UNIVERSITY OF OSLO
Department of Informatics

User-defined Code generation from UML 2.0

Master thesis

Asbjørn Willersrud

22nd May 2006

Acknowledgments

This thesis is submitted to the Department of Informatics at the University of Oslo as part of my Master degree.

I would like to thank my supervisor, Øystein Haugen, for his valuable advice and feedback, and for continuously guiding me in the right direction.

I would also like to thank my family and friends for the support they have given me throughout this challenging period.

Table of Contents

1 Introduction **6**
1.1 Thesis structure 6

2 Technologies **8**
2.1 Model technologies and standards 8
2.1.1 Model Driven Architecture (MDA) 8
2.1.2 Meta-Object Facility (MOF) 8
2.1.3 Eclipse Modeling Framework (EMF) 9
2.1.4 The Unified Modeling Language (UML) 10
2.2 Transformation technologies and standards 11
2.2.1 Properties of transformations 11
2.2.2 QVT 12
2.2.3 ATL 14
2.2.4 IBM Model transformation framework (MTF) 14
2.2.5 Rational Software Modeler transformation framework 15
2.2.6 MOFScript 15
2.2.7 Java Emitter Templates 15

3 JavaFrame **17**
3.1 StateMachines 17
3.2 Composites 20
3.3 Mediators 22
3.4 Messages 22

4 Modularizing transformations **23**
4.1 Rules as modules 23
4.2 Transformations as modules and intermediate meta-models 24

5 Case: UML2JavaFrame **27**
5.1 Intermediate meta-model choice 27
5.1.1 A subset of the UML meta-model 27
5.1.2 Java meta-model 30
5.1.3 JavaFrame meta-model 30
5.2 Transformation architecture and technology 31
5.2.1 Transformation architecture 31
5.2.2 Transformation technology choice 32
5.3 Advanced meta-model concepts 33
5.3.1 Read-only 34
5.3.2 Derived 34
5.3.3 Subsets 34
5.3.4 Derived union 34
5.3.5 Redefines 34
5.3.6 Ecore profile 35

5.3.7 Eclipse vs. RSM ... 35
5.4 JfJava intermediate meta-model ... 35
5.4.1 Java meta-model ... 36
5.4.2 JavaFrame meta-model extension ... 38
5.4.3 JfJava Transformations ... 41
5.4.4 Generating meta-models from BNF ... 42
5.5 JfUml intermediate meta-model ... 45
5.5.1 Transformations ... 47
5.6 Comparing the JfUml to JfJava approaches ... 49

6 Customizing the generated code ... 50
6.1 User roles ... 50
6.2 Classification of changes ... 50
6.2.1 Supporting more of UML ... 50
6.2.2 Supporting extensions to UML ... 51
6.2.3 Model checking ... 51
6.2.4 Quality of Service (QoS) change ... 51
6.2.5 Platform change ... 51
6.2.6 Different semantic variation point implementation ... 51
6.3 Examples ... 52
6.3.1 Generating Tests for JavaFrame models ... 52
6.3.2 Generating C# code ... 52
6.3.3 Adding memory optimization to signals ... 53
6.3.4 Reusing the core ... 54
6.4 Maintenance ... 55

7 Summary and conclusions ... 56
7.1 Future work ... 56

8 References ... 57

Appendix A - JfUml Transformations ... 59

Appendix B - JfJava Transformations ... 79

List of Figures

Figure 1 3-level meta-model architecture 9
Figure 2 JavaFrame concepts 17
Figure 3 Tree view of PtnStateMachine from RSM model explorer 18
Figure 4 The region of PtnStateMachine 19
Figure 5 Internal structure of an example Composite 21
Figure 6 UML2Java composite transformation 25
Figure 7 Transformation architecture with extended intermediate meta-model 32
Figure 8 The JfJava meta-model composed of two parts 36
Figure 9 Java meta-model kernel 37
Figure 10 CompilationUnit, Class and related elements 38
Figure 11 The structure of the JavaFrame extension to the Java meta-model 39
Figure 12 The CompositeStateClass element and its contents representing a UML region 40
Figure 13 JfJava transformation architecture 41
Figure 14 A simplified model showing the process of creating a meta-model from BNF43
Figure 15 Meta-model produces from BNF 44
Figure 16 JfUml adaptations of UML 2.0 elements 46
Figure 17 A JavaFrame meta-model that extends classes from the UML meta-model.... 46
Figure 18 CompositeStateClass and related elements 47
Figure 19 Transformations using a JavaFrame intermediate meta-model that extends the UML meta-model 48
Figure 20 JfUml transformation architecture extended with support for UML Testing Profile 52
Figure 21 C#Frame Transformations 53
Figure 22 QoS Signal memory optimization 53
Figure 23 GWT extension, reusing the core transformation layer 55

1 Introduction

Traditionally compilers for programming languages have been written to generate a specific set of code. The Unified Modeling Language is different in that it does not specify complete semantics for all its concepts. UML is designed to be able to specify software system in a wide variety of domains. The generated code for the different domains will inevitably be different. For example there is a very different set of demands for a real-time system as opposed to a bookkeeping system. This was of course true before UML, but programming languages are at a lower level of abstraction and the set of choices made to implement the concepts of a programming language does not change much from domain to domain. UML on the other hand has a higher level of abstraction and thus require more choices about how to implement it. Since there is a wide variety of ways to implement concepts in UML, with different advantages and disadvantages, there is a need for different code generators and a way for the users to customize the generated code.

This thesis is based on an existing UML compiler written by myself as a plug-in for Rational Software Modeler. I use a subset this compiler as an example scenario throughout this thesis. This scenario is implemented using different transformation technology and a transformation architecture is presented that aims at helping users produce code generators that are customizable in a compact and maintenance friendly way.

1.1 Thesis structure

Chapter 2 Technologies

This chapter gives an overview of technologies and standards relevant to the thesis. Both modeling and transformation technologies are covered.

Chapter 3 JavaFrame

This chapter explains the JavaFrame framework JavaFrame is the target platform of the UML compiler and the example scenario.

Chapter 4 Modularizing transformations

This chapter introduces reuse mechanisms for transformation rules, explains how transformations can be used as modules by inheritance and composition. In addition the concept of an intermediate meta-model is introduced.

Chapter 5 Case: UML2JavaFrame

This chapter introduces a general transformation architecture and implements it using two different approaches. The choices made during the implementations and differences and similarities between the approaches are explained.

Chapter 6

Chapter 6 shows how the transformation architecture introduced in chapter 5 can be used to customize code generation.

Chapter 7 Summary and conclusions

This chapter summarizes the thesis, and explains possible future work.

2 Technologies

This chapter gives brief descriptions of relevant technologies and standards. First modeling and meta-modeling are explained and then transformation technologies and standards.

2.1 Model technologies and standards

2.1.1 Model Driven Architecture (MDA)

Throughout the history of software development new concepts and languages have increased the level of abstraction, from assembler to procedural languages to object orientation. Creating models of systems raises the abstraction level another level. Besides raising the abstraction level, using models has other advantages:

- Models help visualize the system and it becomes easier to get an overview of the system.
- Models are documentation of the system in addition to specification.

Model Driven Architecture (MDA) was proposed by Object Management Group (OMG) in 2002 and is an approach to software development that is based on formal use of models. Several models are used to describe a system. The process begins with creating a model of the system on an abstract platform independent level. Transformations are defined to transform the platform independent model to a platform specific model. The idea is to abstract away the platform by creating several transformations to different platforms and if a system needs to run on a different platform than originally intended all that needs to be done is transform the platform independent model using a different transformation. In this way implementation details are separated from application logic.

The final step is to generate code from the platform specific model [1] and this is the focus of this thesis. Although code generation from models may follow the pattern described by MDA, it does not have to. This thesis makes no assumption about what platform the source model is specified for and whether or not MDA is used.

