase {

        public void testTelephoneExample() {
               StateMachine dslExpr = C.telephoneExample();
               assertTrue(MyEcoreUtil.isWellFormed(dslExpr));
        }

}
```

The particular WFRs to evaluate for each DSL construct can be given as Java or OCL. In both cases an annotation with source `http://www.eclipse.org/emf/2002/Ecore` should be made on the constrained class, listing the name of the constraint methods (as shown in Figure 6). If no OCL is specified, the generated validator method has to be completed manually as shown in Figure 5 for constraint `noDuplicates` in class `Region`.

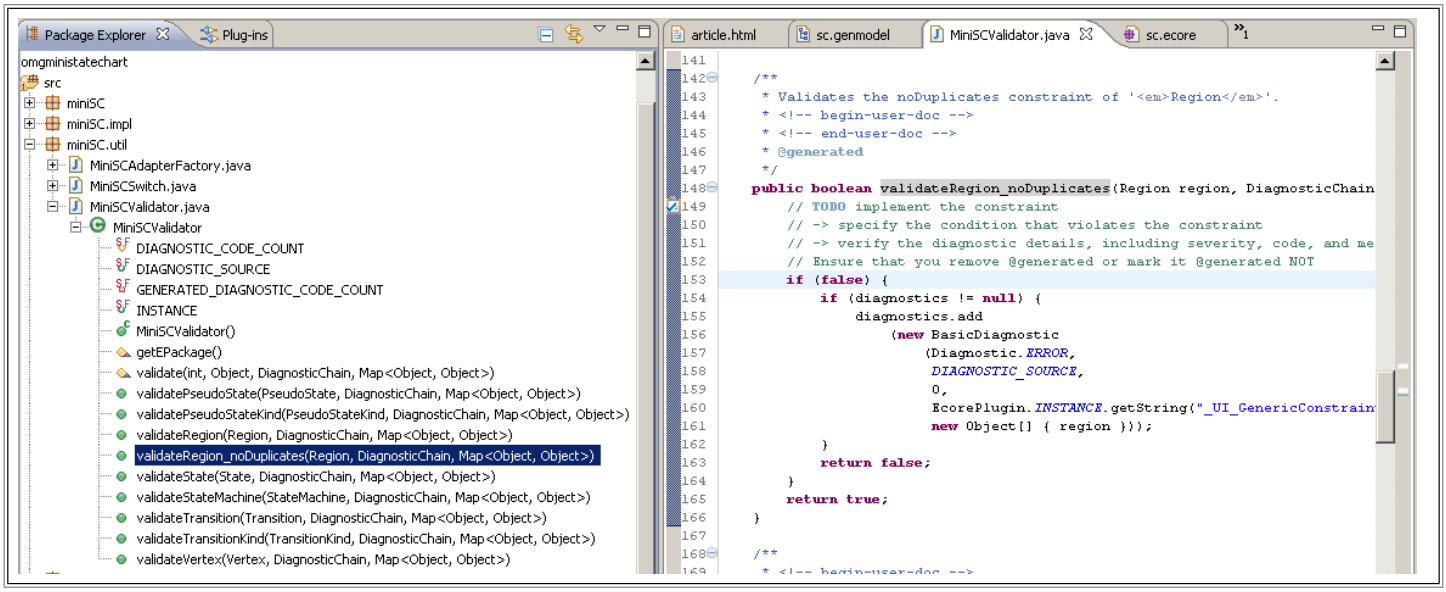


Figure 5: Constraint to complete when no OCL boolean expression was specified

Additionally, the Java method body above can be generated from OCL as explained in the article Implementing Model Integrity in EMF with MDT OCL. The constraint "no duplicate names for states within a region" can be expressed as:

