TUHH
Technische Universität Hamburg-Harburg

STS

Enriching EMF models with behavioral specifications

submitted by
Elena Tibrea

supervised by
Prof. Dr. Ralf Moeller
Msc. Miguel Garcia

Hamburg University of Science and Technology
Software Systems Institute (STS)

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 06.07.2006
Elena Tibrea

For time invested into discussions, for providing books, e-books, for pointing out interesting links, I need to thank my supervisor, who's attitude towards work can be best described as professional.

Thank you, Miguel Garcia!

Contents

1 Introduction **7**

2 Background on EMF **8**

2.1 EMF Core 8

2.2 EMF.Edit 11

2.3 EMF.Codegen 13

3 Behavior achieved through rule engines **15**

3.1 Drools engine functionality 16

3.1.1 Domain specific language 18

3.2 Example EMF model working with Drools 19

3.2.1 Problem statement 19

3.2.2 Rules defined 20

3.2.3 How does EMF tackle the rules defined? 21

4 Behavior achieved through statemachines **24**

4.1 Define models statecharts generation 24

4.1.1 Switch based approach 24

4.1.2 State pattern approach 25

4.1.3 Object oriented approach based on a predefined statecharts metamodel 25

4.2 Case study on reactive models 27

4.2.1 Switch based approach 27

4.2.2 State pattern approach 29

4.2.3 Object oriented approach based on a predefined statecharts metamodel 29

4.3 State pattern generation based on a compact model 31

4.3.1 Case study.Problem.Solution 33

5 Conclusions **39**

A Actions and commands for multi-rooted resources in EMF 40
A.1 Problem statement 40
A.2 Solution suggested 40
A.3 Problematic approach 44

B Decision table for a state machine definition 45

C Statecharts editor based on GMF 47
C.1 Constraints 48
C.2 Validation 48
C.3 Custom images 49

List of Figures

2.1 A model visualized with Omondo 10
2.2 Command pattern used in EMF.Edit 12
2.3 Observer pattern 13
2.4 From left to right:Ecore,GenModel,Generate Model action,Generated code structure 13
2.5 Code generation overview 14

3.1 Drools authoring 16
3.2 AST objects which form a rule 17
3.3 Drools runtime 18
3.4 Working Memory view, Agenda view, Audit view 18
3.5 Rule authoring with DSL 19
3.6 Add a book to existing folder rule 20
3.7 Create folder rule 21

4.1 State charts metamodel(SMM) 26
4.2 Microwaveoven statemachine 27
4.3 Model definition for switch based approach 28
4.4 Reactive object based on SMM 29
4.5 Omondo view of the OO model 30
4.6 EMF.CodeGen structure 32
4.7 Carousel door statemachine 33
4.8 Statemachine modeling with the help of state pattern 34
4.9 Package structure after code generation 37
4.10 What should be taken out in order to enhance the generation 38
4.11 Class diagram used for SP generation 38

A.1 The problem encountered 41
A.2 The desired result 41
A.3 Modifications required in the plugin descriptor 42
A.4 XCommand, XActionBarContributor,XAction classes 43

B.1 Decision table for a state machine 45
B.2 Template rule with parameters defined in the excel worksheet 46

C.1 Statecharts editor- simple state, initial, final state 47
C.2 Statecharts editor- composite states with inner states 47
C.3 Defined link constraints 48
C.4 Audit constraints as validation rules 48

Chapter 1

Introduction

This report contains a series of tryouts, which involve dynamic aspects of models as represented by Eclipse Modeling Framework (EMF).

EMF follows a model driven approach targeting software development.Transformations taken during a classical MDA(Model Driven Architecture) process are addressed by EMF:

- an ECore metamodel is first transformed to another model (known as GenModel), which includes additional information about the structure and organization of future code to be generated. In this case Ecore represents the Platform Independent Model(PIM), while Platform Specific Model is represented by GenModel.
- the second step to be taken is from PSM to the actual code

Automatic transformation chain support that EMF provides, increases developers productivity and lowers code maintenance.

Behavioral aspects are equally interesting for any model, our purpose here being to emphasize behavior of EMF models.

Chapter 2 aims to give the reader, a basic introduction to EMF, by drawing user's attention to some of the features EMF.Edit, EMF.Codegen and EMF Core provide.

Chapter 3 shows a way to externalize behavior by using rule engines.

Chapter 4 of this report investigates several issues around behavioral state machines, which represent a way to capture dynamic capabilities of a system.

Chapter 2

Background on EMF

Summary. *The user will find out about EMF's generator facility, EMF Core features and patterns used inside the most dynamic part of EMF: EMF.Edit. Many valuable issues around Eclipse Modeling Framework are left out(e.g EMF resources, persistence). The purpose of the chapter is not to be a survey of existing literature, but to draw user's attention to some specific points(e.g patterns used in EMF.Edit).*

Eclipse Modeling Framework[2](EMF) provides a set of cooperating classes that make up a reusable design, targeted not only to represent abstract views of the software(called models), but also to generate code.

EMF consists of three main parts:

EMF Core comprises model description complying ECore, model persistence, change notification and reflective API.

EMF.Edit supports the creation of EMF model editors, by including content and label provider classes and command framework.

EMF.Codegen provides the infrastructure for generating all the needed artifacts for an EMF model editor.

A more detailed description of the above mentioned parts follows.

2.1 EMF Core

In the context of Model Driven Engineering, Object Management Group(OMG) has defined Meta Object Facility(MOF) which standardizes meta model definitions in object oriented world. EMF models are described by a meta-model named *ECore* which follows the specifications of EMOF[1] (Essential MOF).

There are at least three ways to define Ecore models:

[1]page 31 in Meta-Object Facility Specifications at http://www.omg.org/docs/ptc/03-10-04.pdf

- **Java interfaces with annotations**

  @Model annotations help us to enrich the initial model definition, by allowing attributes which can specify default values, containment feature(useful for serialization) or object type when a list is returned.

  Listing 2.1: Example Java interface with annotations

```
/**
 * @model
 */
public interface Book {
  /**
   * @model
   */
  String getTitle();
  /**
   * @model default="100"
   */
  int getPages();
  /**
   *  @model
   */
  BookCategory getCategory();
  /**
   * @model opposite="books"
   */
  Writer getAuthor();
}
```

- **XML Schema**

  Some of the mapping rules between EMF world and XML Schema are: a simple type maps on *EDataType*, an attribute is the corespondent of *EAttribute*, while a complex type maps always to a class.

  Listing 2.2: 'Book' defined as a complex type

```
<xsd:complexType name="Book">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="pages" type="xsd:int"/>
      <xsd:element name="category" type="lib:BookCategory"/>
      <xsd:element name="author" type="xsd:anyURI"
           ecore:reference="lib:Writer" ecore:opposite="books"/>
    </xsd:sequence>
</xsd:complexType>
```

- **UML**, by using a modeling tool(e.g Omondo, Rational Rose)

The most common and concise is to use an UML tool and define the model visually, gaining a plus of expressiveness, but if one of the goals is to customize the persistence

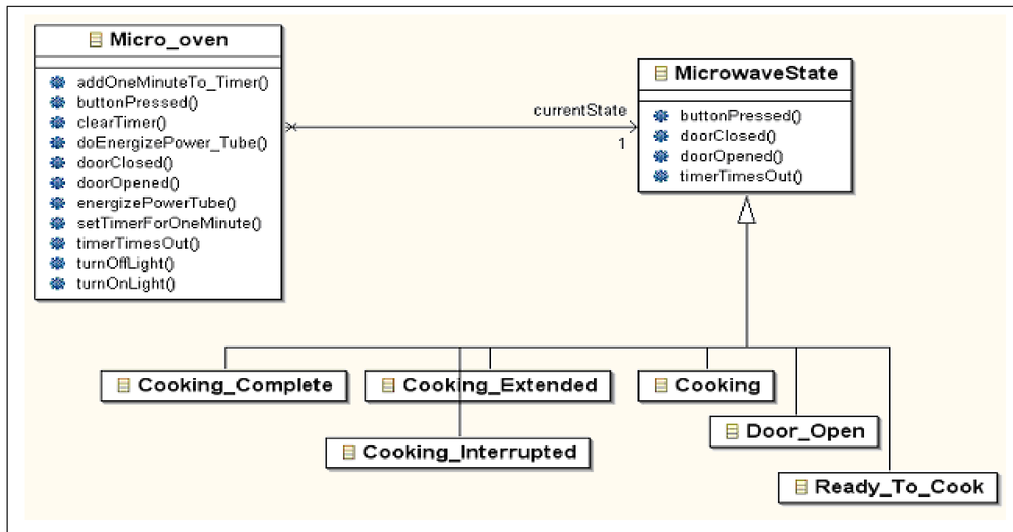

Figure 2.1: A model visualized with Omondo

mechanisms, describing the model using XML Schema will help reach the target. The available paths to be taken when defining a model, are not really equivalent (e.g in XML schema, we can not define bidirectional references), but there are means to correct the generated model(at a later point, by using ecore editor).

EMF's reflective API allows access at run time to the any defined model. *EObject*, the parent of any EMF object, defines reflective methods.

Listing 2.3: EObject reflective methods

```
public interface EObject extends Notifier
{
  // Returns the meta class.
  EClass eClass();

  // Returns the containing object, or null.
  EObject eContainer();

  // Returns the value of the given feature of this object.
  Object eGet(EStructuralFeature feature);

  // Sets the value of the given feature of the object to the new value.
  void eSet(EStructuralFeature feature, Object newValue);

  // Returns whether the feature of the object is considered to be set.
  // usefull for serialization purposes
  boolean eIsSet(EStructuralFeature feature);
  ....
}
```

Reflective methods can be noticed when generating the model code, but this is

not the only use of the EMF reflection API. Instances of non generated classes can also be altered or queried by means of reflection.

Listing 2.4: Reflective eGet(...)generated method

```
public Object eGet(int featureID, boolean resolve, boolean coreType) {
        switch (featureID) {
            case LibraryPackage.WRITER_NAME:
                return getName();
            case LibraryPackage.WRITER_BOOKS:
                return getBooks();
        }
        return super.eGet(featureID, resolve, coreType);
    }
```

As mentioned previously notification responsibilities are borne by every *EObject*, which is a notifier in the first place, by implementing the *Notifier* interface.Due to this intrinsic property of EMF objects, registered observers can be notified of changes.

Listing 2.5: Register observers through *eAdapters()* method

```
Book b=LibraryFactory.eINSTANCE.createBook();
Adapter bObserver=...
b.eAdapters().add(bObserver);
```

Usually, EMF observers are assigned to objects by adapter factories and not using the *eAdapter()* method.At notification time, *notifyChanged()* method of the registered adapter is called to deal with recent changes.

2.2 EMF.Edit

For a better interaction with the user and for highlighting some of the above mentioned capabilities, a model editor is desirable. *EMF.Edit* comes into play, by bridging the EMF's model and the editor. JFace viewers used by the editor require content provider and label provider objects for being able to manage and display model objects.

The advantage of using content providers resides in releasing the model objects of the responsibility to specify how can an objects be viewed. It also prevents the editor from finding on its own, an algorithm to bring the data into JFace viewers. GUI components and model objects talk to each other by making use of objects which implement *ITreeContentProvider* interface.