2.1.2 Meta-Object Facility (MOF)

In order to precisely define a model, for example a UML model, a set of concepts and constraints that define what legal UML models may look like is needed. This is achieved by creating a model of UML, this model is a model of a model: a meta-model. All UML models must conform to this UML meta-model. However all models must conform to their meta-model and the UML meta-model is no exception, a need for a metameta-model[*] arises. This leads to a problem because even this metameta-model needs a meta-

---

[*] I will not use the term metameta-model throughout this thesis. I will only separate between models and meta-models.

model to conform to, leading to an infinite chain of meta-models. MOF [2] solves this problem by defining a meta-model that conforms to itself.

MOF is a standard from OMG for meta-modeling. It was originally created in order to have a meta-model to define UML. MOF may be described using the 3-level architecture shown in Figure 1. The 3-level architecture consists of 3 model levels: M1, M2 and M3.

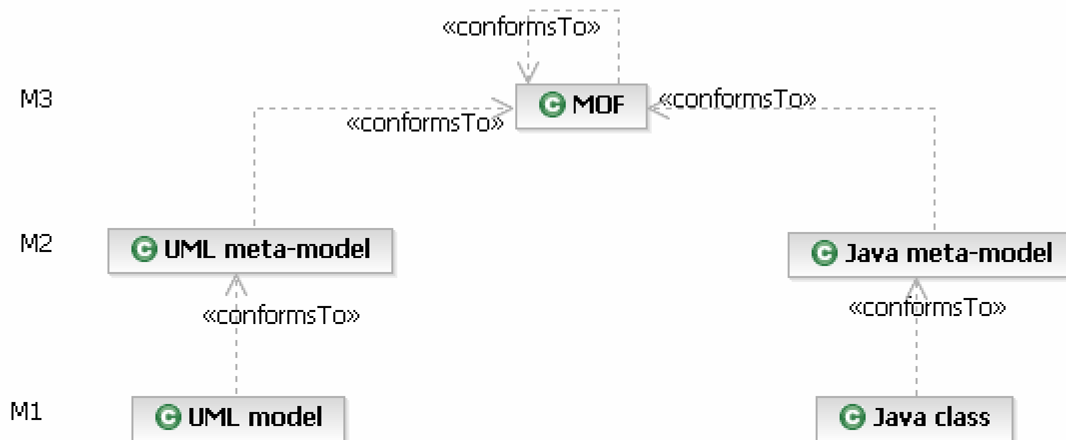

Figure 1 3-level meta-model architecture

At the M1 level there are normal models, at the M2 level there are meta-models (e.g., the UML 2.0 meta-model) and at the M3 level there are metameta-models. MOF is at the M3 level and is its own meta-model. Sometimes a M0 level is included which would represent the actual system a UML model represents.

2.1.3 Eclipse Modeling Framework (EMF)

EMF [3, 4] is a framework for modeling built on top of Eclipse. It was originally based on MOF, but it only provides a partial implementation of the MOF standard. EMF was a big influence on the EMOF (Essential MOF), which is part of the MOF specification. EMF provides essentially the same functionality as EMOF.

Based on an EMF input model EMF can:

- Generate a java implementation of the model
- Generate a simple graphical editor for the model as a plugin for Eclipse.

The input model consists of

- Classes
- Class attributes
- Relationships between classes
- Operations
- Simple constraints (i.e., cardinality)

Essentially it represents a UML class diagram, although it does not have to be specified using UML. Different supported input formats are annotated Java classes, UML model and XML schema.

EMF converts the input model to an Ecore model. Ecore is the meta-model for all EMF models.

The generated java implementation of the model includes code for persistence using XML/XMI. Both serialization and deserialization is supported automatically.

Uses of EMF include, among other things, modeling the data of an application and meta-modeling. In this thesis I will use EMF to model meta-models. I use UML models as input models.

2.1.4 The Unified Modeling Language (UML)

OMG's Unified Modeling Language [5-7] is a language primarily for designing and visualizing software systems, though it is not limited to modeling software and is often used for modeling business processes, organizational structure and many other kinds of systems/domains. UML is built upon object oriented concepts like classes and operation, however non object oriented systems may also be modeled using UML.

UML is the OMG's most used specification and together with MOF it provides the foundation for MDA.

The first version of UML was released in 1997 and has since been expanded and revised several times. The current version of UML is 2.1, although in this thesis all uses of UML will be of version 2.0 because the tools are not yet updated to version 2.1.

The UML standard is divided in four parts: Superstructure, Infrastructure, Diagram Interchange and Object Constraint Language (OCL). The superstructure and infrastructure documents define the UML meta-model using MOF. The diagram interchange document defines an extension to the UML meta-model so that diagram information is also included. OCL is a language for querying both MOF and UML models.

UML specifies 13 different diagram types. These are organized in three categories:

Structure Diagrams: Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram and Deployment Diagram.

Behavior Diagrams: Use Case Diagram, Activity Diagram and State Machine Diagram.

Interaction Diagram: Sequence Diagram, Communication Diagram, Timing Diagram and Interaction Overview Diagram.

The UML specification is very big and in this thesis I mainly use classes, composite structures and state machines.

The Eclipse UML2 project provides an implementation of the UML 2.0 meta-model using EMF.

2.1.4.1 Profiles

UML includes a built in way of extending the language. This is done using profiles. A profile consists of stereotypes, tagged values and constraints.

A stereotype is defined as an extension to a meta-element and the stereotype may redefine the semantics of the element. A stereotype may have tagged values which are properties with name and type (the types of tagged values are restricted to primitive types).

2.2 Transformation technologies and standards

2.2.1 Properties of transformations

T. Mens et al [8] and Czarnecki and Helsen [9] describes several different properties of transformation approaches. I show a short overview.

2.2.1.1 Rules

All transformation implementations use rules. A single transformation consists of set of rules. The rules can be defines either for imperative or declarative execution/interpretation. Rules can be a composite of other rules and it is possible to include object oriented properties such as inheritance and polymorphism. Rules are usually described textually, but can also be described graphically.

2.2.1.2 Transformation composition

A transformation can be a composition of other transformations and these sub-transformations can be run either in sequence, parallel or other more advanced control flow mechanisms.

2.2.1.3 Direction

A transformation can be defined to transform in only one direction, in other words from one meta-model to another, or a transformation can be defined in such a way that it can be run both ways. Even more than 2 directions are possible if more than 2 meta-models are specified as input/output. Declarative transformations are often suited for defining transformation in two directions, however for many transformations it will not be possible to define more than one direction at a time.

2.2.1.4 Trace

A transformation can create trace elements between source and target elements. These trace elements can be of use for among other things debugging of the model and synchronization between models. The trace elements can be stored in the model itself or separately.

2.2.1.5 Relation between source and target model

Some transformations may have the same target and source model (for instance for refactoring), but the most common scenario is to have a separate source and target model.

Some approaches create a new target model and others may also update an existing target model. The updates can be constrained to only extension, that is elements in the target model can not be deleted.

2.2.2 QVT

Query View Transformation [10] is an upcoming standard from OMG, it is currently in the finalization faze. The QVT language is, as the name suggest, a language for querying, transforming and creating views from models.

Queries are expressions that filter or select model elements, combined with imperative logic. QVT uses OCL (Object Constraint Language) for queries.

A view is a model that is derived from another model and that should not be persisted. Views are not handled specifically by the QVT standard.

A transformation generates a target model from a source model. This is the main part of QVT. Views and queries can be seen as side effects of the transformation language. The transformation language is a hybrid declarative/imperative language and actually consists of three languages: Relational, Core and Operational mappings.

Conformance to the QVT standard is defined by a matrix of language levels and interoperability levels. The interoperability levels are: Syntax executable, syntax exportable, XMI executable, XMI exportable and the language levels are Relational, Core and Operational Mappings. Currently only partial implementations of the standard exist.

2.2.2.1 Relational language

The Relational language is a declarative language that specifies relationships between elements in MOF models. The example below is taken from the QVT standard and it shows a transformation between UML and a Relational Database System model.