```
self.subVertex->forAll(s1 : Vertex |
    self.subVertex->forAll(s2 : Vertex |
        s1 <> s2 implies s1.name <> s2.name))
```

For that, an additional annotation with source `http://www.eclipse.org/ocl/examples/OCL` is made on `Region`, as shown in Figure 6. The code generated in method `validateRegion_noDuplicates` will parse the OCL constraint and evaluate it (not shown).

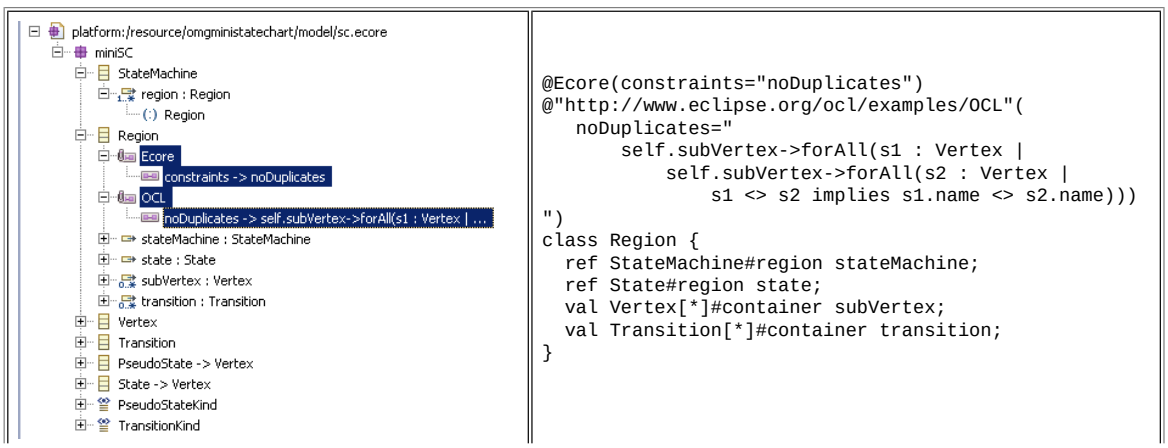


Figure 6:A WFR that `Region`s should fulfill,
as seen in the Ecore editor (left) and in the Emfaticeditor (right)

try it! Armed with this tutorial and with the `DSL2JDT` generator (from the Source Code section), pick your DSL of choice, optionally declare OCL WFRs for it, and start embedding it in the Eclipse JDT!

The majority of the language metamodels available out there lack OCL-based WFRs (remember the story about the cobbler's children?). Those listed below not only include WFRs but also discuss them in some length (we would like to hear about your contributions to this list):

- BPEL 1.1, http://www.cs.kent.ac.uk/pubs/2004/2027/content.pdf
- JPQL 1.0, http://www.sts.tu-harburg.de/~mi.garcia/pubs/atem06/JPQLMM.pdf
- QVT-Relational [URL TODO]

As long as tests are manually coded following the pattern above, all embedded DSL statements will be checked for well-formedness. If the developer overlooks testing some embedded expression, its well-formedness will be known only at runtime (potentially remaining as a bug waiting for happen). The problem is due to the opaque nature (as far as the JDT is concerned) of the embedded DSLs: there is no infrastructure so far to explore the Java code being edited, looking for occurrences of DSL embeddings to check, thus ensuring coverage of WFRs. Achieving such coverage automatically is possible with techniques belonging to *The Grand Plan Way*, the topic of the remaining sections of this article. Before delving into absrtact syntax in those sections, we make first some observations on concrete syntax.

Sidenote: from EBNF grammar to Ecore model and back again

We have been assuming all along that the input to `DSL2JDT` is the metamodel of a DSL, the metamodel that captures the abstract syntax. After all, at the end of the day we want to process ASTs, right? Alas, there are exceptions to that: sometimes we need to process *Concrete Syntax Trees* (CSTs). Let me explain.

In non greenfield scenarios it is often the case that an existing EBNF grammar is available, most likely with a dedicated text editor. Such scenarios have prompted the development of tools to derive an Ecore model from a grammar. The obtained Ecore model *can* be fed as input to `DSL2JDT` (being an Ecore model as any other, `DSL2JDT` won't tell the difference between one representing abstract syntax vs. another representing concrete syntax) thus making possible their embedding in Java. Even if an existing editor is available, embedding may still make sense, for example in the early iterations of porting their AST processing algorithms to EMF. It has been our experience that embedding CSTs makes sense only when unparsing of the CSTs is needed (for example, to generate the input to a legacy tool, a tool not using internally EMF).

The tools in the Textual Modeling Framework allow obtaining an `.ecore` model out of an EBNF grammar. Those tools also generate parsing and unparsing operations, which are inverses of each other (modulo text layout) so that regression tests like the following always pass:

1. parse file `f1` into EMF-based tree `e1`
2. unparse `e1` into file `f2`
3. parse `f2` into `e2`
4. `assert org.eclipse.emf.ecore.util.EcoreUtil.equals(e1, e2);`

So we get an `.ecore` model from EBNF. Is it a "language metamodel"? Not really:

1. when embedded in Java, the CSTs thus built are similar to those prepared by a parser, before the phase where usages are resolved to declarations (i.e. before their conversion to Abstract Syntax Trees)
2. an *Internal DSL* purely generated from an EBNF grammar will lack any constraints to capture static semantics, so you'll have to write them down (which is easier done at the AST level rather than at the CST level)

But, is that a problem? Sometimes it's not. CSTs are ideal for generating structured text (for example, for consumption in a pipes and filters architecture). Besides, the TMF developers have extended EBNF with constructs to specify usual patterns of usages-to-declarations resolving. So the obtained `.ecore` does allow such references. Coming back to pure EBNF, an example of the the CST vs. AST dichotomy for a non-toy DSL can be seen in the Eclipse OCL plugin, where both `OCLCST.ecore` and `OCL.ecore` are available.

To complicate matters further, unparsing can also be done directly from a (well-formed) AST. Given that no layout information is kept there, some pretty-printing mechanism may be necessary. Model-to-text proponents suggest dedicated languages (http://www.eclipse.org/modeling/m2t/). There are DSLs for pretty-printing too, for example one being added to Eclipse IMP (the Box language).

If faced with the alternatives ASTs vs. CSTs, the best choice may be both: before unparsing from a CST, such tree is computed by AST processing. For example, the pseudocode shown left in Figure 7 for a business process can be expanded into the BPEL code shown right. If only the "pseudocode" could be formalized into an embeddable DSL, then its AST could be translated into a CST for unparsing.

Continuing with the example, the DSL part (allowing expressing business processes) need not cover the full spectrum of BPEL (for that, one can directly embed the BPEL metamodel). Rather, the pseudocode-variant could focus on expressing only best practices, which usually amount to subsetting a language. Taking as example another choreography language, the use of XOR-gateways in BPMN programs may express arbitrary (control flow) cycles, just like `GOTO` does in 3GL programs. A "pseudocode" DSL for business proceses could avoid the use of XOR-gateway constructs. The example in Figure 7 and the XOR-gateway observation are reproduced from the diploma thesis of David Schumm (in German).

With this, we conclude our sidenotes on concrete syntax. The remaining sections focus on the advanced uses cases around embedded ASTs, those beyond compile-time well-formedness checking with JUnit.