Listing 2.6: ITreeContentProvider interface

```
public interface ITreeItemContentProvider extends
IStructuredItemContentProvider
{
  public Collection getChildren(Object object);
  public boolean hasChildren(Object object);
  public Object getParent(Object object);
}
```

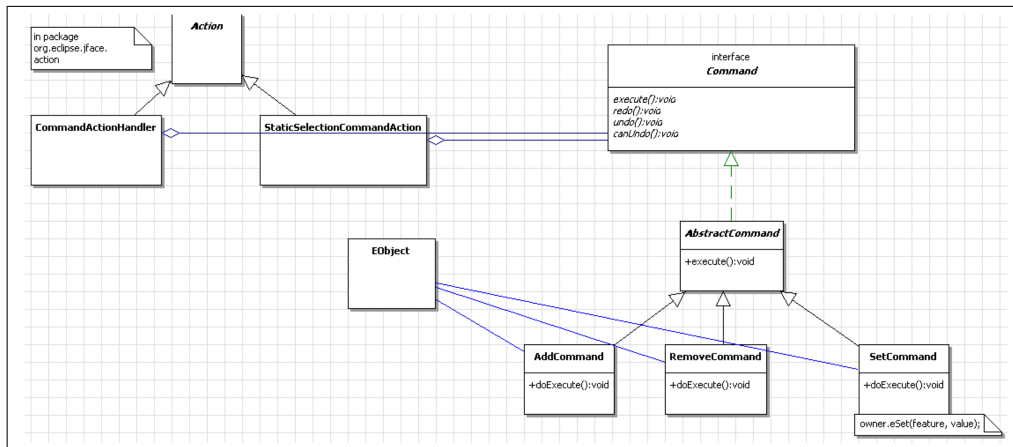

Figure 2.2: Command pattern used in EMF.Edit

EMF.Edit encompasses a set of commands, the user can use for various purposes like: adding new children to an *EObject*, remove an object form a model, set an attribute's value, etc.

The *Command pattern* is used here, to encapsulate requests as objects and support undoable operations. For those interested to recognize the participants of the pattern as described by GoF[1][2], *Command* and *ConcreteCommand* roles can be easily inferred, while the Invoker's role is played by an *Action* class and the *Receiver* is always an *EObject*.

Creating, managing, maintaining the command stack (useful for undoing commands) are the responsibilities of the *EditingDomain*, part of the editor.

Implementation of change notification is a common ground for both EMF core and EMF.Edit framework. This can be best acknowledged by inspecting the generated model code and the generated code for edit.

An *EObject* is the subject, the observable (see fig.2.3), also called in EMF notifier (because *EObject* extends the Notifier interface), while the observer is called *Adapter* because besides it's observer role, it has assigned other responsibilities as well.

Not such an easy thing in EMF.Edit is to understand that the generated *ItemProviders* are multi functional.They act as factories for commands, they are observers for the model objects and they are said to be also, adapters for model objects. In the sense GoF[1][3] describes the *Adapter pattern*, the participants :*Target, Adapter, Adaptee* are difficult to be identified. Making the logical assumption the EObjects are the adaptees and the ItemProviders are the adapters, at a closer code inspection one won't be able to find neither aggregation, nor inheritance used to link the *Adapter* and the *Adaptee*.

[2]p.233 in GoF
[3]p.139 in GoF

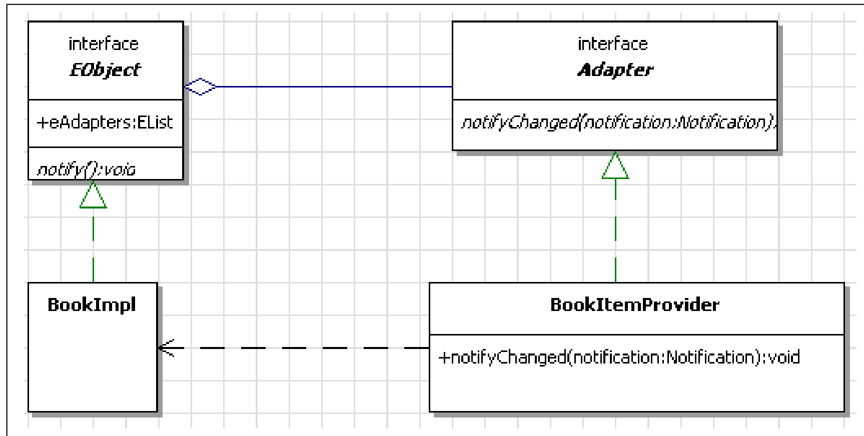

Figure 2.3: Observer pattern

For an example using actions and commands, please refer to appendix A

2.3 EMF.Codegen

EMF's code generation facility is able to generate all the needed resources for a complete editor to be built.At the editor level, we manipulate model objects, the model and the editor are two of the three different code generation levels, that EMF provides us with. The glue between the model and the editor, is responsible for content and label providers, as well as for adapting the model objects for various editing tasks. EMF.Edit governs this middle code generation level.

For generating the model code one should follow the series of actions below:
For generating edit and editor code levels, one should choose *Generate edit* followed

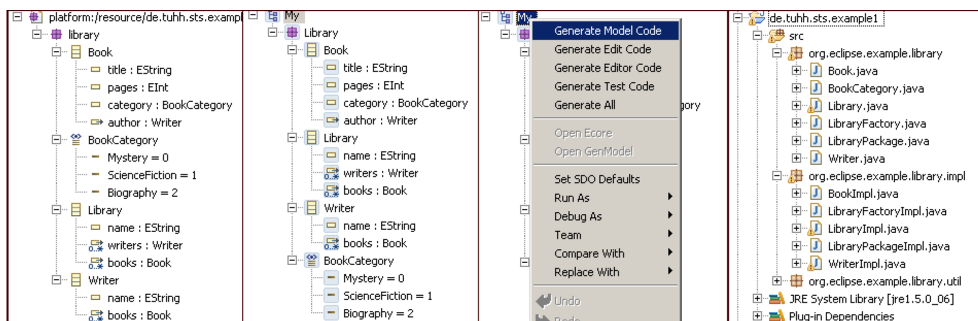

Figure 2.4: From left to right:Ecore,GenModel,Generate Model action,Generated code structure

by *Generate editor* from GenModel context menu.

The following picture serves as an overview of the code generation process.¡The input of the generator is *GenModel*. Although, *Ecore* model supplies the content, for

what is supposed to be finally generated, the information encapsulated in ECore lacks additional specification like: name of packages to be generated, reflective methods are to be generated or not, external templates to be used, etc.

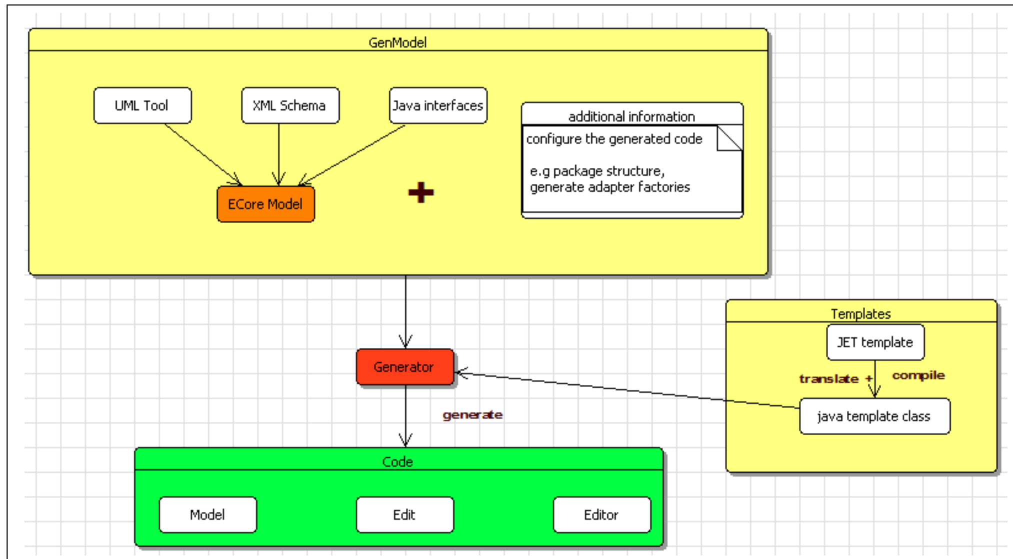

Figure 2.5: Code generation overview

Having only the model, but not the 'patterns'(templates) to be used and filled in, the generator can not accomplish its tasks.

Java Emitter Templates(JET) helps us to define Java Server Pages(JSP) like templates.Users familiar with JSP expressions, scriptlets, object passing as input parameters will find the same functionality of the prior mentioned elements, in JET too.

Listing 2.7: JET template fragment

```
<%if (!isImplementation)
{%>
 <%=genFeature.getListItemType()%>
                              get<%=genFeature.getAccessorName()%>(int index);
<%} else {%>
  public <%=genFeature.getListItemType()%>
                              get<%=genFeature.getAccessorName()%>(int index)
  {
      return (<%=genFeature.getListItemType()%>)
                            <%=genFeature.getGetAccessor()%>().get(index);
  }
<%}%>
```

As can be seen in the picture 2.5, the generation process having as a source a JET template, is a two step process: *translation* followed by *compilation*(JET templates are transformed in java source files and than compiled) and generation (the bytecode is used to generate all the necessary files for the model, edit and editor).

Chapter 3

Behavior achieved through rule engines

Summary. *A brief introduction to the context which requires rule engines existence, will precede a section explaining the basic functionality of JBoss Rules.The last section of this chapter is dedicated to an example, showing how an EMF model and Drolls work together, emphasizing some of JBoss Rules (Drools) features like domain specific language (DSL).*

Static models could not exist in isolation, in order to be useful, to meet demands and users expectations within an application context. Large amounts of data are difficult to be dealt with, by the business layer of any application, when decision making factors are involved to produce the desired outcome. Sometimes, the inner logic of an application might not be overwhelmingly complex, but the frequency of needed changes is high.

By formulating statements of truth and specifying the actions to be taken, as rules, parts of the business logic layer can be externalized and further processed by rule engines. We can infer from here, that rule engines are nothing but software modules, responsible for processing priori specified rules.

Working with EMF models or just POJO based models(Plain Old Java Objects) makes no difference for the rule engines. The objects the rule engine is working with, have to obey though some rules(e.g Drools works with JavaBeans objects).

Independent of the rule engine used (e.g Jess, Drools, ILog Business Studio,Versata), there are some advantages of using inference engines, like:

Separation of concerns and centralization of data is clearly a positive issue when the logic changes often, otherwise it may seem like breaking one of the basic principles of object orientation: encapsulation.

Declarative programming readable by domain experts, is another important fea-

ture of rule engines. By using a declarative style in specifying the rules, we describe what we want, but not a way to accomplish the task, like in procedural programming.

3.1 Drools engine functionality

There are many rules engines available on the market, some of them targeting Java platform, like: *Jess Rule Engine*[1] or *JBoss Rules*[2] (Drools 3.0), but there are also rule engines for .NET like,*inrule*[3].