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {

    relation PackageToSchema /* map each package to a schema */
    {
        domain uml p:Package { name = pn }
        domain rdbms s:Schema { name = pn }
    }

    relation ClassToTable /* map each persistent class to table */
    {
        domain uml c:Class {
            namespace = p:Package{},
            kind = 'Persistent',
            name = cn
        }
        domain rdbms t:Table {
            schema = s:Schema{},
            name = cn,
            column = cl:Column {
                name = cn + '_tid',
                type = 'NUMBER'
```

```
				},
				primaryKey = k:PrimaryKey {
					name = cn + '_pk',
					column = cl
				}
			}
			when {
				PackageToSchema(p, s);
			}
			where {
				AttributeToColumn(c, t);
			}
		}
		...
}
```

This relational transformation may be executed in both directions. The different directions are represented by the domain blocks in each rule. If the above transformation was executed from SimpleUML to SimpleRDBMS, the ClassToTable rule would first check if any source class elements match the ‘c’ domain. In order to match kind = ‘Persistent’ must be true and the package that owns ‘c’ must be mapped to a Schema ‘s’. If these conditions hold there should be a Table element ‘t’ in the target model with ‘s’ as schema and the same name as ‘c’. If no such element exists the it is transformation engine’s responsibility to create one.

2.2.2.2 Core language

The Core language implements the same semantics as the Relational language, but at a lower level of abstraction. Transformations defined in the Core language are therefore more verbose than Relational. In the Relational language trace elements are automatically created on execution, in the Core language this must be specified manually.

A transformation, defined in the Core language, from the Relational language to the Core language is provided in the QVT specification. This is so a transformation engine can simply implement the Core language and transform any relational language input before executing it.

2.2.2.3 Operational mappings

The operational mappings part of QVT is an imperative language. It can either be used to define a completely imperative transformation or to implement relations from a relational transformation.

Below is an example of a package to schema mapping:

```
mapping Package::packageToSchema() : Schema
	when { self.name.startingWith() <> "_" }
{
	name := self.name;
	table := self.ownedElement->map class2table();
}
```

Operational Mappings provides OCL extensions with side effects that allow a more procedural style. Some problem are not possible to define using the declarative constructs found in the Relational language, the Operational Mappings language is more expressive.

2.2.2.4 Black box operations

QVT provides a way to 'plug-in' any MOF Operation as an implementation of a Relation. This makes it possible to implement Relations using any programming language that has a MOF binding (e.g., Java).

This is advantages because some things may be difficult to express using the Relations or Operational Mappings languages. In addition this allows for reuse of existing domain specific APIs which would be a lot of work to recode using QVT.

2.2.3 ATL

The Atlas Transformation Language [11] is developed by INRIA research group located at the University of Nantes. Like QVT it is a hybrid declarative and imperative language and currently ATL is probably the language that most closely resembles the QVT standard.

Unlike QVT ATL is unidirectional, that is a transformation definition can only be executed in one direction.

Below is an example of a declarative transformation with only one rule.

```
module SimpleUML2SimpleRDBMS;

create OUT : UML from IN : RDBMS;

rule Package2Schema {
	from p : UML!Package
	to
		s : RDBMS!Schema(
			name <- p.name
		)
}
```

The 'from' and 'to' statements are the equivalent to QVT Relational's domain blocks.

2.2.4 IBM Model transformation framework (MTF)

MTF [12] was developed as a prototype for a transformation language and it is a completely declarative language for implementing transformations between EMF models.

Below follows a rule that relates a package to a schema:

```
relate Package2Schema( uml:Package p, rdbms:Schema s) {
	equals(p.name, s.name),
	Class2Table(over p.ownedMember, over s.tables)
}
```

2.2.5 Rational Software Modeler transformation framework

The Rational Software Modeler (RSM) transformations framework is a Java framework for transforming EMF models. Like transformation languages a RSM transformation consists of a set of rules. These rules are defined using Java classes that extend the framework class AbstractRule. Each rule implements a method for creating the target object (or target code if it is model-to-text). This rule directly manipulates the generated EMF APIs for the target and source meta-models.

Using this approach is a lot more verbose than using a transformation language, but given the low level of maturity for current transformation languages, it may yet be a good option.

2.2.6 MOFScript

MOFScript [13] is an extension to QVT for model-to-text transformations. It is based on QVT Operational Mappings, which is the imperative part of QVT. MOFScript is currently under development by SINTEF.

```
jfuml.JavaFramePackage::mapPackage() {
	self.ownedMember->forEach(c : jfuml.JavaFrameClass) {
		c.mapClass()
	}
	self.ownedMember->forEach(c:jfuml.JavaFramePackage) {
		c.mapPackage()
	}
}

jfuml.JavaFrameClass::mapPackageDeclaration() {
	println("package "+self.owner.getQualifiedName()+ ";")
}

jfuml.JavaFrameClass::mapClassDeclarationStart() {
	<%public class %> self.name <% extends %> self.getGeneralLiteral(OBJECT) <% { %>
	nl
}
```

The above code shows examples of built in OCL expressions (the mapPackage rule), print statements and the template functionality.

2.2.7 Java Emitter Templates

Java Emitter Templates (JET) [14] is a template based language for text generation. A JET template consists of pure text with embedded java code in <% %> markers. An example is shown below.

```
<%	ClassBuilder c = (ClassBuilder)argument; %>
package <%= c.package %>;

public <%= c.abstract ? “abstract ” : “”%>class <%= c.className %> extends <% c.superType %> {
```

```
<% …. %>
}
```

Each JET template has a Java object passed to it as an argument; the ClassBuilder object in the example above. A JET template file is compiled to a java class. This generated class has a single method which takes a single object as a parameter and returns the generated code as a String.

3 JavaFrame

This thesis is based on an existing transformation written in Java that generates Java code from UML 2.0. I will use the same scenario and equivalent transformations implemented using different technologies as an ongoing example throughout this thesis. The generated Java code is based on JavaFrame. In order to explain these examples I will first explain how generated JavaFrame code represents the source UML model.

JavaFrame is a framework for implementing a subset of UML in Java. It consists of a Java API (Application Programming Interface) for implementing models in Java and a set of programming guidelines. The API contains classes for model elements like StateMachine, Composite and Mediator (representing a UML port). A class diagram for the JavaFrame concepts is shown in Figure 2. A JavaFrame system is a Composite which contains ActiveObjects. These ActiveObjects can be either other Composites or StateMachines. The ActiveObjects communicate with each other by sending asynchronous messages through Mediators. However the details of how JavaFrame systems work are beyond the scope of this thesis.

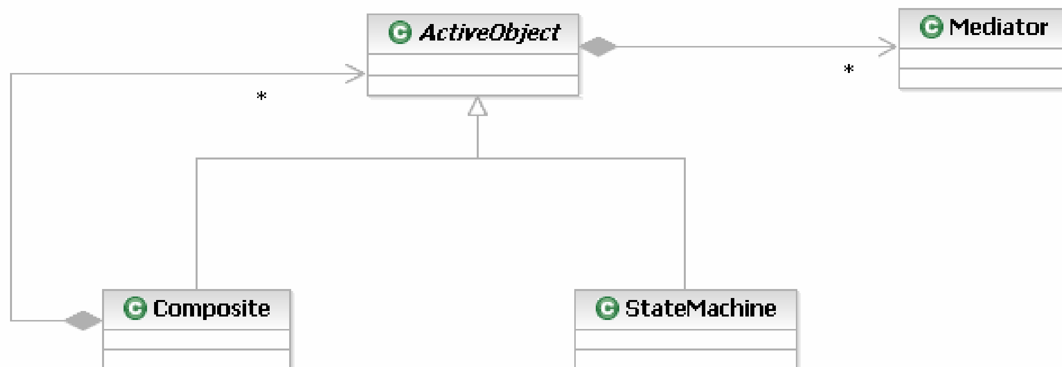

Figure 2 JavaFrame concepts

To implement models using JavaFrame new classes that extend the API classes must be implemented using the programming guidelines/patterns. Below follows examples of UML 2.0 model elements and the JavaFrame code that implements those elements.

3.1 StateMachines

A UML Statemachine is implemented by two Java classes, one extends the JavaFrame StateMachine class and the other extends the JavaFrame CompositeState class. Respectively they represent UML statemachine and region elements. Like UML statemachine elements a JavaFrame statemachine can have ports, attributes and operations, in addition it has a static compositestate representing its UML region.

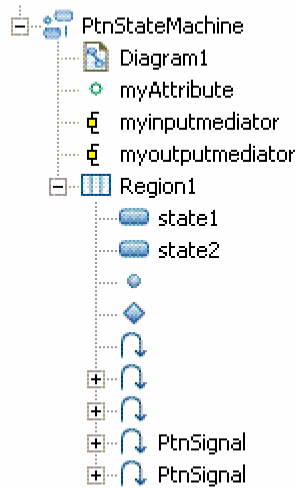

Figure 3 Tree view of PtnStateMachine from RSM model explorer

Figure 3 shows a statemachine with two ports and one attribute (of type String which is not shown) and below is the code that implements this statemachine in JavaFrame.