```
shipOrder := receive();
if (shipComplete) then
```

```
<sequence>
 <receive partnerLink="customer" operation="shippingRequest" variable="shipRequest">
  <correlations>
    <correlation set="shipOrder" initiate="yes" />
  </correlations>
 </receive>
 <if>
    <condition>
                bpel:getVariableProperty('shipRequest','props:shipComplete')
    </condition>
    <sequence>
        <assign>
            <copy>
            <from variable="shipRequest" property="props:shipOrderID" />
            <to variable="shipNotice" property="props:shipOrderID" />
            </copy>
            <copy>
                <from variable="shipRequest" property="props:itemsCount" />
```

```
  shipNotice := shipRequest;
  send(shipNotice);
else
  itemsShiped := 0;
  while (itemsShiped < itemsTotal)
  do ...
```

```
                <to variable="shipNotice" property="props:itemsCount" />
            </copy>
        </assign>
        <invoke partnerLink="customer"
            operation="shippingNotice" inputVariable="shipNotice">
            <correlations>
                <correlation set="shipOrder" pattern="request" />
            </correlations>
                </invoke>
    </sequence>
<else>
    <sequence>
        <assign>
            <copy> <from>0</from>
            <to>$itemsShipped</to> </copy>
        </assign>
        <while>
            <condition>
                $itemsShipped <
                    bpel:getVariableProperty('shipRequest','props:itemsTotal')
            </condition>
            <sequence> ...
```

Figure 7: Further evidence on BPEL's verbosity

Processing DSL statements beyond well-formedness checking

Setting the stage: useful APIs for the task at hand

The JDT incrementally checks the static semantics of Java during editing. A similar capability for embedded DSLs can be achieved by implementing a *compilation participant*:

> A new extension point [as of 3.2] (`org.eclipse.jdt.core.compilationParticipant`) allows plugins that are dependent on `org.eclipse.jdt.core` to participate in the Java build process, as well as in the reconciling of Java editors.
>
> By implementing `org.eclipse.jdt.core.compiler.CompilationParticipant` and extending this extension point, one can be notified when a build is starting, when a clean is starting, or when a working copy (in a Java editor) is being reconciled. During these notifcations, types can be added, changed or removed, build markers can be created, or errors can be reported to the Java editor.
>
> Code that participates in the build should in general be implemented with a separate Builder, rather than a CompilationParticipant. It is only necessary to use a CompilationParticipant if the build step needs to interact with the Java build, for instance by creating additional Java source files that must themselves in turn be compiled.
>
> [Class `ReconcileContext`] ... A reconcile participant can get the AST for the reconcile-operation using `getAST3()`. If the participant modifies in any way the AST (either by modifying the source of the working copy, or modifying another entity that would result in different bindings for the AST), it is expected to reset the AST in the context using `resetAST()`.
>
> A reconcile participant can also create and return problems using `putProblems(String, CategorizedProblem[])`. These problems are then reported to the problem requestor of the reconcile operation.

These excerpts are reproduced from the Javadoc of `CompilationParticipant` and `ReconcileContext`.

What to do with the AST of a Java compilation unit once we have it? Samples answering that question can be found in the reports listed in subsection *Inspection and manipulation of Java ASTs*, under *Related Work*.

For the record, there are at least two other approaches (besides compilation participants) for performing Java language processing: (a) *annotation processors* and (b) an Eclipse workbench builder. Annotation processors are ruled out as they cannot explore the AST of Java method bodies, and thus cannot access the embedded DSL statements. A workbench builder can inspect the AST of the Java compilation units being built, and would otherwise be a viable solution were it not for one of the use cases of interest, *in-place translation*, where such Java AST is modified, as will be seen shortly.