We will take a brief look at Drools engine, with the remark that rules engine functionality does not differ so much in concepts or structure.

Before rules can be executed, they have to be created, parsed and built (the sum of code generation and compilation). The afore mentioned process bears the name of rule authoring. The environment in which rules are executed and objects (called facts) asserted is called working memory, the essential component of Drools runtime.

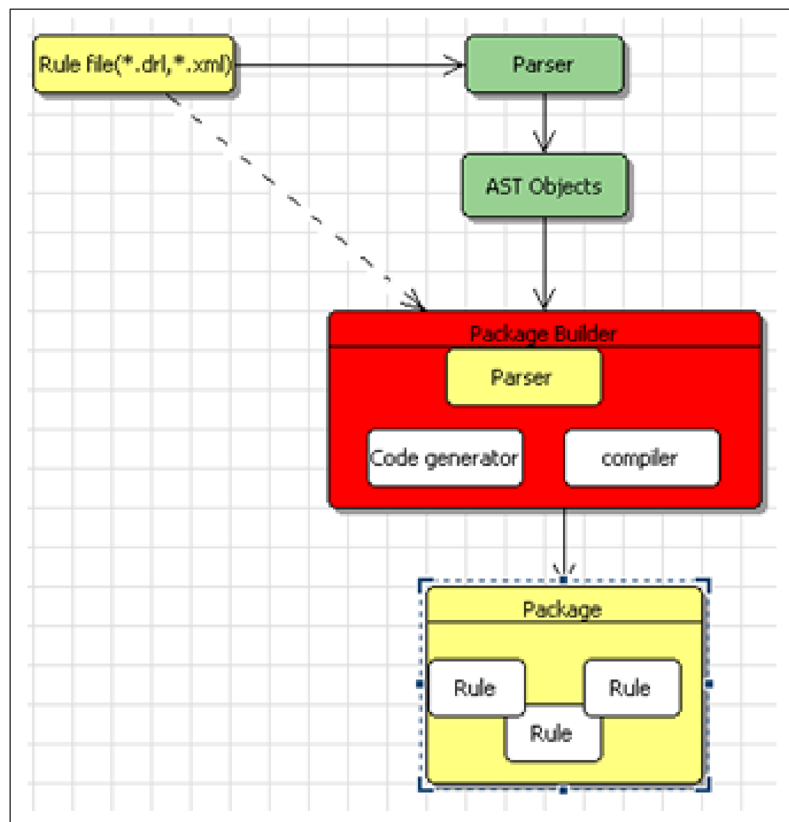

Figure 3.1: Drools authoring

[1]http://www.jessrules.com/
[2]http://www.jboss.com/products/rules
[3]http://www.inrule.com/

A brief description of the most important components involved, follows:

- **A rule file** can be delivered in xml or drl format. For editing a rule in drl format the user has the benefit of an integrated editor support. An illustrative example for a rule in drl format is:

Listing 3.1: Rule example

```
rule "Hello World"
    when
        m : Message(status == Message.HELLO, message : message)
    then
        System.out.println( message );
        m.setMessage( "Goodbye cruel world" );
        m.setStatus( Message.GOODBYE );
        modify( m );
end
```

The part between *when* and *then* keywords, is known as LHS (Left Hand Side) or the condition part and the fragment following *then* is known as RHS (Right Hand Side) or the action part of a rule.

- **The parser** has as input the rule file and as output the AST objects describing the rule.For the rule above we have the following objects(red in the photo):

RuleDescr					
lhs	AndDescr				
	descr[0]	ColumnDescr			
		descrs[0]	LiteralDescr	status == Message.HELLO	
		descrs[1]	FieldBindingDescr	message : message	

consequence	System.out.println(message);
	m.setMessage("Goodbye cruel world");
	m.setStatus(Message.GOODBYE);
	modify(m);				

Figure 3.2: AST objects which form a rule

- **PackageBuilder** uses the AST objects in order to produce a self contained object, the Package which consists of one or more rules. A Package Builder can take over parsing responsibilities, without being compulsory to do so, while code generation and compilation are mandatory steps for building the Package.

- **The RuleBase** is the runtime component which aggregates one or more Packages.

- **The WorkingMemory** is the most important class for the rule engine at runtime. It holds references to asserted facts (objects), it takes care to 'move' matched rules on the *Agenda* and finally, rules execution it is also, the responsibility of the same class.

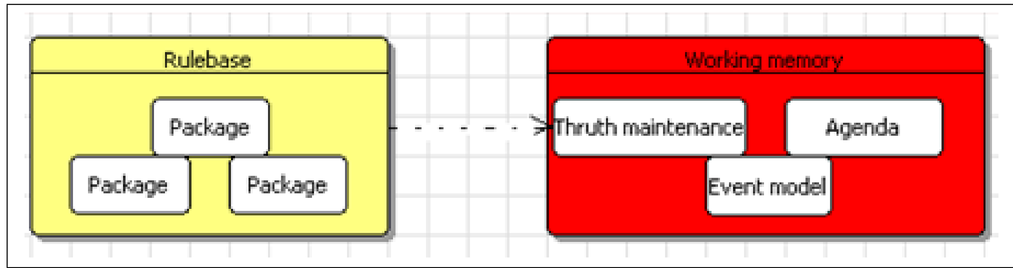

Figure 3.3: Drools runtime

JBoss Rules helps us at debug time with three useful views: working memory view, agenda view and audit view (not dependent on debug mode)

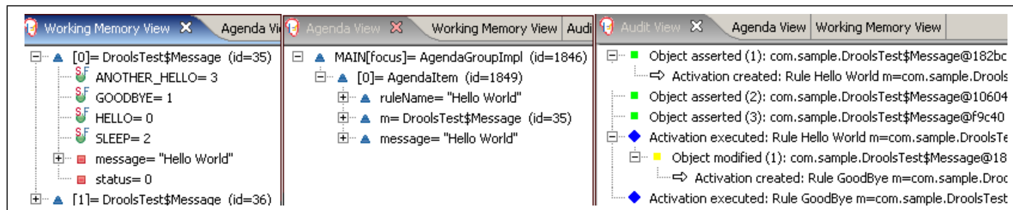

Figure 3.4: Working Memory view, Agenda view, Audit view

- **An event package** provides a notification mechanism for rule engines related events (e.g object assertion, object retraction).
- **Truth maintenance** is related to logical assertion of facts. When logically assert an object, the retraction takes place automatically, when the condition used for assertion is no longer valid. A regular fact assertion does not fall into the scope of truth maintenance component.

3.1.1 Domain specific language

An interesting feature offered by Drools for rule authoring are *DSL* (Domain Specific Language) files. *DSL* is in fact located at the intersection of problem domain and solution domain (related to domain model objects and the role played by those in rules world). Both non- technical and rule engine specialists can contribute to rule authoring, as DSL allows a natural like language expression.

Let's see how a rule having a DSL in background, is defined(fig. 3.5):

The attention should be directed first, to the line including the text: *expander Sample.dsl* , because this tells the drl file, where to find all the expressions used in LHS and RHS of the rule.

The emphasized expression, part of the rule's action is defined in Sample.dsl like:

```
[then]Add book into the folder corresponding to the category=f.getBooks().add(b);
```

Three points are to be noticed:

```
#created on: 13-Jun-2006
package org.pe
expander Sample.dsl

import myLib.Library;
import myLib.Folder;
import myLib.Book;
import myLib.MyLibFactory;

rule "Add book to existiong folder"
    when
        There is a book which has the category folder already in the library
    then
        Add book into the folder coresponding to the category
        Log categorization of book
        Remove already categorized book from working memory
end
```

Figure 3.5: Rule authoring with DSL

- *Then* defines the scope of the expression
- Problem domain part of the expression is: *Add book into the folder corresponding to the category*
- Solution domain part of the expression is: *f.getBooks().add(b)*

As can be easily noticed the parts of a DSL expression are separated by '=' character.

Although not difficult to create by using a text editor, the user has the support of an integrated DSL editor, and also auto complete on DRL side.

3.2 Example EMF model working with Drools

3.2.1 Problem statement

A library holds books which can be reorganized into folders, based on various criteria. Currently, a book has only a title and a category description. The books will be reorganized based on their category(e.g a folder for French literature, a folder for astronomy), but if we enrich book attributes, the categorization criteria could become more complex. Reorganizing books based on publisher, category description and year, might require that a folder can include other folders, which is not possible given the defined model, but otherwise would not impose further problems.

Listing 3.2: Emfatic view of the library model

```
package myLib;

class Library {
  val Folder[*] folders;
  val Book[*] books;
}
```

```
class Folder {
  attr String name;
  val Book[*] books;
}

class Book {
  attr String title;
  attr String categoryDescr;
}
```

We assume the classification criterion will change often. As an effect of the change frequency, we will keep the organizational issues out of EMF, by assigning the task to a rule engine. What we have to do, is to take care that a rule engine knows all the objects to operate with, as well as the rules to be used in order to classify correctly the available books.

3.2.2 Rules defined

Two rules have been defined for classification purposes.

1. Add a book to an existing folder
   When trying to classify a book, the existing folder names are inspected. If at least one matches the category of the book than the book is added to that folder. After a book has been categorized, we will retract the book from the working

```
rule "Add book to existiong folder"
    when
            f : Folder ()
            b : Book()
            eval (myfunc(b,f))
    then
            System.out.println("Book: "+ b.getTitle()+
                               " added to EXISTING folder: "+
                                b.getCategory());
            f.getBooks().add(b);
            retract(b);
end

// is books's category the same as folder's name?
function boolean myfunc(Book b, Folder f)
{
  return f.getName().equals(b.getCategory());
}
```

Figure 3.6: Add a book to existing folder rule

memory, as there are no more rules needed to process a categorized book.

2. Create folder
   The category, the book is in, is not present in the model, so the new folder (corresponding to the category description) is going to be created and the book added to it. The action part of this rule uses both 'modify' and 'assert' clauses. After a folder has been created, we need to inform the working memory about it, as there might be other books (in the future), suitable for inclusion in the new folder. This rule is only responsible for creating a new folder and asserting it. The current book will be categorized with the help of the first introduced rule. To signalize that we want the processing to continue, we inform the working memory by using 'modify'.

```
rule "Create folder"
    when
      l : Library()
      b : Book()
    then
        Folder fold=MyLibFactory.eINSTANCE.createFolder();
        fold.setName(b.getCategory());
        l.getFolders().add(fold);
        System.out.println("Book: "+ b.getTitle()+
                           " added to a NEW folder: "+
                           b.getCategory());
        assert(fold);
        modify(b);
end
```

Figure 3.7: Create folder rule

3.2.3 How does EMF tackle the rules defined?

Going into the EMF code side, what we have to do is:

- When a model is loaded, add all the folders and all the books to the working memory. 'All the books' which are children of the library only, or in other words :only the books which haven't been classified yet(embed code at editor level)
- Add the library into the working memory when the library is created using the wizard (embed code at editor level)
- When a book is added or deleted this should be marked in the working memory by asserting or retracting the corresponding facts (at edit level)

- When a book's related data is modified this should be also signalized to Drools (at edit level)
- Fire all rules when the user chooses reorganize (at editor level)

As we can deduce, we need a common access point for the working memory, as the code needing access to it resides in both, the edit and editor plugin. A new wrapper class (on *org.drools.WorkingMemory*) working as a singleton is defined as a part of the edit plugin, so that the editor can gain access to it too.