```
public class PtnStateMachine extends StateMachine {

        static CompositeState states = new PtnStateMachineStates("outermostState");

        /* Formal ports */
        public Mediator myinputmediator;
        public Mediator myoutputmediator;

        /* Attributes */
        public String myAttribute;

        protected void execStartTransition() {
                states.enterState(this);
        }

        public PtnStateMachine(Scheduler sched, Mediator myinputmediator,
                        Mediator myoutputmediator) {
                super(sched);
                myinputmediator.addAddress(this);
                this.myinputmediator = myinputmediator;
                this.myoutputmediator = myoutputmediator;
        }
}
```

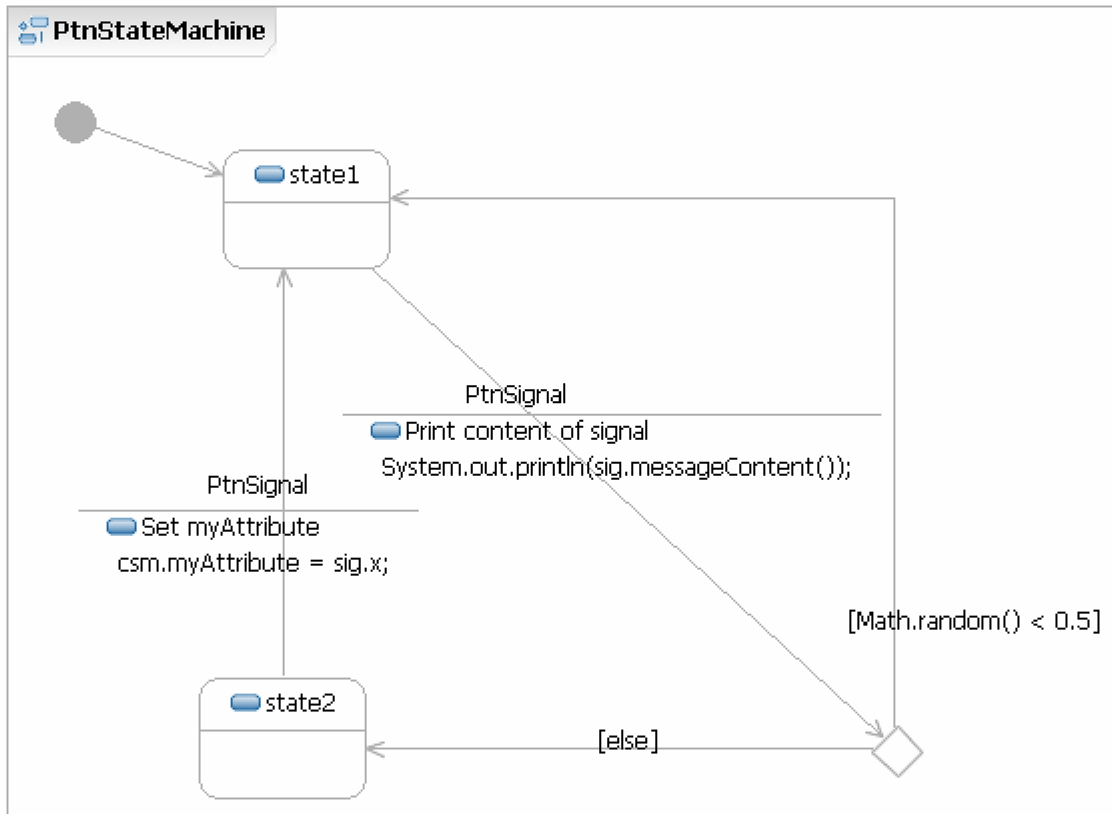

Figure 4 The region of PtnStateMachine

Figure 4 shows a regular UML state machine diagram of PtnStateMachine. Note that the effects of the transitions are represented by UML activity element. These are used to represent java code as text. This code is executed whenever the transition fires.

The JavaFrame class that implements this region is shown below. The states are implemented as static fields and each statemachine instance has a pointer to its own current state. All the transition logic is implemented by the execTrans method. This checks the current state and the received message and fires any transitions according to the model.

```
public class PtnStateMachineStates extends CompositeState {

   static State state1 = new State("state1");
   static State state2 = new State("state2");

   public PtnStateMachineStates(String sn) {
      super(sn);
      state1.enclosingState = this;
      state2.enclosingState = this;
   }

   public void enterState(StateMachine curfsm) {
      PtnStateMachine csm = (PtnStateMachine) curfsm;
      entry(curfsm);
      state1.enterState(curfsm);
```

```
    }

    protected boolean execTrans(Message signal, State st, StateMachine curfsm) {
        PtnStateMachine csm = (PtnStateMachine) curfsm;
        if (st == state1) {
            if (signal instanceof PtnSignal) {
                PtnSignal sig = (PtnSignal) signal;
                performExit(csm);
                //Begin effect code
                System.out.println(sig.messageContent());
                //End effect code
                if (Math.random() < 0.5) {
                    state1.enterState(curfsm);
                } else {
                    state2.enterState(curfsm);
                }
                return true;
            }
        } else if (st == state2) {
            if (signal instanceof PtnSignal) {
                PtnSignal sig = (PtnSignal) signal;
                performExit(csm);
                //Begin effect code
                csm.myAttribute = sig.x;
                //End effect code
                state1.enterState(curfsm);
                return true;
            }
        }
        return false;
    }
}
```

3.2 Composites

Figure 5 shows the internal structure of the UML class PtnComposite. It has two parts: sm1 and sm2. Both parts are of type PtnStateMachine. The tree-view to the left shows the child elements of PtnComposite and PtnStateMachine. PtnComposite has 9 child elements: 4 connectors, 2 ports, 2 parts (sm1 and sm2) and 1 diagram. The parts are actually property elements with the isComposite attribute set to true.

Note that in the diagram both the parts are shown with ports. These ports are inferred from the type of the part, in this case PtnStateMachine. As shown in the tree-view PtnStateMachine has two ports, myinputmediator and myoutputmediator.

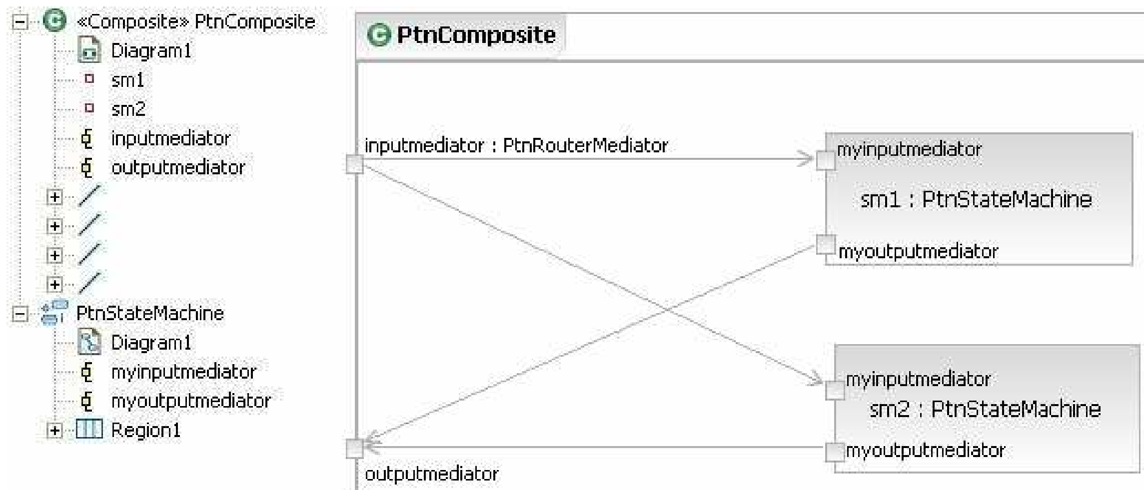

Figure 5 Internal structure of an example Composite

The produced JavaFrame code for the PtnComposite class is shown below. The ports owned by PtnComposite (i.e., formal ports) are declared as fields and so are the parts. In addition the ports of the parts (i.e., actual ports) are also declared as fields using the concatenation of the name of the part and the name of the port as the name of the field.