Before getting into the discussion of a sample compilation participant, we review first by means of example the additional uses cases around DSL embedding (in-place translation, statement-level annotations, and DSL-specific views). We believe that the additional implementation effort can be justified if such functionality is encapsulated for reuse across DSLs. Although we're not there yet, this section highlights the design decisions involved (you may interpret this as an invitation to contribute to this project). Unlike the `DSL2JDT` generator it is still in a prototype phase, and has not been checked into CVS.

In-place translation

GUI programming using APIs like Swing or JFace can get quite verbose, a situation that has sparked a number of *GUI description languages* (mostly in the form of XML dialects, usually for interpretation at runtime) such as XUL, AIUML, and XForms, with a longer list at http://en.wikipedia.org/wiki/List_of_user_interface_markup_languages. In terms of Eclipse RCP, the closest examples known to this author are Glimmer (which is Ruby-based and embedded) and StUIML.

Such languages are a prime candidate not only for embedding, but also for in-place translation: we want a JDT extension to expand (say) embedded XUL snippets into their verbose Swing (or JFace or ...) formulation. That way, Java code appearing afterwards may refer to the GUI widgets implicit in the GUI description snippet (for example, to wire event handlers to the widgets, as many GUI description languages only specify the structural and layout aspects of a user interface).

The idea is so compelling that others have already implemented it, which allows us to quote an example from their work and see what adaptations are necessary in the context of `DSL2JDT`. The example we've chosen comes from the JavaSwul DSL, and is itself based on a Sun tutorial example on setting up menus using Swing. The resulting GUI widgets are shown left in Figure 8, with the JavaSwul snippet for them shown just below. Most of the real estate in Figure 8 however is taken up by *an excerpt* of the expanded Swing code *without event handlers*, which is shown on the rightmost column.

The original JavaSwul involves extending the Java grammar and writing so called *assimilators* to desugar JavaSwul snippets into Java ASTs. The resulting embedded syntax looks better (once you've managed to get it right without Content Assist ;-) and has more degrees of freedom than `DSL2JDT`'s bag of tricks (method chaining, static imports, variable length argument lists). In contrast, the approach to embedding favored by `DSL2JDT` does not require up-front knowledge of the productions of the Java grammar. Moreover, one *could* in principle use a compilation assistant to behave as an *assimilator* (i.e., weave information gathered from the surrounding Java AST nodes and the embedded snippets into the output).

The prototype we're building for a GUI description language avoids however the weaving scenario. So far, we've found that self-contained embeddings include all the input

required for our target use cases, because of a simple reason: whenever we've stumbled upon missing input necessary for expansion, we have updated our DSL metamodel to account for it.

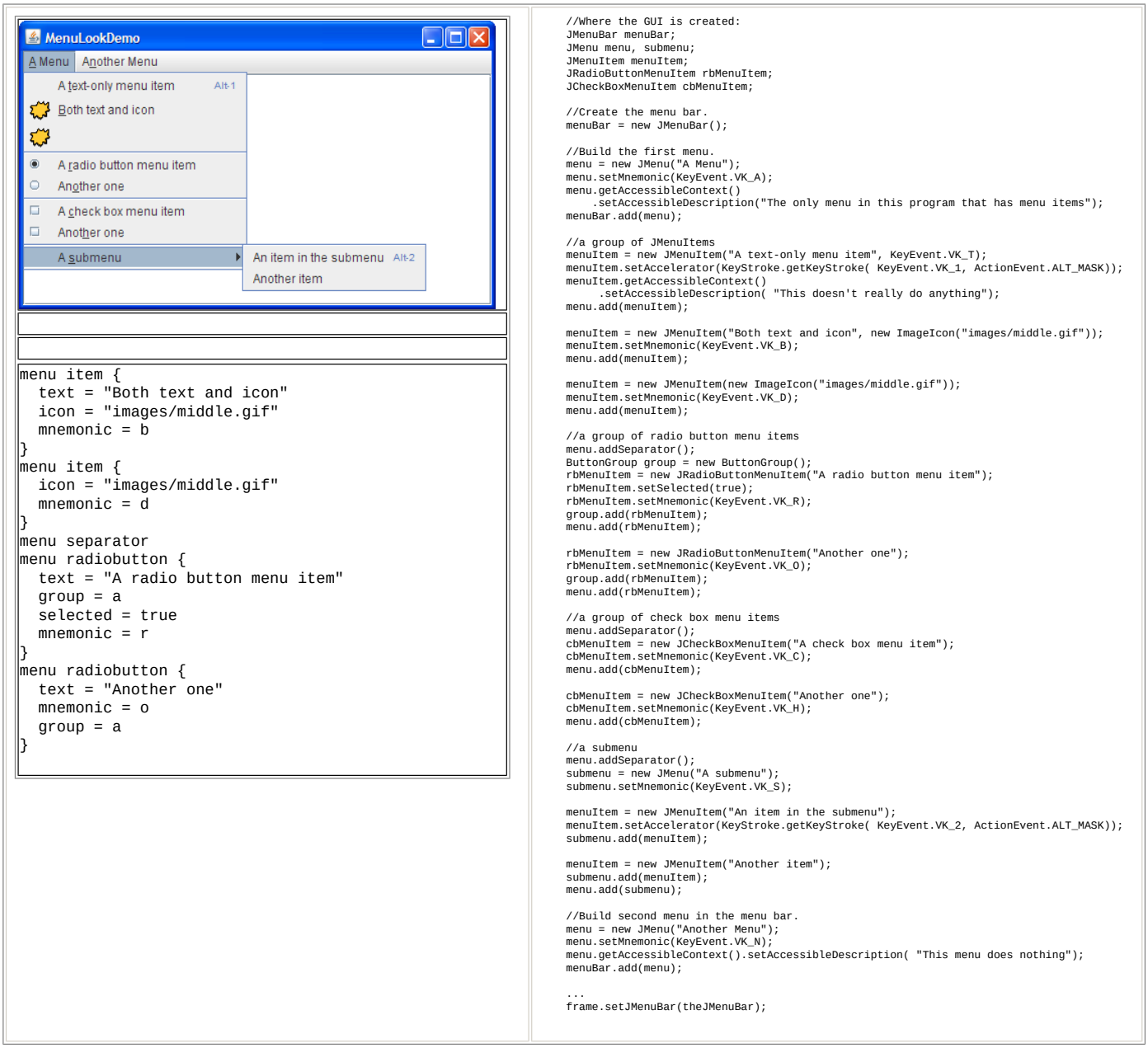


Figure 8: A menu as seen by the user (top left), its GUI description snippet (bottom left), and its abridged Java Swing counterpart (right)

Statement-level annotations

Several language processing applications call for decorating Java programs with additional structured information. A lightweight approach to providing such metadata (short of extending Java syntax) involves defining custom annotations. These and other usages of annotations will only increase. Two examples can be mentioned:

- As part of the ongoing JSR-308 (Annotations on Java types), extensions to the Java 7 syntax are proposed: http://groups.csail.mit.edu/pag/jsr308. The current prototype patches OpenJDK for parsing and for generating bytecode in an extended class format.
- Similarly, Harmon and Klefstad propose a standard for worst-case execution time annotations at the stamement level, metadata that is important for Real-Time Java