As an example, we will reproduce here two important code snippets.

Assertion, retraction of objects from the working memory are embedded into overwritten commands:

Listing 3.3: BookItemProvider.java-override *SetCommand*

```
protected Command createSetCommand(EditingDomain domain, EObject
owner, EStructuralFeature feature, Object value, int index)
{
        try {
            DroolsInitializator dInit=DroolsInitializator.getInstance();
            dInit.modifyBook((Book)owner);
        } catch (Exception e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return super.createSetCommand(domain, owner, feature, value);
}
```

The above *SetCommand* updates a book object already resilient in the working memory.

In the beginning, when the model is loaded the assertion of facts, looks like:

Listing 3.4: Assertion of facts

```
DroolsInitializator dInit=DroolsInitializator.getInstance();
dInit.cleanWorkingMemory();
dInit.assertLibrary(lib);
for (Iterator it=lib.getBooks().iterator();it.hasNext();)
{
   dInit.assertBook((Book)it.next());
}
for (Iterator it=lib.getFolders().iterator();it.hasNext();)
{
   dInit.assertFolder((Folder)it.next());
}
```

Another way to represent rules, besides drl and dsl files, is to have them embedded in spreadsheet format, as decision tables. The need for having this feature raised as many companies have their data in excel files.Decision tables are not useful only in a

'business' context, but also when we can define a certain template of a rule, which is supposed to take its data from the same spreadsheet document.

For an example, using Drools decision tables, please refer to appendix B.

Chapter 4

Behavior achieved through statemachines

Summary. *By reading this chapter the reader will find out about three different paths taken, to define models and generate code for statecharts implementation: switch based approach, state pattern and object oriented approach based on a predefined statecharts metamodel.*

4.1 Define models statecharts generation

Given that we want to generate code for statecharts implementation, having in mind the structure of the desired generated code, is a must.

For code generation,we will rely on *EMF.Codegen* capabilities, our focus being to achieve a declarative representation of the reactive behavior of a system (a representation at the model level).

4.1.1 Switch based approach

The first step towards defining state machines, involves the specification of a model which follows a classical, procedural approach. The model for representing behavior in this case, is self-contained. *States* are specified as part of an enumeration and the events are methods of the reactive object. *Transitions* between states are taken with the help of switch statements. Functional aspects can be defined directly at meta-model level, due to *GenModel* annotations in *Ecore*.

A model defined this way has the advantage of being simple and compact, and the drawback of being hardcoded. Topology changes within the same state machine or definition of a new state machine for another context class reveal maintenance problems for such an approach.

4.1.2 State pattern approach

An improvement of the switch approach would be the use of *State design pattern*. Its intent is to: "allow an object to alter its behavior when its internal state changes. The object will appear to change its class"[1]. The problem statement, for the state pattern, maps to our first approach: "A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application"[1].

The model is more flexible because new state classes can be easily added to the model (in accordance with open-close principle of software development), but there is a coupling between particular state classes for implementing the transitions.

4.1.3 Object oriented approach based on a predefined statecharts metamodel

Another object oriented path to take, allows customization of the state machine at run-time. It brings the advantage of *separation of concerns*, in the sense that a static entity becomes reactive, only after a state machine has been attached to it. This object oriented approach enables reuse of statechart metamodel (SMM), as well as the generated code. Having a SMM, forms a basis for defining a graphical editor for statecharts. Object oriented style of defining reactive systems pays off as well, in respect to amount of code generated for slightly bigger systems, comprising more than 3-4 classes.

Note: If we accept the notion of *reactive entity* as an atomic behavioral unit, then *attaching a state machine to an object* does not make sense. '*A static entity, become reactive, only after a state machine has been attached to it*' wants to suggest that a reactive object aggregates a state machine, for which a meta-model was defined previously.

We are dealing now with objects, and their structural relationship is depicted in the diagram(figure 4.1) ma ti The statechart model we have defined lacks completeness, but due to EMF model driven features, further enrichments at meta-model level would not be problematic, as we just use (by referencing) and not embed the SMM into the context object meta-model definition; plus we rely on EMF code generation facility.

How does a state machine work?

Note: this fragment contains implementation details as well

We will study a behavioral state machine, not a protocol state machine, in the sense UML2 specification makes this distinction.The focus will not be on specifying the

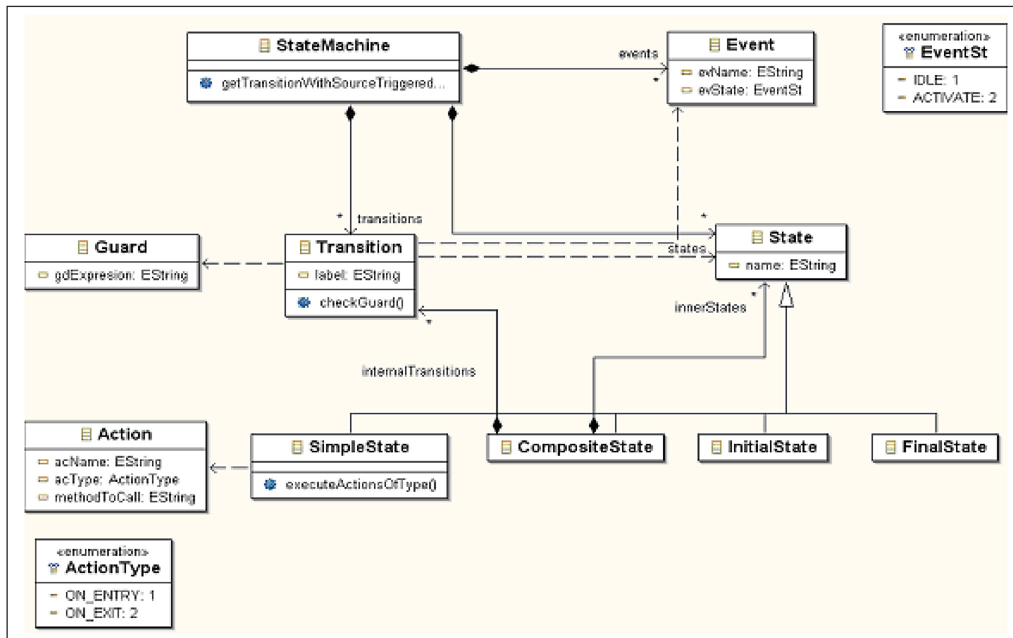

Figure 4.1: State charts metamodel(SMM)

right sequence of events an object responds to (event protocol), but instead on the behavioral aspect: specification of states and transitions, on-entry actions, on-exit actions, etc.

The functionally will be described with focus on what has already been defined.

A reactive entity has a current state to be in. The current state is updated based on state machine 'internals'. A *state* can be a *simple state* (has no sub-states), a *composite state* (has nested states) and *pseudo-states* (like final state, initial state, history states).

Note: The distinction between different pseudo states makes no sense by having them as separate classes in the model (at least not for the initial and final states). This was useful for achieving fast, a functional goal, when developing the statechart graphical editor.

The current state of a reactive object is updated based on transitions taken. A *transition* is described by the construct: *event [guard]/action*.

When an event, which triggers a transition, is fired, the condition hosted in the guard is first checked, and if it evaluates to false, the action specified as part of the transition is not performed, and the current state won't be changed.

An event is raised when its state attribute is set to ACTIVE. An event is active for the duration of one step. A step would mean a change in the current state of the context object, in case that the guard of the transition taken allows it, otherwise if the guard is false the event will be again set to IDLE.

Given 2 simple states **A** (A is the current state) and **B**,plus a transition described by *E[G]/act*, the response of the system when the event **E** is raised, is:

- the guard is checked (further steps are taken if the result is true)
- the exit actions from state **A** are performed
- action '*act*' and the entry actions in state **B** are performed
- the current state of the context object is set **B**

More elements connected to state machines should be added: *history states* for representing the most recent active configuration of a composite state (with history support), *regions* as parts of composite state, *null* transitions (not being triggered by events), *timeout events* (raised after a certain amount of time, a state was entered). No further distinction between different types of events: synchronous or asynchronous are going to be made.

4.2 Case study on reactive models

For the ease of reference the statechart of a microwaveoven is reproduced below:

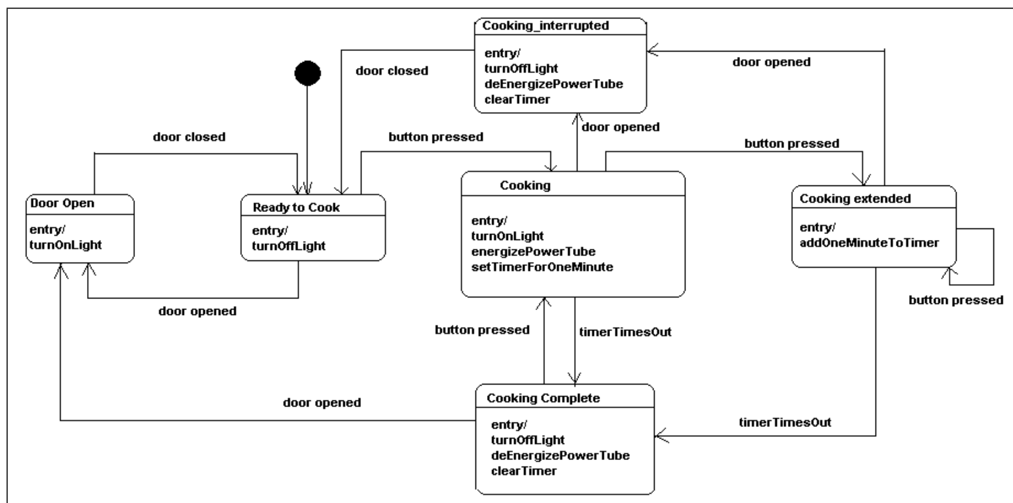

Figure 4.2: Microwaveoven statemachine

4.2.1 Switch based approach

For the switch modeling approach the ecore metamodel, as visualized in Omondo, looks like depicted in figure 4.3.

Methods named *onEntryXYZ* are just for grouping in a single function, the other method calls, specified in the state diagram.