```
public class PtnComposite extends Composite {

    /* Used internally by JavaFrame */
    protected Scheduler sched;

    /* Formal ports of PtnComposite */
    public PtnRouterMediator inputmediator;
    public Mediator outputmediator;

    /* Part sm1 */
    protected PtnStateMachine sm1;
    public Mediator sm1myinputmediator;
    public Mediator sm1myoutputmediator;

    /* Part sm2 */
    protected PtnStateMachine sm2;
    public Mediator sm2myoutputmediator;
    public Mediator sm2myinputmediator;

    public PtnComposite(Scheduler sched, PtnRouterMediator inputmediator, Mediator
                                        outputmediator) {
        super();
        this.sched = sched;

        this.inputmediator = inputmediator;
        this.outputmediator = outputmediator;

        /* Creating property sm1 */
        sm1myinputmediator = new Mediator();
        sm1myoutputmediator = new Mediator();
        sm1 = new PtnStateMachine(sched, sm1myinputmediator,
```

```
                                    sm1myoutputmediator);
        addActiveObject(sm1);

        /* Creating property sm2 */
        sm2myoutputmediator = new Mediator();
        sm2myinputmediator = new Mediator();
        sm2 = new PtnStateMachine(sched, sm2myinputmediator,
                                    sm2myoutputmediator);
        addActiveObject(sm2);

        /* Setting addresses for the mediators based on UML connectors */
        inputmediator.addAddress(sm1myinputmediator);
        inputmediator.addAddress(sm2myinputmediator);
        sm2myoutputmediator.addAddress(outputmediator);
        sm1myoutputmediator.addAddress(outputmediator);
    }
}
```

The four last lines of addAddress function calls represent the UML connectors.

3.3 Mediators

Mediators are JavaFrame representations of UML ports. Or rather an instance of a mediator represents a UML port. A mediator class represents the possible type of a port. The PtnComposite class shown as an example above has two formal ports. One of them has no type the other is of type PtnRouterMediator. PtnRouterMediator is a UML class with the Mediator stereotype applied. In the code this is represented by the fields of the PtnComposite class.

Custom mediator classes are implemented by extending the Mediator class from the JavaFrame API. The Mediator class from the JavaFrame API implements the Addressable interface. This interface contains one method: public void forward(Message sig). Any custom mediator class must implement this method. In the UML model the code implementing the forward method is represented as regular text by either an Activity or the Actions of an Activity.

3.4 Messages

Messages are the JavaFrame representations of UML signals. Each signal element is transformed to a class that extends the Message class from the JavaFrame API.

In UML signals are sent between ports, and in JavaFrame message objects are sent between mediator instances. This is achieved by using the forward method implemented by mediator classes. Like UML signals JavaFrame messages also trigger transitions in statemachines.

4 Modularizing transformations

Although UML elements like classes and attributes can be directly mapped to programming language classes and attributes, most elements in UML do not have any direct counterpart in programming languages (e.g. there is no concept of statemachines and composite structures in Java). When a transformation bridge abstraction levels it will inevitably become complicated and the need to split the transformation into manageable modules arises.

This chapter explains how transformations can be modularized using both rules and entire transformations as modules. The concept of an intermediate meta-model is introduced.

4.1 Rules as modules

A transformation definition consists of a set of rules. These rules form the basic modular elements of a transformation. This is discussed in detail by Kurtev et al in [15].

In order to effectively use rules as modules there need to be reuse mechanisms for the rules. There are several different ways of achieving this, including rule inheritance, composition, polymorphism and others. I will give some examples of reuse mechanisms from different model transformation languages.

In the declarative transformation language MTF rules can be reused by inheritance and composition. The two rules shown below is an example of this. The rules are taken from a transformation that copies a UML model.

```
abstract relate mapClassifier extends mapNamedElement(uml:Classifier e1, uml:Classifier e2) {
	mapGeneralization [0..1] (over e1.generalization, over e2.generalization)
	,ordered mapPackageImport(over e1.packageImport, over e2.packageImport)
}

relate mapClass extends mapClassifier(uml:Class e1, uml:Class e2) {
	ordered mapProperty(over e1.ownedAttribute, over e2.ownedAttribute)
	,ordered mapOperation(over e1.ownedOperation, over e2.ownedOperation)
	,ordered mapActivity(over e1.ownedBehavior, over e2.ownedBehavior)
}
```

The first rule relates Classifier elements in the source model to Classifier elements in the target model. This rule extends the mapNamedElement rule which is not shown and just makes sure the elements have the same name. The mapClassifier rule explicitly invokes two other rules: mapGeneralization and mapPackageImport, this is an example of a rule composed of two other rules.

The second rule, mapClass, inherits from the first. The semantics of this is similar to normal object oriented inheritance. The mapClass rule inherits the two rules invoked from mapClassifier. And the signature of mapClass strengthens the constraints of the mapClassifier signature (Class is a subtype of Classifier).

Using this approach if I were to define a mapSignal rule that extended mapClassifier (Signal is a subtype of Classifier, but not of Class) I would not have specify mapGeneralization and mapPackageImport again. As meta-models usually have large type hierarchies such a mechanism is very useful.

Unfortunately the QVT Relational (declarative) language does not have such an inheritance mechanism, however the QVT Operational Mappings (imperative) language does. Below is an example of inheritance in the Operational Mappings language.

```
mapping Classifier::mapClassifier() : Classifier inherits mapNamedElement {
	generalization := self.generalization->map mapGeneralization();
	packageImport := self.packageImport->map mapPackageImport();
}

mapping Class::mapClass() : Class inherits mapClassifier {
	ownedAttribute := self.ownedAttribute->map mapProperty();
	ownedOperation := self.ownedOperation->map mapOperation();
	ownedBehavior := self.ownedBehavior->map mapActivity();
}
```

MOFScript is a model-to-text transformation language so it is a little different. It does not have inheritance between rules, but it does have polymorphism. MOFScript rules are defined with a context element. The two MOFScript rules shown below is an example of polymorphism.