The projects above require modifications to the Java grammar, parser, and compiler, thus explaining why those efforts take so long in the making. This integration burden is unfortunate as it stifles innovation, making more difficult the early adoption of language extensions. How many of the following extensions do you regularly use?

- static analyses around references: `@NonNull`, `@Immutable`, `@ReadOnly`. http://groups.csail.mit.edu/pag/pubs/
- bug-finding and verification tools such as JML which extend Java with pre- and postconditions, loop and class invariants, and behavioral interfaces (The JDT vs. non-JDT ways to extend Java syntax for JML are compared in this report)
- security-typed languages such as Jifclipse

As we have seen, embedded DSLs are a non-intrusive way to enrich a Java program with non-Java information. From the point of view of language processing, they lower the cost of proofs of concept. If implemented together with the other use cases described in this section, the resulting IDE extensions are also comparable in usability with dedicated IDEs, as the additional language constructs they manipulate are just that: syntactic *extensions* to Java, not completely new grammars.

DSL-specific views

Some graphical notations are considered standard, with textual counterparts playing a minor role although they convey the same information (for example, musical notation vs. MIDI sequences, bond diagrams vs. chemical formulas, etc.) In these cases, the usability of an embedded DSL would be increased by displaying alongside the textual formulation a read-only view of its 2D or 3D representation. This may be derided as a poor man's WYSIWYG, but as with `DSL2JDT` in general we see instead a lot of leverage being gained from a no-frills architecture. And not to be forgotten, textual notations improve the accessibility of IDE tooling for the visually impaired.

In fact, some Eclipse-based plugins already adopt this "editable text mapped to readonly diagram" metaphor, only that one-way view update is triggered by the build process or a user action. This to make sure that the data source has reached a stable state, unlike the case during interactive editing. For example, the TextUML plugin follows that

metaphor, as shown below, with the PDE Dependency Visualization tool being another case in point.

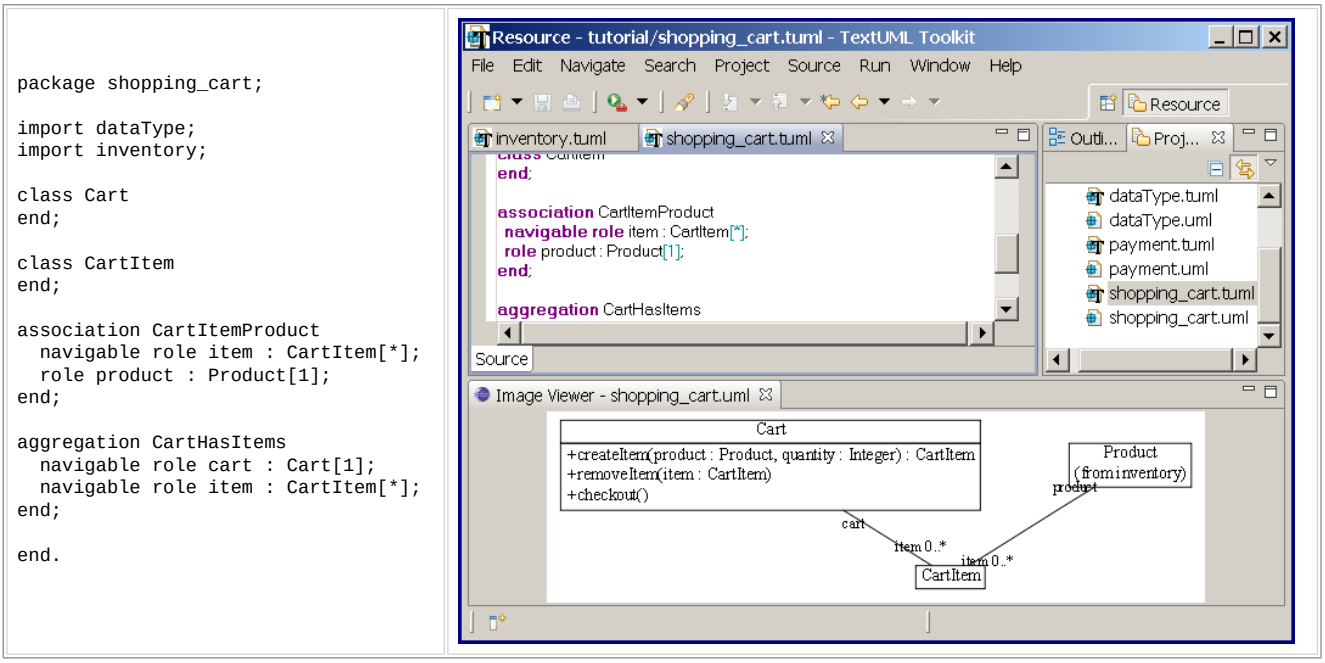


Figure 9: Textual input notation (left) in TextUML, alongside visual (output) notation for feedback
(reproduced from TextUML tutorial)

Given that 2D graph layout libraries are available for Eclipse (for example, GraphViz and Zest) we believe that *a subset of the mapping files created as part of a GMF project* are enough to realize the embedded-DSL-to-diagram use case in the JDT.

Implementation of in-place translation

Of the three advanced use cases, the one we would like to see implemented first is in-place translation. The previous summary of the compilation participant extension point is augmented in this section with an example that shows (a) how to identify Java methods marked with the `ReturnsEmbeddedDSL` annotation, and (b) how to visit the AST of their method bodies. We stop short of translating the embedded DSL (because we're working on that, and we couldn't wait to let others know about our progress with `DSL2JDT` so far).

> tip Although the example in this section directly builds upon the compilation participant API, there are tools and frameworks to simplify the inspection and manipulation of Java 1.5 ASTs. For example, SpoonJDT allows defining *spoonlets*, Java classes that can be plugged in a pipes and filters architecture to process Java ASTs. SpoonJDT also contributes preference pages to configure spoonlets to be active on a per project basis. Interestingly, spoonlets can be developed (and debugged) in the same workspace where the target projects reside (with a compilation participant a second Eclipse instance is required). Finally, a converter from JDT Core ASTs to EMF-based counterparts is available. The prototype we're working on for DSL-specific processing is based on SpoonJDT. We choose however to base our example on the compilation participant API only, as the underlying concepts are the same irrespective of the particular implementation technique.

Listing 4: A compilation participant to add problem markers to methods annotated with `ReturnsEmbeddedDSL`