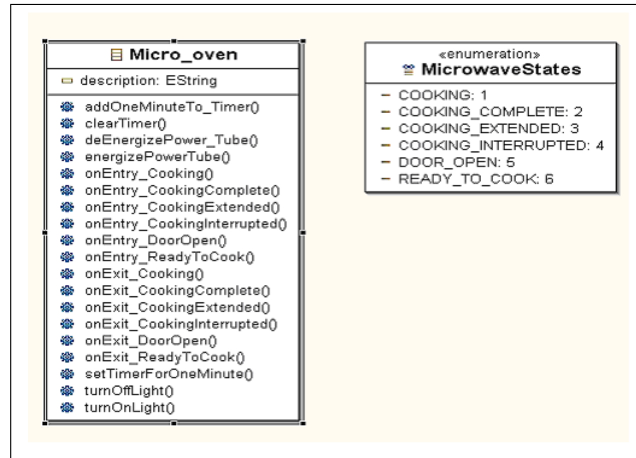

Figure 4.3: Model definition for switch based approach

Listing 4.1: onEntry method for Cooking state

```
public void onEntry_Cooking()
{
    turnOnLight();
    energizePowerTube();
    setTimerForOneMinute();
}
```

The code for events are defined as part of the meta-model making use of annotations:

Listing 4.2: GenModel annotations

```
...
@GenModel(body="switch (currentState.getValue())
{
case MicrowaveStates.READY_TO_COOK:
{
        this.onEntry_DoorOpen();
        currentState = MicrowaveStates.DOOR_OPEN_LITERAL;
        break;
}

case MicrowaveStates.COOKING_INTERRUPTED:
{
        onEntry_ReadyToCook();
        currentState = MicrowaveStates.READY_TO_COOK_LITERAL;
        break;
}
}")
op void doorClosed();
...
```

4.2.2 State pattern approach

Although not a burden to define, an ecore model following state pattern[1], requires knowledge of the mentioned design pattern. The user can be relieved from having the knowledge by redefining the model in a more compact and comprehensive manner, and leave all the details for generation in accordance with the afore mention pattern, to the code generation engine. The benefits of having a new, compact model defined, as the input of the state pattern generation process, will be emphasized in section 4.3.

4.2.3 Object oriented approach based on a predefined statecharts metamodel

Statecharts metamodel(fig. 4.1) has to be linked to the definition of the context class (the microwave class), by making use of the import clause.

Actually, the 'object oriented features' are conveyed by the SMM in this case. The *statecharts.ecore* file resides inside a deployed plugin. The *currentState* a microwave finds itself in, is referenced in the context class, this being part of the *stMachine*, as can be seen in figure 4.4.

```
package mw;

import "platform:/plugin/de.tuhh.sts.gmfexample/model/statecharts.ecore";

class Micro_oven
{
  attr String description = "Super Microwave";
  ref statecharts.State currentState;
  val statecharts.StateMachine stMachine;

  @GenModel(body="System.out.println(\"onEntry_DoorOpen\");
                  turnOnLight();")
  op void onEntry_DoorOpen();

    @GenModel(body="System.out.println(\"\tonExit_DoorOpen()\");")
  op void onExit_DoorOpen();

  @GenModel(body="System.out.println(\"onEntry_CookingInterrupted\");
                   turnOffLight();
                   doEnergizePower_Tube();
                   clearTimer();")
  op void onEntry_CookingInterrupted();
```

Figure 4.4: Reactive object based on SMM

Omondo does not help us too much in visualizing the meta-model this time (fig. 4.5)

Note: Making use of priori defined meta-models can also be noticed when defining the ecore of OCL. A Constraint (class member of OCL ecore) is defined as a specialization of NamedElement(class member of ecore's metamodel)

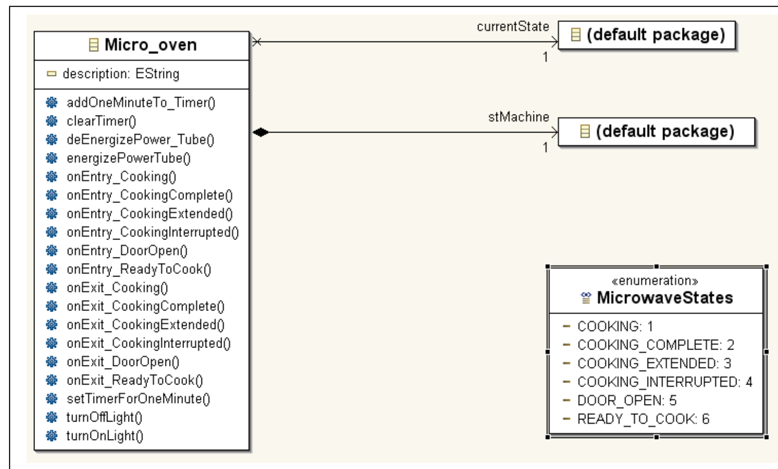

Figure 4.5: Omondo view of the OO model

Not only objects and attributes have been specified in the SMM, but also some additional methods like constraint checking or executing actions of a certain type.

The method responsible for guard evaluation is defined as part of *Transition* class, as follows:

Listing 4.3: onEntry method for Cooking state

```
@GenModel(body=" Query query=QueryFactory.eINSTANCE.createQuery(
                  getGuard().getGdExpresion(), ctxtObj.eClass());
               return query.check(ctxtObj);")
op boolean(EObject ctxtObj);
```

The guard expression is a String, representing an OCL expression. For OCL expression evaluation we rely on EMFT-OCL[1] (for this case the package *org.eclipse.emf.ocl* was needed)

Execution of actions rely on method invocation, therefore the processing of actions is defined by *GenModel* body as:

Listing 4.4: Actions execution making use of reflection

```
@GenModel(body="EList actions=getInternalActions();
               try{
                for(Iterator<Action> it=actions.iterator();it.hasNext();)
                {
                  Action cAction=it.next();
                  if (cAction.getAcType().equals(actType))
                  {
                     Class c=ctxtObj.getClass();
                     Method met= c.getMethod(cAction.getMethodToCall(),null);
                     met.invoke(ctxtObj, null);
                  }
```

[1] http://www.eclipse.org/emft/projects/ocl/

```
        }
    } catch (Exception ex){}")

op void executeActionsOfType(EObject ctxtObj, ActionType actType);
```

ctxtObj will be in this case the microwaveoven, which has all the functional methods defined.

Validation issues related with the statechart metamodel were not defined in the context of the EMF model, but they were approached, when the graphical editor for statecharts was partially developed. The help of *EMF Validation framework*[2] was essential here.

Regarding transactional aspects, the need for transactions pops up when we deal with actions. On-entry and on-exit are supposed to be atomic units of behavior, but do-activities are not (do-activities are possible to be interrupted by events). The process followed for the execution of one step, should also be performed in terms of 'all or nothing' principle (e.g. if on entry action for the new state throws exceptions the effects of on-exit actions performed in the context of the current state should be reversed). EMF-Technology provides transactional support[3] which can be used to fulfill the above mentioned goal.

4.3 State pattern generation based on a compact model

Ultimately, the users of Eclipse Modeling Framework beneficiate of full Model Driven Architectural features, due to code generation facility included in EMF. EMF code generator is template based. The Java Emitter Template engine, part of *org.eclipse.emf.codegen* plug-in is responsible for generating all the necessary files, in a two step process (following pipes and filters architecture): translation and generation. As a result of translation, a java source file corresponding to each JET template is created. The compiled 'translated' java file from the previous step is the input for the generation step. The result of the generation can be 'any kind of text content' as Dave Steinberg, one of the architects of EMF declares. In practice, that can be easily noticed after running the generation process for one of EMF's main artefacts: model, edit or editor. The result of applying code generation consists, not only in java source files, but also xml files (e.g the deployment descriptor) and text files (e.g plugin.properties), in other words, it generates everything needed to create full fledged plug-ins. An interesting point which should draw our attention, due to its recent dynamic, is EMF's generator extensibility. The changes are made available starting with 2.2.0RC2 stable build, dated at 2 May 2006. Achievement of extensibility can be

[2]EMFT Validation Framework http://www.eclipse.org/emft/projects/validation/
[3]EMFT Transaction component http://www.eclipse.org/emft/projects/transaction/

summarized by: making use of extension point architecture and relieving the generator model from the actual generation responsibilities.

GenModel is actually a decorator of the *Ecore* model. The ecore model is the correspondent of platform independent model (PIM) in MDA terminology, while the generator model is the platform specific model (PSM).

The GenModel contains additional information needed for the generation as: the structure of the plug-ins(e.g which directories are going to be created for holding generated artifacts, which external templates are going to be used, reflective methods are to be generated or not, etc.). Each *Gen*-object hold a reference to the corresponding *E*-object (e.g *GenPackage* to *EPackage*) plus additional code information regarding code generation (e.g is *AdapterFactory* going to be generated for the model?).

Before EMF's 2.2.0RC2 release, *GenModel* objects were responsible for generating their own code, which obviously led to the undesirable feature of having the generation facility tightly coupled with the *GenModel*. An effect of the afore mentioned 'feature' is the impossibility of generating code for another Ecore model.

The problem was solved by introducing a *Generator* class, responsible for code generation. In fact, what the *Generator* class does is to delegate generation to the attached generator adapters. In this manner the generator model is decoupled from the actual generation burden.

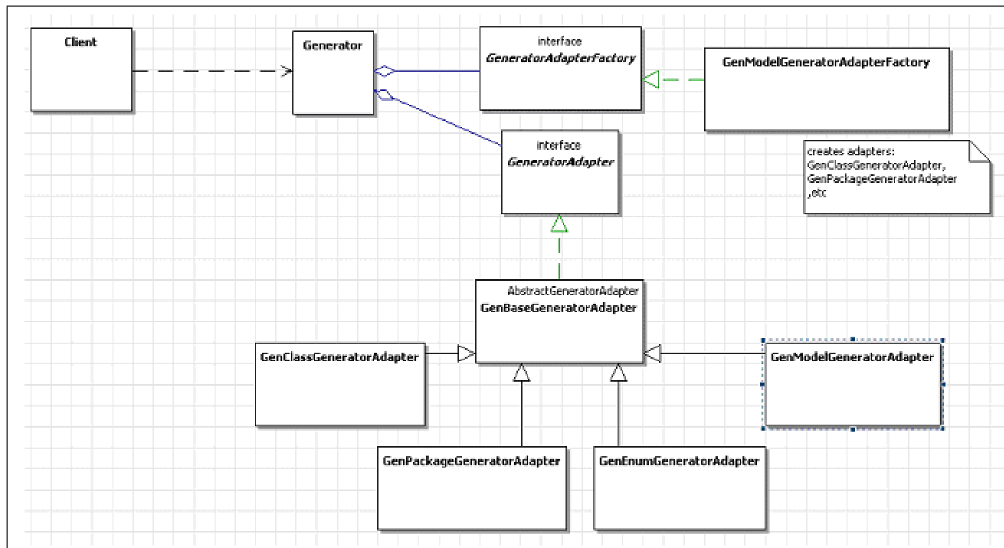

Figure 4.6: EMF.CodeGen structure

The structure of the packages in a recent release has two new packages: *org.eclipse.emf.codegen.ecore.generator* containing the generator and abstract adapter

classes used in
org.eclipse.emf.codegen.ecore.genmodel.generator, where the specific adapters for each *Gen*-object reside.

EMF generator is said to be extensible because the plug-in declares an extension-point for generator adapters.

Listing 4.5: Extension point declaration

```
<extension-point id="generatorAdapters"
                 name="%_UI_GeneratorAdapters_extensionpoint"
                 schema="schema/generatorAdapters.exsd" />
```

This plug-in plays the role of a host plug-in and acts as a coordinator and controller for other extensions.