```
jfuml.JavaFrameClass::mapClassDeclarationStart() {
	<%public class %> self.name <% extends %> self.getGeneralLiteral(OBJECT) <% { %>
	nl
}

jfuml.CompositeClass::mapClassDeclarationStart() {
	<% public class %> self.name <% extends Composite { %>
	nl
}
```

Since CompositeClass is a subtype of JavaFrameClass the second rule overrides the first. Like normal polymorphism the type of the actual object at run-time specifies which rule is executed.

4.2 Transformations as modules and intermediate meta-models

In addition to rule modularization, transformations may be used as modules as well. There are two ways to achieve this, either by transformation inheritance or by composite transformations.

Reuse mechanisms for transformation rules forms the basis for transformations inheritance. The rules in the sub-transformation may reuse rule definitions in the super-transformation.

The second way is to introduce an intermediate meta-model and split a transformation in two parts. Figure 6 shows an example of a transformation that has been split it two composite parts. In [16] Oldevik discusses composition of transformations in more detail.

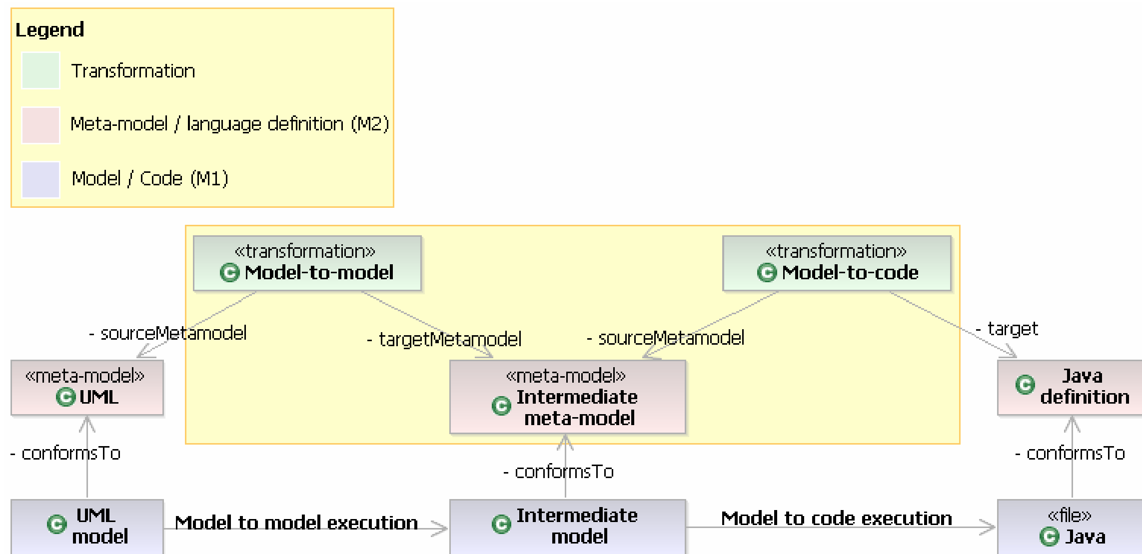

Figure 6 UML2Java composite transformation

The use of general intermediate representations, in the form of data structures and code, is used by compilers in order to separate the front end from the back end, and for splitting the code generation in modular parts [17]. The first intermediate representation a compiler uses is usually an abstract syntax tree of the source code. This abstract syntax tree is then transformed to intermediate code. A normal form for the intermediate code is stack based byte-code, effectively bridging the abstraction level between the abstract syntax tree and assembler type code. Examples of intermediate code include Java byte-code and CIL (Microsoft's Common Intermediate Language). The intermediate code may be interpreted at run-time or assembler code for the target machine may be generated.

For the same reasons compilers use intermediate representations it might be a good idea to use intermediate meta-models for code generation from models. The intermediate meta-model can be any MOF model, optionally with additional constraints. It may for instance be a UML model with a certain profile applied. In [18] Vanhooff and Berbers suggest using UML profiles to specify input and output model characteristics. Below follows some alternatives for what an intermediate meta-model may be:

- A subset of the UML meta-model that directly maps to code (i.e. no statemachines and composite structures, just classes/attributes/operations). A profile for the programming language will probably be needed. Suggested by Chauvel and Jézéquel in [19].
- A meta-model of the abstract syntax of the programming language.
- A dedicated meta-model for the implementation (e.g., a JavaFrame meta-model).

These different choices are discussed in detail in chapter 5.

An intermediate model should ideally satisfy these requirements:

- Simple to generate code from
- Simple to generate from the source UML model
- Simple to define
- Preferably not add any platform restrictions beyond what is already in the source UML model

5 Case: UML2JavaFrame

In chapter 3 I gave an explanation of JavaFrame and how it relates to UML. The UML 2.0 compiler this thesis is based on implements a transformation from UML to JavaFrame. I have implemented a subset of that transformation using two different intermediate meta-models and different transformation technologies. This chapter explains the process and the choices that were made while creating both the intermediate meta-models and the transformations.

First different ways of creating an intermediate meta-model is discussed and two different approaches are chosen. Then I present a general transformation architecture and discuss the decision of both model-to-model and model-to-text transformation technologies. Finally I show and explain the intermediate meta-models and the implemented transformations.

5.1 Intermediate meta-model choice

In chapter 4 I introduce the concept of an intermediate meta-model. In this section I propose different choices for intermediate meta-models, discuss the advantages and disadvantages of each choice and choose two options to implement.

5.1.1 A subset of the UML meta-model

A subset of the UML meta-model is used as an intermediate meta-model. The subset is a direct mapping of Java code to UML elements. Because UML has a lot in common with object oriented programming languages like Java it is possible to represent a Java program with UML elements.

Like Java, UML contains concepts like class, interface, and package. UML attributes are very similar to Java fields and UML operations are almost equivalent to Java methods. There are differences though, for instance UML supports multiple inheritance, but Java does not. Also UML operations support multiple return values, which are not possible in Java.

The structure of a Java language may be represented in UML using these previously described elements, but UML operations really only represents the signature of a Java method (i.e., the name, return value and parameters). The implementation of an operation is represented by an activity element. It may be possible to use a UML activity element to represent the body of a Java method, but activities are not designed to be a Java action language. In [20] Rumbaugh et al discusses action languages for UML:

> *"The selection of one programming language as the basis for an action language would, therefore, have the effect of discouraging the others, which the designers did not want to do. The semantics of actions have therefore been left low level and free of implementation concerns within UML itself. For many practical uses, such as code generation, UML must be augmented with the action language (often standard programming language) that is being used."* [20] page 144

5.1.1.1 Behavior problem

As described above the main problem with using a subset of UML as an intermediate meta-model is how to deal with behavior / method bodies. How should a UML activity element represent Java code? One option is to simply insert the code as text. This is not a good solution for an intermediate meta-model because the model to model transformation would have to generate this code, something that is better left to the final model to text transformation.

A way of solving the behavior problem is to use stereotyped activity and action elements to represent the code. The idea would be not to use the control flow mechanisms of regular UML activities, but rely on the ordering of elements. This may be a problem with certain declarative transformation technologies if the order of generated model elements is nondeterministic. A possible stereotype could be ChoiceActivity which represents either switch or if-else statement others could be Call, Assignment and VariableDeclaration. This would alter the semantics of the Activity and Action UML elements and it is probably better to use an approach like one of the two suggested below.

Another option is to use the control flow mechanisms in UML activity diagrams and develop a Java profile for UML which contains stereotypes for activities and actions. UML already has different action elements which may be reused. Examples of these include Create action, Raise exception action, Return action, Test identity action and Write action.

Yet another way of representing behavior is to use a programming language independent UML action language, such as the Kermeta action language [21].

5.1.1.2 Fixed code problem

Even if the method body problem were solved there is another problem which remains. When a single element in the source model is implemented by a large amount of fixed code, this must be represented by a large number of elements in the intermediate meta-model. For example the JavaFrame Main class is generated once for every JavaFrame model and large parts of the generated code is fixed (i.e., it does not vary from model to model). One method which is an example of this is shown below.