```
public class MyCompilationParticipant extends CompilationParticipant {

	@Override
	public boolean isActive(IJavaProject project) {
		return true; // springs into action for all Java projects
	}

	@Override
	public void reconcile(ReconcileContext context) {
		super.reconcile(context);
		try {
			org.eclipse.jdt.core.dom.CompilationUnit ast = context.getAST3();
			org.eclipse.jdt.core.dom.ASTVisitor myVisitor = new MyVisitor(); // see declaration below
			for (Object oTypeDecl : ast.types()) {
				if (oTypeDecl instanceof org.eclipse.jdt.core.dom.TypeDeclaration) {
					TypeDeclaration td = (TypeDeclaration) oTypeDecl;
					for (MethodDeclaration md : td.getMethods()) {
						for (Object oModifier : md.modifiers()) {
							if (oModifier instanceof org.eclipse.jdt.core.dom.Annotation) {
								Annotation ann = (Annotation) oModifier;
								String fqn = ann.getTypeName().getFullyQualifiedName();
								if ("dsl2jdt.annotation.ReturnsEmbeddedDSL".equals(fqn) ||
								     "ReturnsEmbeddedDSL".equals(fqn)) {
									addSampleProblem(ast, md, context);
								}
							}
						}
					}
				}
			}
			ast.accept(myVisitor);
		} catch (JavaModelException e) {
			e.printStackTrace();
		}

	}
```

```
        private void addSampleProblem(CompilationUnit ast,
               MethodDeclaration md, ReconcileContext context) {

                char[] originatingFileName = ast.getJavaElement().getPath().toOSString().toCharArray();
                String message = "default dsl2jdt error message";
                int severity = ProblemSeverities.Error;

                int startPosition = md.getName().getStartPosition();
                int endPosition = startPosition + md.getName().getLength();

                int line = -1;
                int column = -1;

                EmbeddedDSLProblem pro = new EmbeddedDSLProblem(
                        originatingFileName, message, severity, EmbeddedDSLProblem.NO_ARGUMENTS, severity,
                                startPosition, endPosition, line, column);
                CategorizedProblem[] problems = new EmbeddedDSLProblem[] { pro };
                context.putProblems(EmbeddedDSLProblem.DSL2JDT_PROBLEM_MARKER, problems);
                // see also IJavaModelMarker

        }

        @Override
        public void buildStarting(BuildContext[] files, boolean isBatch) {
                // TODO Auto-generated method stub
                super.buildStarting(files, isBatch);
        }

}
```

Listing 5: And the accompanying visitor (more examples can be found in *Static Analysis for Java in Eclipse*)

```
package compa.basic;

import org.eclipse.jdt.core.dom.SimpleName;

public class MyVisitor extends org.eclipse.jdt.core.dom.ASTVisitor {

  public boolean visit(org.eclipse.jdt.core.dom.MethodInvocation inv) {
    org.eclipse.jdt.core.dom.Expression rcvr = inv.getExpression();
    // null if implicit 'this' call
    System.out.println(inv);
    if (rcvr == null) { //  skip
      return false; // don't bother looking at children (actual arguments)
    } else if (!(rcvr instanceof org.eclipse.jdt.core.dom.SimpleName)) {
      return true; // examine children (actual arguments)
    }
    org.eclipse.jdt.core.dom.SimpleName rcvrNm = (SimpleName) rcvr;
    org.eclipse.jdt.core.dom.IBinding rcvrBinding = rcvrNm.resolveBinding();
    System.out.println(rcvrBinding);
    return true;
  }

  public boolean visit(org.eclipse.jdt.core.dom.MethodDeclaration node) {
    return true;
  }

  public boolean visit(org.eclipse.jdt.core.dom.TypeDeclaration node) {
    return true;
  }