Listing 4.6: Extension point declaration

```
<extension point="org.eclipse.emf.codegen.ecore.generatorAdapters">
 <adapterFactory class="org.eclipse.emf.codegen.ecore.genmodel.generator.GenMode
 <adapter
       modelPackage="http://www.eclipse.org/emf/2002/GenModel"
       modelClass="GenClass"
       class="org.eclipse.emf.codegen.ecore.genmodel.generator.statemodel.GenClass
</extension>
```

In order to emphasize the problems encountered and the proposed solution, we will follow an example showing a state machine for a carousel door.

4.3.1 Case study.Problem.Solution

An ecore model, following state pattern was defined for a carousel door state machine.

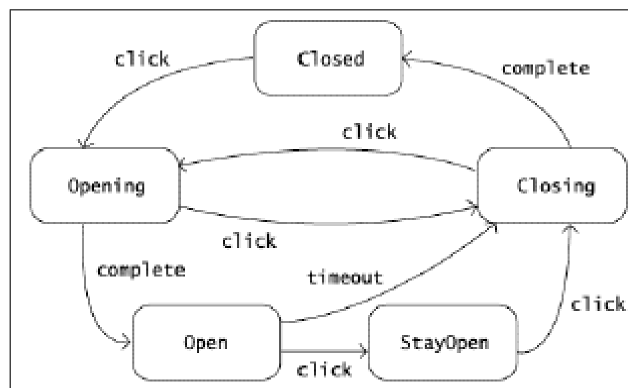

Figure 4.7: Carousel door statemachine

Applying code generation facility at model level, we have a package structure as in figure 4.9:

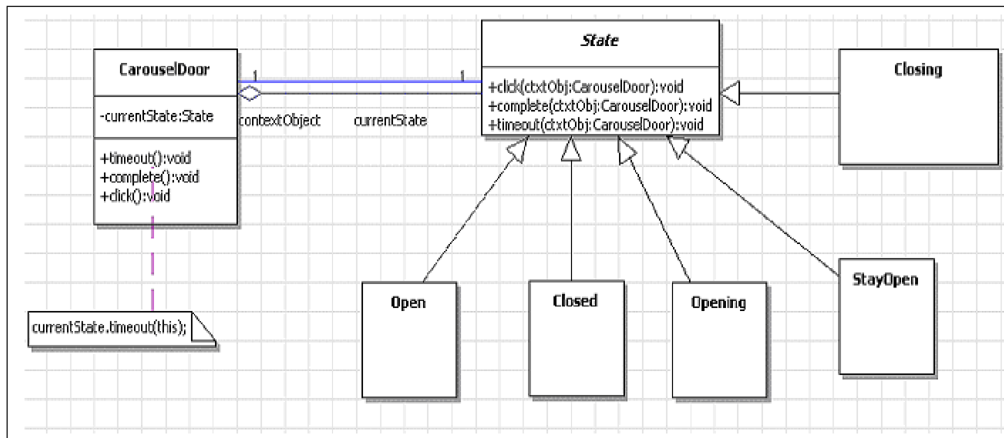

Figure 4.8: Statemachine modeling with the help of state pattern

EMF generates for each entity class in the model, two elements: an interface (residing in *states* package) and an implementation class (residing in *states.impl*). Given that all the states inherit from an abstract class, the generation of an interface for each particular state is not justified, so as an effect, that will be removed when the actual state pattern generation is to be performed.

State *Open* will react to click and timeout events, according to the model defined, but the generated code fails (by not-generating) in embedding click and timeout methods in the implementation class. Obviously, methods body declared with *@GenModel* annotation will not have the desired effect in this case.

Specifications of events handled as well as methods body will be further tackled using annotations.

After removing the interface generation for all particular states and canceling the generation for the adapter factory (which would reside in *states.util* package), the structure of the plugin should look like in figure 4.10

Apparently an 'inoffensive' modification triggers changes in *StatesFactory.java* and *StatesFactoryImpl.java* for each particular state creation: e.g

Listing 4.7: Three example transformations

```
Open createOpen();
//should be transformed to
State createOpen();

/***********************************/

public Open createOpen() {
 OpenImpl open=new OpenImpl();
 return open;
}
// should be transformed to
public State createOpen() {
```

```
 OpenImpl open=new OpenImpl();
 return open;
}

/************************************/

public class OpenImpl extends StateImpl implements Open
// should be transformed to
public class OpenImpl extends StateImpl
```

For dealing with those kind of changes a new template for *FactoryClass.javajet* has to be defined. In order to cope with the changes in *StatesPackageImpl.java*, a new *PackageClass.javajet* will to be defined.

Given that for particular states only the implementation class is generated (the interface for *State* class has to be defined because is needed in the creational methods of the factory), a new template *Class.javajet* will be pluged into the generator.

We already mentioned that defining the *ecore* model in accordance with the diagram above, has two inconveniences, namely: the user has to know about the state pattern and the generated code is not well structured, lacking method bodies which cannot be defined as part of the *ecore* model.

A more convenient manner to specify the model would be to encapsulate all the states the context object can be in, as well as the events triggering transition among these states, in a single entity.

Listing 4.8: The new model defined

```
interface CDoor
{
    attr int Open=1;
    attr int Opening=2;
    attr int Closed=3;
    attr int Closing=4;
    attr int Stayopen=5;

    @GenModel(Open="int c=3;int m=45;",
              Opening="",
              Closed="",
              Closing="",
              Stayopen="")
    op void click();

    @GenModel(Opening="",
               Closing="")
    op void complete();

    @GenModel(Open="")
    op void timeout();
}
```

All the states are defined as attributes. The events are defined as methods. If a transition triggered by a certain event is valid from a particular state, the method body can be specified in a *@GenModel* annotation having as a key the name of the state(e.g 'complete' can be taken only if the current state of the state machine is one of :'Opening' or 'Closing' states)

We just mentioned that states are defined as attributes. Actually, the only important issues for the generation is the name of the attribute (or in the case of operations only the operation's name), not the type nor the initial value (missing or not). For this reason, model validation issues would also be a plus, assuring the integrity of the defined model.

Another way of defining the ecore model would be to have all the states defined as a part of an enumeration and make use of custom annotations to specify valid transitions.

Having the model defined as described above, the information will be extracted from the model and the gen-model for the actual generation constructed (the classes used for modeling can be inspected below). The path taken is problematic because it alters the initial generated model, making further generations (of model, edit, editor) inconsistent if the model is not brought to the initial state. This approach has the advantage of keeping changes in JET files minimal.

As we mentioned in the extensibility strategy for the generator, what we still have to do is to add the needed generator adapters.