```
public static void main(String[] args) {
    String hostname = "";
    int portnumber = 0;
    for (int i = 0; i < args.length; i++) {
        String token = args[i];
```

```
                if (token.equals("-remote")) {
                    if (i == (args.length - 1)) {
                        System.out.println("No remote specified.");
                        usage();
                        System.exit(1);
                    }
                    int index = args[++i].indexOf(':');
                    if (index != -1) {
                        hostname = args[i].substring(0, index);
                        Integer portnumberInt = new Integer(args[i]
                                .substring(index + 1));
                        portnumber = portnumberInt.intValue();
                    } else {
                        usage();
                        System.exit(1);
                    }
                }
            }
            new CompositeNameMain(hostname, portnumber);
        }
```

This code is the same for all models. The only exception is the constructor call on the last line where the name of the composite is used (CompositeNameMain is an example). If this entire method should be represented by using UML action elements or an action language it would be a very large and complex representation. The first model-to-model transformation would have to construct this fixed representation from one source element. This is a very cumbersome approach compared to generating this code in a model-to-text transformation where the code can just be written as is.

It might be possible to solve this problem by introducing parameterized template elements in the intermediate meta-model. However this will leave the responsibility for generating this fixed code to the model-to-model transformation. In my opinion it this is better left to the final model-to-text transformation.

5.1.1.3 Existing code problem

Another problem arises when the generated code should reuse existing code. In this case the generated code is based on an existing JavaFrame API. How can the UML model specify for instance that a class should extend a JavaFrame statemachine? One way is to use a JavaFrame extension to UML as described in 5.1.3, another is to reverse engineer the JavaFrame source code into a UML model and use a regular UML generalization to the appropriate class in the reverse engineered model. Although this will require that the transformation is aware of this third model and which classes in it represents which JavaFrame concepts.

Because of the problems listed above, the behavior problem, the fixed code problem and the existing code problem, it is my conclusion that using a subset of UML to directly represent the code leaves too much of the transformation logic to the first model-to-model transformation and I will not implement transformations using this intermediate meta-model. Though if these problems were solved it would be an interesting experiment too perform.

5.1.2 Java meta-model

The intermediate meta-model is a meta-model of the Java language. Like the previously described option this intermediate meta-model is a direct representation of the code. As a consequence of that some of the same problems apply.

The behavior problem is not a problem for a Java meta-model because Java has a well defined way of representing method bodies using various statement and expression elements.

The other two problems of directly representing the code remain; the fixed code problem and the existing code problem. In addition a Java meta-model adds a platform restriction in the capacity of being specific to Java. It may be possible to generate other kind of object oriented code (e.g., C++, C#) from a Java meta-model, but it will probably be more inconvenient than if UML was used.

Creating a meta-model for a programming language may require a lot of work. To make this process a little more automatic the meta-model may be generated from BNF using a tool such as *agramm* / *mmm* which is described in [22].

The conclusion for this option is the same as the last one; I believe this representation is too close to the code.

5.1.3 JavaFrame meta-model

As explained above both the previously described intermediate meta-models are basically another way to represent the actual code. This leaves almost all the transformation logic to the first model to model transformation, which probably is too much for a single transformation and much of the problems with generating code directly from the original UML model remain. In addition, the ability for the model to text transformation to customize the generated code is very limited if such an intermediate meta-model is chosen. A way of solving this is introducing JavaFrame specific elements in the intermediate model, either as a stand alone JavaFrame meta-model or as an extension to one of the previously described intermediate meta-models.

Having the JavaFrame meta-model extend either the UML meta-model or a Java meta-model provides the possibility of modularizing the transformations. The transformations may be split in a core part and an extended part as explained in section 5.2.1. Because of this I decided to drop the stand alone version of the intermediate meta-model and implement two intermediate meta-models with transformations, one that extends the UML meta-model and another that extends a Java meta-model. These will be called JfUml and JfJava respectively and are explained in section 5.4 and 5.5.

Defining the intermediate meta-model as an extension to the UML meta-model provides both advantages and disadvantages. A disadvantage is that there is a dependency on UML. If UML changes from one version to the next it may effect the intermediate model and any transformations from it. An advantage of extending UML is that elements that exist both in UML and JavaFrame do not need to be redefined (e.g., both UML and JavaFrame have concepts like class, attribute and operation). This provides advantages not only in that it is easier to define the meta-model, but also when defining transformations because these elements can simply be copied to the target model.

If the JavaFrame meta-model is defined as an extension to the UML meta-model inter-model references may also be used. That is, the target model may link to elements in the source model. This may be useful if creating an alternative representation of an element is not needed for code generation. For instance it may be decided that UML region elements are sufficiently easy to generate code from, and that creating alternative intermediate meta-model elements to represent a UML region is not necessary.

An essential property of the JavaFrame meta-model is how close it is to the code and how close it is to the UML model. When I modeled the meta-models I tried to create a one to one mapping between the meta-model and the produced JavaFrame code, this turned out to be a good guideline, although on some occasions it was necessary to deviate from this.

Another question is if any relations between model elements be replaced by string attributes? For instance can a generalization between two classes simply be replaced by the name of the extended class? Certain such shortcuts might make it easier to generate code, but the meta-model looses navigability. It is worth to note that navigability is very important in a meta-model and that the code generation should be very simplified to justify such a change. For this reason I decided not do this in my intermediate meta-models.

5.2 Transformation architecture and technology

5.2.1 Transformation architecture

Both of the chosen intermediate meta-models consist of a core part and an extension. The core parts are UML and Java meta-models and the extension is a JavaFrame meta-model in both cases. This allows me to create a general transformation architecture that I will use with slight modifications in both cases.

In chapter 4 I described a composite transformation, which consisted of an intermediate meta-model, a model-to-model transformation, and model-to-text transformation. When an intermediate meta-model extension is introduced, two more transformations need to be implemented. Both of these extend the original transformations. The general architecture is shown in Figure 7.

UML and Java are used as source and target in this diagram, but any meta-model and programming language could be substituted.

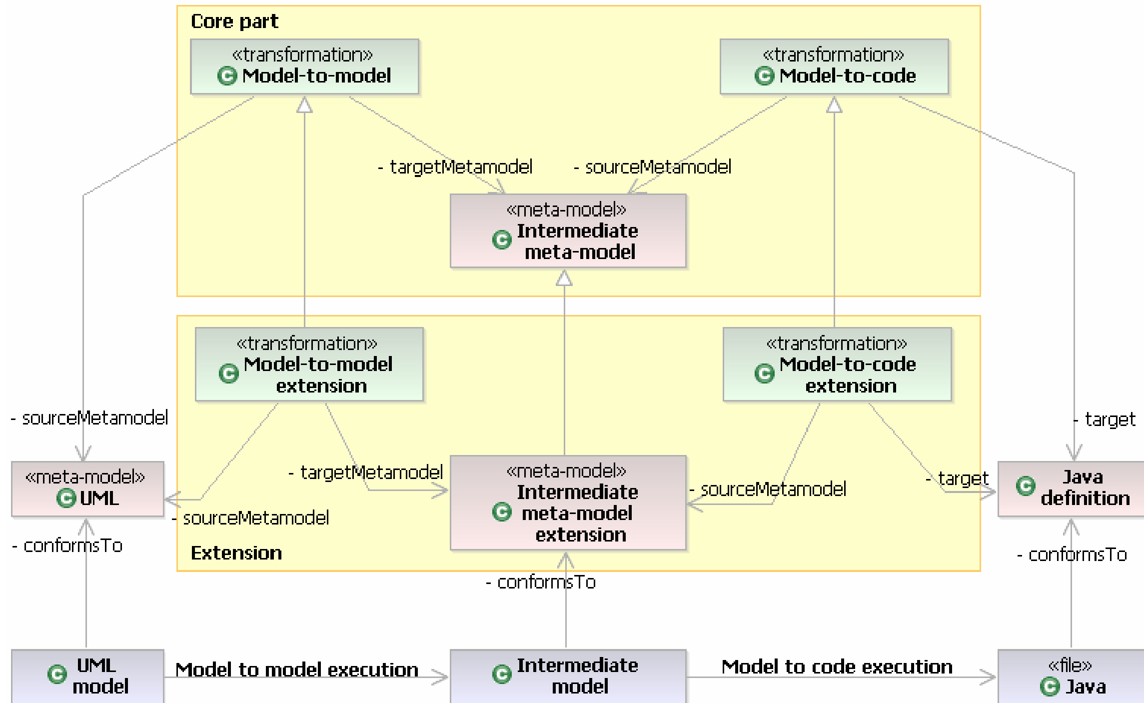

Figure 7 Transformation architecture with extended intermediate meta-model

Using this architecture what started out as one big transformation is now divided in four much smaller parts.

5.2.2 Transformation technology choice

Having chosen the architecture previously described, I needed to choose the transformation languages/technologies to implement the transformations. The architecture sets two requirements for the transformation technologies:

1. Transformation extension/inheritance must be supported.
2. The source or target meta-model must be possible to specify using two meta-models in separate files.

The second requirement is needed because when transforming for instance from a JavaFrame meta-model that extends the UML meta-model the transformation rules need access to not only elements from the JavaFrame meta-model, but also elements from the extended meta-model (i.e., the UML meta-model). In order to achieve this, the transformation must specify that the source meta-model is actually defined in two files, one for JavaFrame and one for UML. For model-to-model technologies it is the target meta-model that needs to be defined using two files, while for model-to-text it is the source.