}
```

Related Work

Language tooling is a vast field. We summarize four areas directly related to DSL embedding: (a) proposed embeddings in other languages (Scala and Ruby), (b) well-formedness checking over XML artifacts, (c) inspection and manipulation of Java ASTs, and (d) the competing approach of IDE generation.

DSL Embedding in Scala and Ruby

The syntax of Java 5 contributes to the readability of internal DSLs (variable length argument lists, static imports). Still, DSLs embedded in Java cannot circumvent the `subject.verb(object)` bias of the language: no additional infix operators can be defined nor existing ones overloaded. In Scala, binary operators can be overloaded. The resulting advantages for DSL embedding are reported by Dubochet in this paper. In turn, DSL embedding in functional languages has a long tradition, Leijen and Meijer were already reporting in 1999 how to embed SQL in Haskell. Although superficially similar to other embedding efforts like SQL/J, the DSL embeddings we're talking about do not require modifying the front-end of a compiler, as is the case with SQL/J.

DSL embedding is also popular with dynamically typed languages. Two recent examples in Ruby include:

- Glimmer, an embedding of a high-level language for JFace/SWT programming
- embedding SVG: SVuGy and RVG

Both Scala and Ruby allow for a more compact notation, and the same techniques reported here can be applied in their respective IDEs to take care of well-formedness checking at compile time. That might suggest they are a better choice for DSL embedding. We see it differently. To us, what all these examples have in common is the tension between *language-level* as opposed to *IDE-level* extensibility, a matter that exceeds the particular host-embedded language pair being considered. Our reasoning can be summarized as follows: as long as the JDT (including extensions) allows for reasonable solutions, it pays off to stick with it for DSL embedding. Or maybe it's just me who don't know how to write auto-morphing code in Scala ("ASTs as first-class citizens"). In any case, the debate will likely go on among the language camps.

Besides, any improvements to Content Assist in JDT can be leveraged by all DSL embeddings in Java. For example, ideas around API completion as a *planning problem* have

been explored in Prospector. Unlike with custom generated IDEs, we benefit from all those improvements for free.

Heuristics to derive more "abstract" metamodels out of EBNF are discussed by Kunert in his paper Semi-Automatic Generation of Metamodels and Models from Grammars and Programs.

Static analysis of XML artifacts

The proliferation of XML dialects has prompted the development of tools to check good old static semantics. A tool in this problem space is SmartEMF, being developed by Hessellund as part of his PhD. He identifies typical kinds of integrity constraints to check across the XML artifacts developed for consumption by some framework (for example, referential integrity constraints across configuration files in projects extending the Apache Open for Business (OFBiz) framework). Once such constraints have been made explicit, SmartEMF takes charge of checking them. Additionally, those editing operations that are feasible for the current editing state are found, much like Content Assist works in the JDT:

> Given a portfolio of metamodels specified in SmartEMF, i.e., DSLs conforming to Ecore, we can represent languages, domain constraints, and models in a uniform way. All artifacts are mapped into a single constraint system implemented in Prolog that facilitates constraint checking and maintenance, and allows us to infer possible editing operations on a set of models.
>
> Anders Hessellund. *SmartEMF: Guidance in Modeling Tools*. Doctoral Symposium, OOPSLA'07, Montreal, Canada, October 2007.
> http://www.itu.dk/people/hessellund/work/Hessellund07b.pdf

Taking into account the large number of XML dialects in use today, it makes sense to think about ways to embed them in Java, while keeping the XML format as a serialization format (for communication between machines, not humans). We have not explored this scenario with a case study, but plan to do so (and would like to hear about the application of `DSL2JDT` for this purpose). After all, although Scala supports an object syntax for XML, the Scala IDE does not check the well-formedness of whatever DSL that XML represents.

Proposals are regularly made around non-XML syntaxes for XML dialects, a case in point for XUL (GUI description language) is the shorthand syntax Compact XUL. A once-and-for-all solution to this recurrent problem is offered by Dual Syntaxes: http://www.brics.dk/~amoeller/papers/xsugar/journal.pdf

Inspection and manipulation of Java ASTs

The SpoonJDT tutorial contains examples of in-place code modifications (not in-place translations, however) such as adding Javadoc and preconditions to existing methods.

The processing of ASTs is the focus of the following reports:

- Robert M. Fuhrer, *Static Analysis for Java in Eclipse*
- Thomas Kuhn, Olivier Thomann. *Abstract Syntax Tree*. Eclipse Technical Article,
- Tobias Widmer. *Unleashing the Power of Refactoring*. Eclipse Technical Article,
- Manoel Marques *Exploring Eclipse's ASTParser: How to use the parser to generate code*. DeveloperWorks article.
- ASTView, visualization of AST of Java source file. http://www.eclipse.org/jdt/ui/astview/index.php
- Other plugins involving code management: http://eclipse-plugins.info/eclipse/ plugins.jsp?category=Code+mngt

A capability similar to in-place translation is realized by Octel, where instructions about what to generate appear in Java comments following an XML syntax. For example, the comment below generates a setter method for an UML attribute:

```
public void generateAttributeInClass(IStructuralFeature att, OJClass owner){

  FEATURE = new StructuralFeatureMap(att);
  owner.addToImports(FEATURE.javaTypePath());