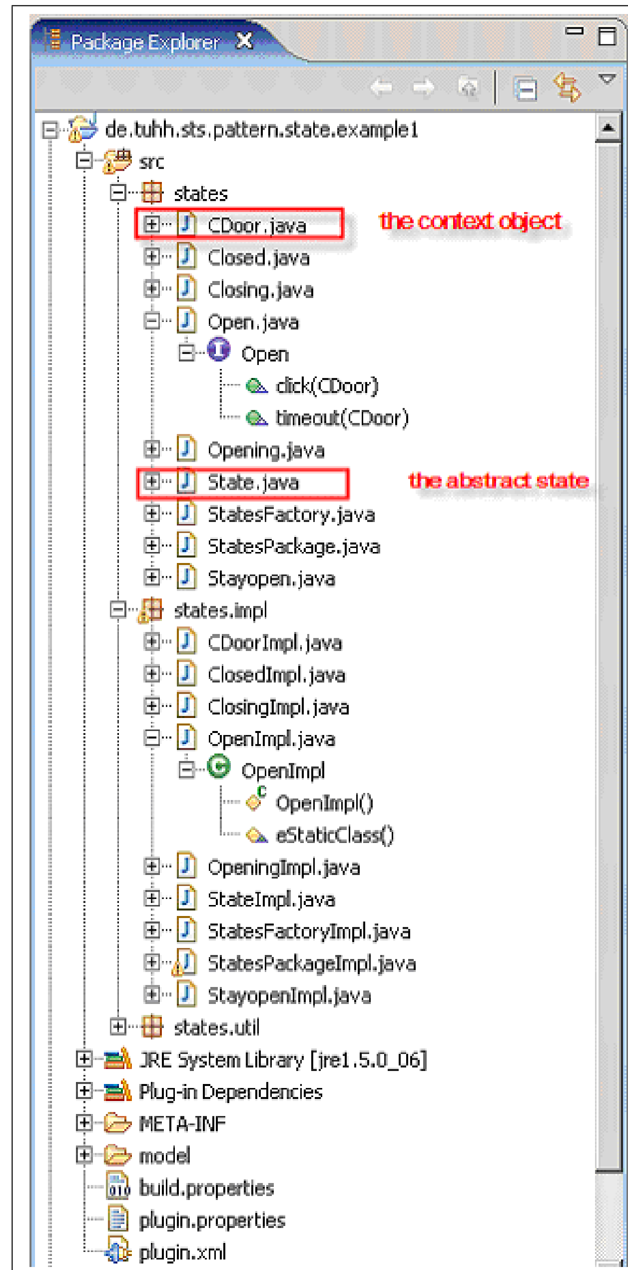

Figure 4.9: Package structure after code generation

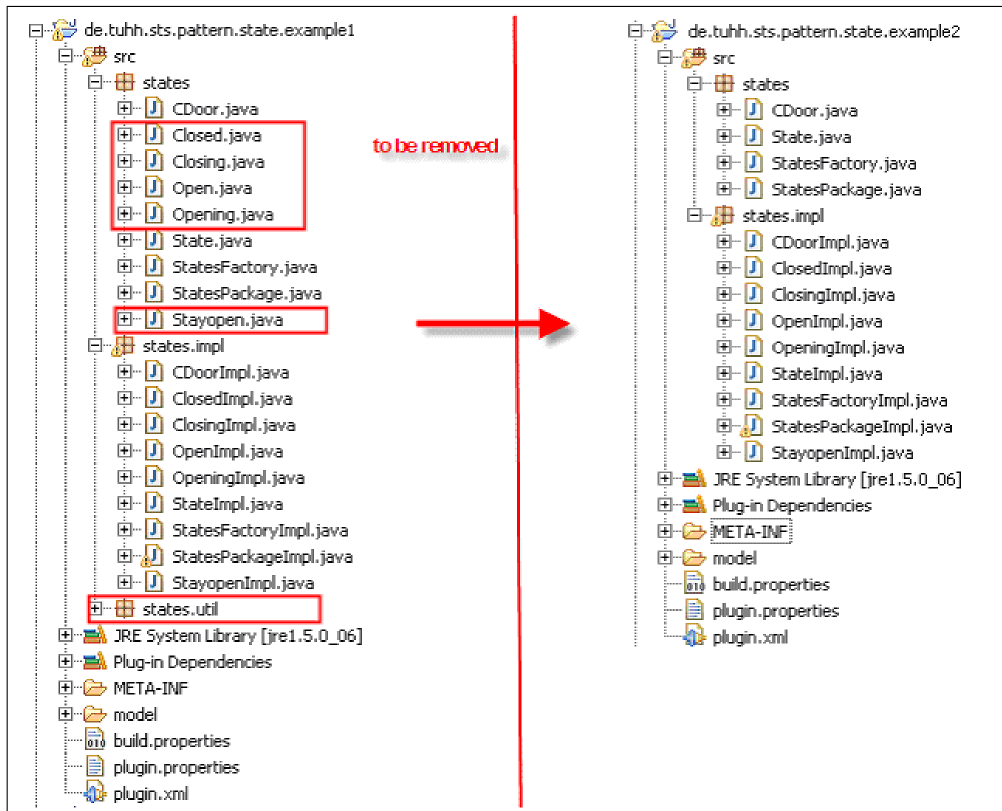

Figure 4.10: What should be taken out in order to enhance the generation

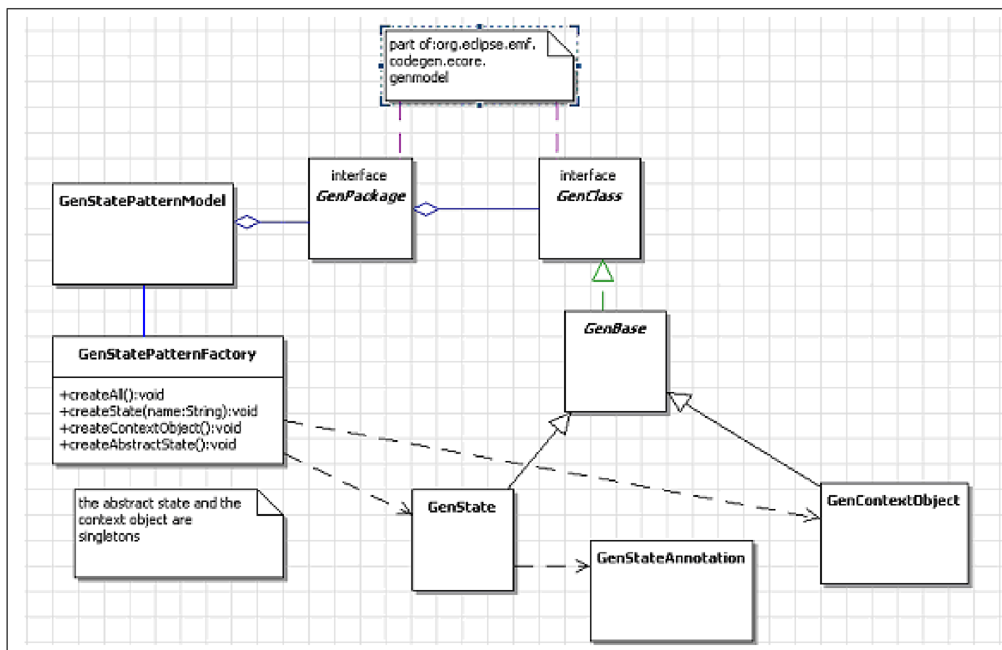

Figure 4.11: Class diagram used for SP generation

Chapter 5

Conclusions

Eclipse Modeling Framework's model driven features ease the path of translating a defined model into the corresponding code. Relying on EMF's code generation facility, we focused on conducting inquiries into behavioral aspects gravitating around EMF models.

Attaching behavior, by making it external to EMF's generated code, can be achieved when using rule engines[9]. In this case, the responsibility on EMF side is brought down to making facts known to the rule engine.

State machines are a standardized way to capture behavior, therefore defining models for generating statecharts specific code and relying further on EMF.Codegen for generating code is an interesting issue, approached in [3] for plain java models.

Formal representation of behavior as statecharts can be used to generate user interfaces[6] or to define device simulators in Flash MX[7]

The same meta-model used to describe statecharts was used for defining a graphical editor for statemachines and for specifying reactive entities. But unifying the both in order to offer the user the facility of visualizing a reactive object at execution time, could be planned as future work.

OpenArchitectureWare[8] presents a solution for generation of state machines following a switch approach and using POJO objects, while a state pattern generation out of a simpler model (requiring no knowledge of the mentioned pattern) has been performed here.

Appendix A

Actions and commands for multi-rooted resources in EMF

A.1 Problem statement

The default editor generated from an EMF model allows having only one root object. Unless the model is very simple and one type aggregates all the others, multi-rooted support at editor level becomes necessary.

Example:
When defining a statechart, one defines states. It makes sense to define the states in the context of a state machine. In this case, a statemachine is the natural top level container for states. But, a package may contain several state machines.

A fast way to circumvent the problem is to define a top level object, a container, for all the model entities that we would like to define. The only problem is that from a conceptual perspective, the artificial container might not make sense from a domain modeling perspective (what does it mean "TopLevelObject" when talking about LoyaltyAccounts, Services, ProgramPartners, and so on)

A.2 Solution suggested

The lack of support for multi rooted resources can be dealt with at the editor level, by providing us with the necessary pop up menu item *New child* at the resource level. We need the same menu item as the one in the context of a model element.

This observation guides us in tackling the problem. We now know the place where to include our extension: the editor. The EMF and EMF.Edit generated code need not be changed.

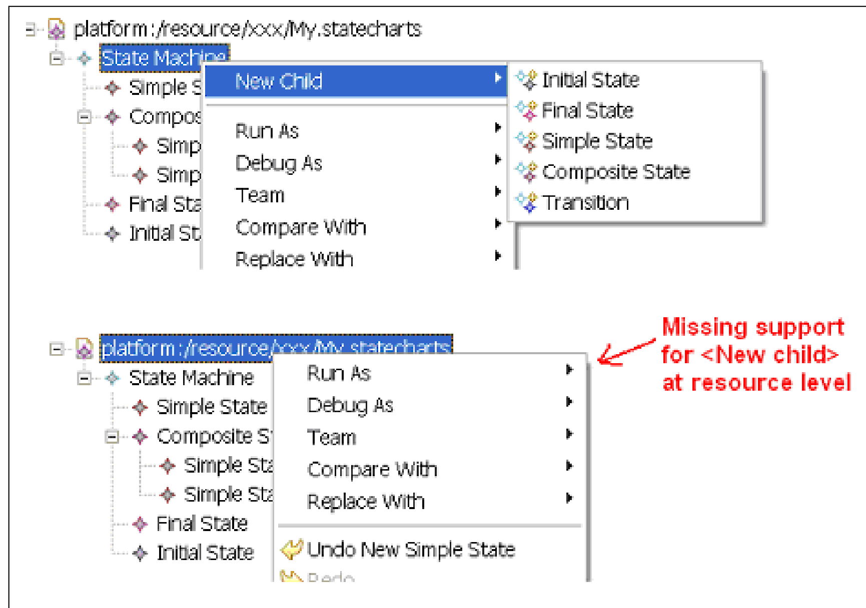

Figure A.1: The problem encountered

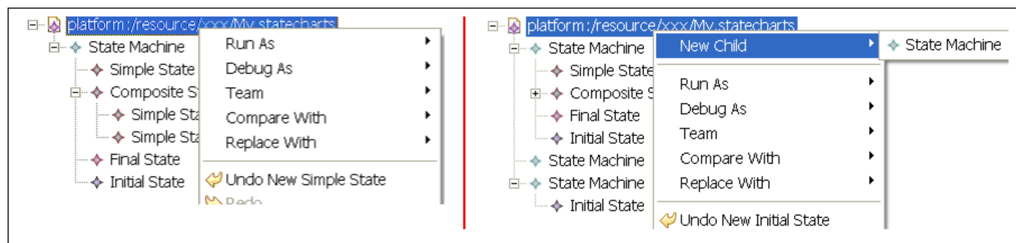

Figure A.2: The desired result

In order to contribute menu items to a popup menu, a class, derived from *EditingDomainActionBarContributor*, was automatically generated in the editor.

The editor plugin knows about the contributor class because of the extension point defined in the plugin's descriptor. The method named *generateCreateChildActions* is a good starting point.

Listing A.1: The method to start with

```
protected Collection generateCreateChildActions(Collection descriptors,
                                                ISelection selection)
{
    Collection actions = new ArrayList();
    if (descriptors != null)
    {
      for (Iterator i = descriptors.iterator(); i.hasNext(); )
      {
          actions.add(new CreateChildAction(activeEditorPart, selection, i.next()))
      }
```

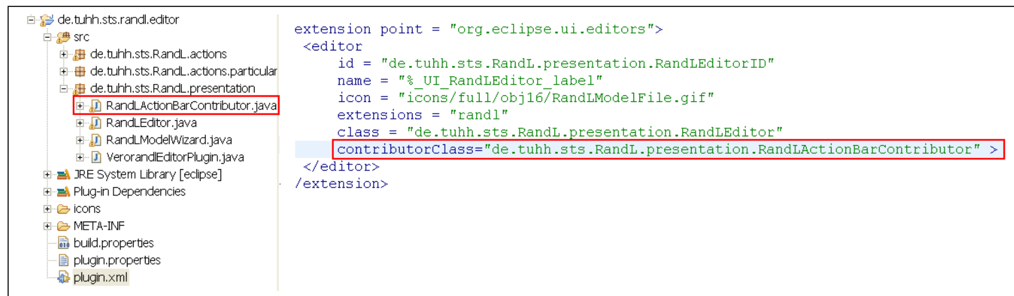

Figure A.3: Modifications required in the plugin descriptor

```
    }
    return actions;
}
```

The collection returned by this method will contain one *CreateChildAction* corresponding to each descriptor generated for the current selection by the item provider

This is the actual hint for the solution: define another action bar contributor class by extending the existing class (e.g *de.tuhh.sts.RandL.presentation.RandLActionBarContributor*) and override the above mentioned method making sure that *CreateChildAction* are being added when the selected item is the resource.

Listing A.2: Redifined method in the contributor class

```
protected Collection generateCreateChildActions(Collection descriptors,
                                                ISelection selection)
{
  Collection actions = new ArrayList(
           super.generateCreateChildActions(descriptors, selection));
   if ((selection instanceof IStructuredSelection) &&
       ((IStructuredSelection)selection).size()==1)
  {
   Object object= ((IStructuredSelection)selection).getFirstElement();
   if (object instanceof Resource)
   {
       actions.add((Resource)object);
   }
  }
   return actions;
}

public Action generateResourceAction(Resource resource)
{
   return new LoyaltyProgramAction(activeEditorPart,
                                   new StructuredSelection(resource));
}
```

As can be seen from the redefined method, a new action was added to the collection. The action is of type *LoyaltyProgramAction* and was defined to inherit from

StaticSelectionCommandAction. The *StaticSelectionCommandAction* object (in our case the: *LoyaltyProgramAction*) delegates all the required behavior to a Command object.