  /**<octel var="owner">
     <method type="%FEATURE.javaTypePath()%"
       name="%FEATURE.setter()%"
       static="%FEATURE.isStatic()%"
       visibility="%FEATURE.visibility()%">
       <comment> implements the setter for feature '%att.getSignature()%' </comment>
       <param type="%FEATURE.javaTypePath()%" name="element"/>
       <body>
         <if> %FEATURE.javaFieldName()% != element
         <then>
           %FEATURE.javaFieldName()% = element;
         </then></if>
       </body>
     </method>
  **/
}
```

Given that the DSL input appears as a comment, there is no guidance on the part of the IDE about how to get the dedicated syntax right, nor compile-time checking of its static semantics.

Checking Java source code beyond static semantics is the realm of tools like FindBugs, which operates in a batch manner. Other tools however perform background yet non-incremental checks, as implemented by EzUnit and by the *Continuous Testing Plug-in for Eclipse*:

> Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background, providing rapid feedback about test failures as source code is edited. It reduces the time and energy required to keep code well-tested, and prevents regression errors from persisting uncaught for long periods of time.

Competing approach: IDE generation

Before getting involved with Internal DSLs and starting the `DSL2JDT` tool, I spent my fair amount of time with IDE generators. So I guess a comparison is in order. Here it goes.

The generation of custom text editors is an active field. The following is a partial list (in alphabetic order) of projects offering such capability:

- MontiCore, http://www.sse-tubs.de/monticore/
- Sdf2imp, https://svn.strategoxt.org/repos/WebDSL/imp/trunk/
- TCS, http://wiki.eclipse.org/index.php/TCS, part of the Eclipse Textual Modeling Framework (TMF)
- TEF, http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/index.html

- xText, http://wiki.eclipse.org/Xtext, part of the Eclipse Textual Modeling Framework (TMF)

By itself, a custom text editor generated from a grammar alone does not enforce the static semantics of the DSL (which by definition, are those well-formedness rules that exceed the expressive power of the grammar). So some additional coding is necessary. Those text editors internally maintaining an Ecore-based representation of the AST simplify the integration of such additional code.

The Eclipse IDE Meta-Tooling Platform Eclipse IMP goes beyond the generators above in that it aims at generating debugging infrastructure, moreover enabling the integration of complex analyses, such as control- or data-flow based. The integration of translation capabilities remains however the task of the developer. Another tool addressing debugging (and visual interpretation) of a DSL is EProvide. Eclipse IMP is rather unique in addressing user-provided analyses, which can get quite elaborate very quickly. For example, a web search for the phrases "sql injection" and "static analysis" will return papers describing such analyses, ready for implementation.

The ASTs we embed with `DSL2JDT` have all been self-contained: their terminals are compile-time constants. We also skipped on providing any kind of refactoring support for the embedded DSL, as they are necessarily DSL-specific. Similarly, staged compilation, partial evaluation, and weaving (to account for the surrounding Java AST nodes) are all very interesting yet unsupported use cases from the `DSL2JDT` perspective. Completing the infrastructure put forward in this article is a first step towards enabling the implementation of DSL-aware language processing in the JDT.

Conclusions

We see many application areas for embedded DSLs, with the discussion about in-place translation and DSL-specific views just showing some of the possibilities. All along we've tried to maintain the main value proposition of well-designed DSLs: offering an easily consumable form of expert knowledge. We think embedding makes a DSL only easier to consume.

In particular, the capability to perform in-place translation brings together two seemingly opposite camps: those favoring "abstractions in DSLs" and those promoting design patterns. As we have seen, in-place translation keeps side by side the source DSL statements and their Java translation (wich follows the design patterns captured by the DSL implementation).

Open platforms like Eclipse and EMF (and their communities) make possible the kind of cross-pollination that `DSL2JDT` has benefited from. Now it's your turn to take these techniques to a next level.

Acknowledgments

An initial version of the statechart example was developed by Paul Sentosa as part of his master thesis on generating text editors for custom DSLs. The concepts in Martin Fowler's online notes on *Internal DSL* acted as a catalyzer to develop `DSL2JDT`.

Source Code

TODO There is an issue with the URIs that the the `.genmodel` may contain (this is due to my lack of expertise with EMF URIs :-) In all examples (including `omgministatechart`), I'm using workspace-relative URIs, of the form `"platform:/resource/omgministatechart/model/sc.ecore#//PseudoStateKind/initial"`.
If you use other kinds of URIs then
method `generateInner(IFile genModelFile, IProgressMonitor monitor)`
in class `org.eclipse.gymnast.generators.embeddeddsl.EDSLGenerator`
won't be able to get the contents of the `.genmodel` file.
I have no idea why. I only know that I'm opening the file with `URI.createFileURI(genModelPath.toString())`. Help is welcome.

- `DSL2JDT` can be downloaded from CVS (user `anonymous`, host `dev.eclipse.org` , repository path: `/cvsroot/modeling`). And then
    1. `HEAD`
    2. `org.eclipse.emf`
    3. `org.eclipse.emf.emfatic`
    4. `plugins`
    5. check out `org.eclipse.gymnast.generators.embeddeddsl`
    6. check out `org.eclipse.gymnast.generators.embeddeddsl.ui`
- The Statechart example is available for import into the workspace as a zipped Eclipse project: `omgministatechart.zip`