The method *createActionCommand* of *RandLAddCommand* class, is responsible for creating the command:

Listing A.3: Redifined method in the contributor class

```
public class RandLAddCommand extends AddCommand
                              implements CommandActionDelegate
{
...
 public Command createActionCommand(EditingDomain inputEditingDomain,
                                    Collection collection)
 {
   return new RandLAddCommand(inputEditingDomain,
                              ((Resource)collection.toArray()[0]).getContents(),
                              RandLFactory.eINSTANCE.createLoyaltyProgram());
 }
...
}
```

So far, in order to add a new child on the resource root, three classes have to be declared: the action contributor, the action class, and the command class. We also

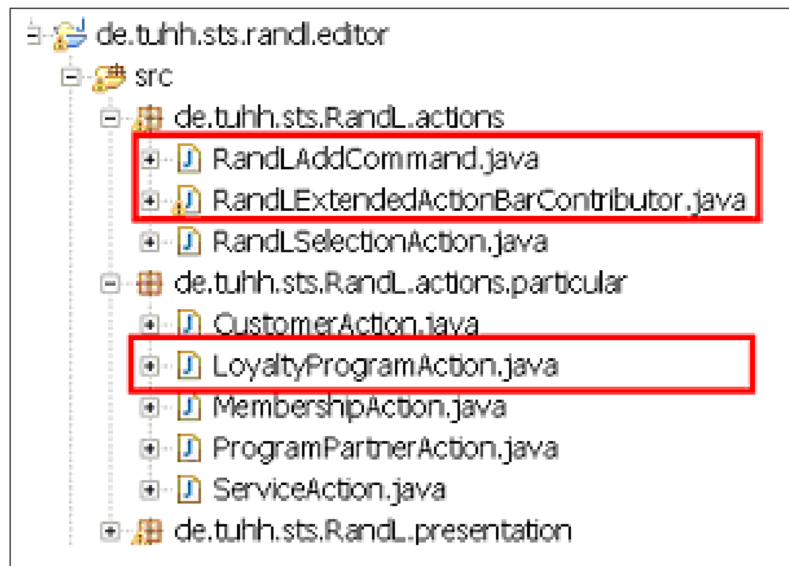

Figure A.4: XCommand, XActionBarContributor,XAction classes

have to make the plugin descriptor aware of the existence of a new action contributor, by modifying the attribute's value contributorClass, to point to the redefined contributor.

A.3 Problematic approach

The solution given, which conforms to EMF extension capabilities, is problematic when one wants to have a multi-rooted resource consisting of different types. The number of actions defined will increase with each different type of new child that we want to add to the resource root. Only one action (CreateStateMachineAction.java) is needed for adding multiple state machines to a resource.

An action is needed for each new type of child desired, as can be inferred from the above picture.

One way to circumvent the need to customize that many artifacts (one action class for each new child item at the resource level) would be to generate the code **automatically** from *genmodel*. The user should be given the option to choose, before generation takes place, the model entities that may appear as roots.

It should be possible to add as a root on a resource, each item being displayed in the drop-down list by the model wizard. Normally, only a limited number of model members are supposed to be added as roots (e.g it makes sense to have *CustomerAccount* only in a *Customer* context).

To conclude, as a tooling support for multi rooted editors, it makes sense for all these actions to be *automatically* generated based on user specifications in an extended genmodel model.

Appendix B

Decision table for a state machine definition

A decision table defining the transitions for the microwave state machine are defined inside an excel worksheet. LHS of a the rule template is specified in the CONDITION,

RuleSet	de.domainModel		
Import			
Sequential	true		
RuleTable MicroW Machine			
CONDITION	CONDITION	CONDITION	ACTION
source:State(name == "$param"); ctxtObj:ContextObject(currentState == source, target:currentState)	target:State(name == "$param")	ev:Event(event == $param)	takeAction(ctxtObj, target); modify(ctxtObj); retract (ev);
Source state	Target state	CurrentEvent	Transition description
ReadyToCook	DoorOpen	1	noDesc
CookingComplete	DoorOpen	1	noDesc
Cooking	CookingInterupted	1	noDesc
CookingExtended	CookingInterupted	1	noDesc
DoorOpen	ReadyToCook	2	noDesc
CookingInterupted	ReadyToCook	2	noDesc
ReadyToCook	Cooking	3	noDesc
Cooking	CookingComplete	4	noDesc
CookingComplete	Cooking	3	noDesc
CookingExtended		3	noDesc
Cooking	CookingExtended	3	noDesc
CookingExtended	CookingComplete	4	noDesc
Functions	function void takeAction(ContextObject ctxtObj, State target) {ctxtObj.getCurrentState().onExit(); ctxtObj.setCurrentState(target); ctxtObj.getCurrentState().onEntry(); }		

Figure B.1: Decision table for a state machine

while the ACTION contains the RHS part of the rule. Each row in the table results in a rule.This can be made visible to the user by printing out the results of the

RuleTable MicroW Machine			
CONDITION	CONDITION	CONDITION	ACTION
source:State(name == "$param"); ctxtObj:ContextObject(currentState == source, target:currentState)	target:State(name == "$param")	ev:Event(event == $param)	takeAction(ctxtObj, target); modify(ctxtObj); retract (ev);

Figure B.2: Template rule with parameters defined in the excel worksheet

SpreadSheetCompiler.The first rule, generated from the ninth row of the decision table, by filling in the parameters with the data taken from the xls file, look like:

Listing B.1: Rule 'instance' generated form the rule template

```
#From row number: 9
rule "r1"
 salience 65527
  when
      source:State(name == "ReadyToCook");
      ctxtObj:ContextObject(currentState == source,  target:currentState)
      target:State(name == "DoorOpen")
      ev:Event( event == 1)
  then
      takeAction(ctxtObj, target);
      modify(ctxtObj);
      retract (ev);
end
```

State,ContextObject,Event are objects from the defined domain model. The responsibilities of the test class are:

- Read the rules from the decision table with the help of the SpreadsheetCompiler

  Listing B.2: Use the appropriate compiler to load data from *.xls files

  ```
  ...
  InputStream src= new FileInputStream("states.xls" );
  SpreadsheetCompiler ssCmp=new SpreadsheetCompiler();
  String str=ssCmp.compile(src,InputType.XLS);
  ...
  ```

- Assert the states and the events into the working memory and fire the rules

Note: Only one event is present into the working memory at a time, the rule being responsible to retract the event from the memory in its action part

Appendix C

Statecharts editor based on GMF

Graphical Modeling Framework provides the infrastructure for defining graphical editors based on GEF(Graphical Editing Framework) targeting EMF models(Eclipse Modeling Framework).

Using GMF, a graphical editor for statecharts, which could be used to depict visual execution of statemachines, was defined.

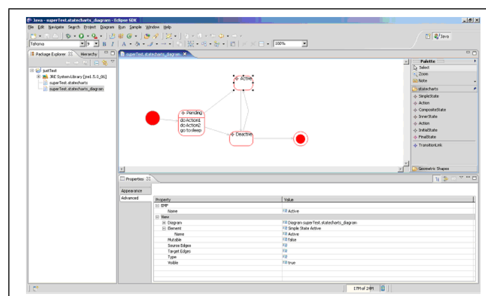

Figure C.1: Statecharts editor- simple state, initial, final state

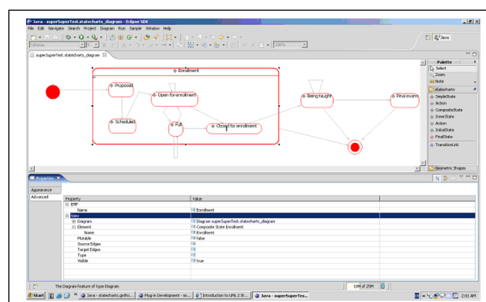

Figure C.2: Statecharts editor- composite states with inner states

C.1 Constraints

Two constraints were specified: one does not allow a transition to have as a source an instance of the final state, and the other one does not allow a transition to have as a target an instance of the initial state. Both constraints are attached to the link, one as a constraint on the source and the other as a constraint on the target of the link, as can be seen below.

Figure C.3: Defined link constraints

C.2 Validation

As validation rules for the model, two audit rules were added to the audit container. The first rule takes care that a model won't be validated unless it has one and only one initial state. The second rule, validates a model only if there are no multiple states having the same name.

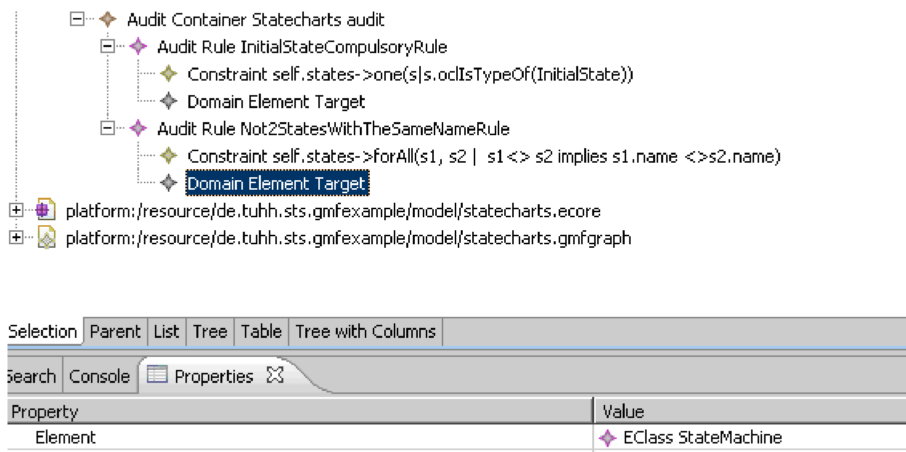

Figure C.4: Audit constraints as validation rules

C.3 Custom images

Composite figures are not yet smoothly supported by GMF. One has the option of adding Custom figures to the Figure gallery of the .gmfgraph file. Although a custom figure allows children (right click Add child) there is no mechanism yet to combine those figures into the resulting graphical component. As pointed out here http://dev.eclipse.org/newslists/news.eclipse.technology.gmf/msg01365.html one way of defining composite figures is too handcode them.

Taking the example of a FinalState figure for statecharts (a simple circle containing another concentric filled circle inside) the code written is:

Listing C.1: FinalState figure

```
public class FinalStateFigure extends Ellipse
{

 public final static int interv=200;
 public FinalStateFigure()
 {
     this.setForegroundColor(org.eclipse.draw2d.ColorConstants.red);
 }

 public void paintFigure(Graphics g)
 {
     super.paintFigure(g);
     g.setBackgroundColor(org.eclipse.draw2d.ColorConstants.red);
     g.fillOval(bounds.x+interv, bounds.y+interv,
                bounds.width-2*interv, bounds.height-2*interv);
 }
}
```

After having the class defined as above, the qualified name should be filled in the Properties view of the Custom figure(gmfgraph editor).

Bibliography

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *RDesign Patterns: Elements of Reusable Object-Oriented Software* 2001.

[2] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose *Eclipse Modeling Framework* 2004.

[3] Iftikhar Azim Niaz *Automatic Code Generation From UML Class and Statechart Diagrams* 2005.

[4] Jos Warmer, Anneke Kleppe *The Object Constraint Language. Getting your models ready for MDA* 2003.

[5] avid Harel and Hillel Kugler *The Rhapsody Semantics of Statecharts (or, On The Executable Core of the UML) 2003.*

[6] Ian Horrocks Constructing the User Interface with Statecharts *1999.*

[7] Jonathan Kaye, David Castillo Flash MX for Interactive Simulation *2002.*

[8] Markus Voelter, Bernd Kolb, Sven Efftinge, and Arno Haase From Front End ToCode *2006.*

[9] JBoss Rules http://labs.jboss.com/portal/jbossrules/docs*.*