For model to model transformation the ideal option would be to use QVT. However there are no sufficient implementations of QVT available at the moment. Borland has implemented part of the QVT standard, but it is only an implementation of the Operational Mappings language, none of the declarative languages (Relational and Core) are implemented. In addition the implementation is based on the QVT-Merge [23]

submission, not the final adopted specification. However the biggest problem is that no rule reuse functionality is implemented, making it very difficult to use with my transformation architecture.

The most commonly used model-to-model transformation language today is probably ATL, but this does not support any of the two requirements specified.

Two viable options are MTF and using the RSM transformation framework. I decided to use both of them; MTF for the intermediate meta-model that extends UML and the RSM framework for the intermediate meta-model that extends Java.

For model to text the alternatives are JET, MTF and MOFScript. MTF has a very limited support for text generation, which is primarily designed to create documents like for instance html. Because of MTF's limited support for text generation and that it only has declarative rules I chose not to use MTF.

JET is what I used for code generation in the transformation this thesis is based on. It offers limited modularization and no form of inheritance, although extension points may be defined. JET is not a transformation technology in its own, but a template language. To create a model-to-text transformation using JET, Java would have to be used to traverse the model and invoke the templates.

MOFScript has good QVT like imperative rules and is specifically designed to transform models to text. Although MOFScript currently support neither of the requirements specified it is in early stages of development and is likely to implement this at a later stage.

The final choice for model-to-text technology was MOFScript because it is very similar to QVT and once its limitations are overcome it will be a very good text-transformation language.

The problem that MOFScript does not support transformation extension sufficiently in the current version was solved by creating only one transformation and marking what part of it should be a subtransformation.

The problem of MOFScript not supporting source meta-models in several files was solved differently by each approach.

5.3 Advanced meta-model concepts

Before I explain the intermediate meta-models I created I will explain some not so well known UML constructs, I will explain the semantics of these and how they are shown in the diagram. I assume the reader is familiar with concepts like generalization, composition, visibility and multiplicity. More details about all these concepts can be found in [4, 6, 20].

Both meta-models are implemented using EMF and so a few advanced EMF concepts are also explained.

All these advanced concepts are regarding meta-properties of property elements.

5.3.1 Read-only

The read-only property means just what it says, it can only be read. In practice this means that the EMF code generator will generate a get method, but not a set method for the property. Because read-only and other properties do not have a graphical representation in class diagrams I use curly braces and list which of these properties are applied.

5.3.2 Derived

Derived means that the value of the property is computed from other properties. A derived property does not represent any additional object state. For instance a qualified name property of a java package is the concatenation of its own name and the names of its owning packages (with '.' in between). Such a property should be derived. Derived is denoted by a '/' in front of the property name. The generated code is unaffected by the derived property, but when a model object is copied derived properties are not copied.

5.3.3 Subsets

A property may subset another property. This means that the collection associated with an instance of the subsetting property must be a subset of the collection associated with the corresponding instance of the subsetted property. For instance Class has a 'field' property (a containment list of type Field), this property subsets 'ownedElement' from Element. This means that all fields owned by a class will also be contained in the 'ownedElement' property. The fact the a property is a subset of another does not effect the code generated for the subsetting property, however it does effect the generated code for the property that is subsetted as explained under derived union.

5.3.4 Derived union

A property marked as a derived union is derived by the union of all properties that subset it. The 'ownedElement' property of Element is an example of this. Continuing the example from subsets, Class not only has a 'field' property that subsets 'ownedElement', but also a 'method' property. This means that the 'ownedElement' property will contain all methods and fields of a class. The effect of marking a property derived union is that the generated get method will return a list containing the union of the values of all subsetting properties.

5.3.5 Redefines

A property may redefine another. It can change the name, multiplicity, type (to a subtype of the original) and other values. Multiplicity constraints can only be strengthened. That is the lower bound may be increased and the upper bound may be lowered. An example from the JfJava meta-model is the 'mediator' property of MediatorField which redefines 'type' from TypedElement, constraining the type to be a MediatorClass.

Most EMF properties are generated by a protected field holding the actual value of the property and a get and set method. A redefined property generates a set and get method like other properties, the difference is that the field that stores the value of the property is

shared with the redefined property. In the example of mediator from MediatorField redefining type from TypedElement, the MediatorField implementation has both getType and getMediator methods, but both return the protected mediator field.

5.3.6 Ecore profile

The previous concepts are all part of UML 2.0. However the model also includes some additional EMF specific concepts. This is achieved through a Ecore profile. The profile includes eAttribute and eReference stereotypes. Both have several tagged values, the relevant ones are isVolatile and isTransient.

If a property is transient it means that it should not be persisted. That is when a model instance is serialized as XMI (XML Metadata Interchange) any transient properties are not stored.

Usually EMF generates both getter and setter methods for properties. However if a property is marked as volatile just a method signature is generated, the implementation must be written manually.

Usually any derived properties are both volatile and transient (e.g., qualifiedName).

5.3.7 Eclipse vs. RSM

Rational Software Modeler is built on Eclipse version 3.0. This version includes versions of EMF and UML2 that do not support subsetting and redefinitions of properties. To work around this problem I originally planned to model the meta-model in RSM, export it and generate code and plug-in from Eclipse version 3.1. Then I could install the generated plug-in in RSM. However it turns out that even the generated code requires a newer version of UML2 than the one in RSM.

In the end I had to generate code from RSM and this means that subsetting and redefinitions do not currently work as intended. When a newer version of RSM is released what is needed is only a new code generation.

Without support for subsetting and redefinitions in the intermediate meta-models the transformations may have to do some workarounds.

5.4 JfJava intermediate meta-model

This section first explains the two parts of the JfJava meta-model, and then the implemented transformations are presented. Finally I examine the possibility of using a tool to generate a Java meta-model from BNF.

The JFJava intermediate meta-model consists of a Java meta-model and a JavaFrame extension. Even though they should be two separate models I decided to model them as one. This is because if they were modeled as separate models, a MOFScript transformation could only handle elements from one of them. If this situation changes it should be very easy to refactor the model into two separate models.

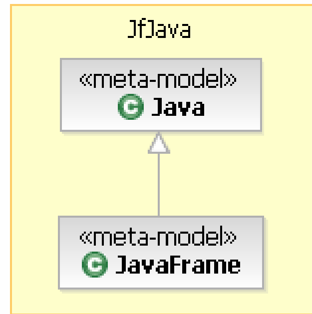

Figure 8 The JfJava meta-model composed of two parts

Figure 8 shows the JfJava meta-model and its two parts. Sections 5.4.1 and 5.4.2 describe the contents of these two meta-model parts.

5.4.1 Java meta-model

The Java meta-model was created partly based on the BNF for the Java language, partly from an existing Java meta-model from Sun which is used internally in Netbeans [24] and partly from my special needs. In addition the UML meta-model was used as inspiration for more general meta-model features.

Figure 9 show the core parts of the Java meta-model. The Java meta-model represents only a subset of Java. The model is very similar to an abstract syntax tree representation used by compilers, however this subset does not include any form of behavior. By behavior in this context I mean detailed expressions and statements, like for instance MethodInvocation, ForEachStatement and ConditionalExpression. These would have to be included if the model should be a complete representation of Java. To replace these elements there is a generic CodeString element (specializing both Statement and Expression), which represents any arbitrary code in a string attribute. This is used whenever code is represented as strings in the UML source model. For instance the existing transformation interprets the default value of a property as code; this string is transformed to a CodeString element.

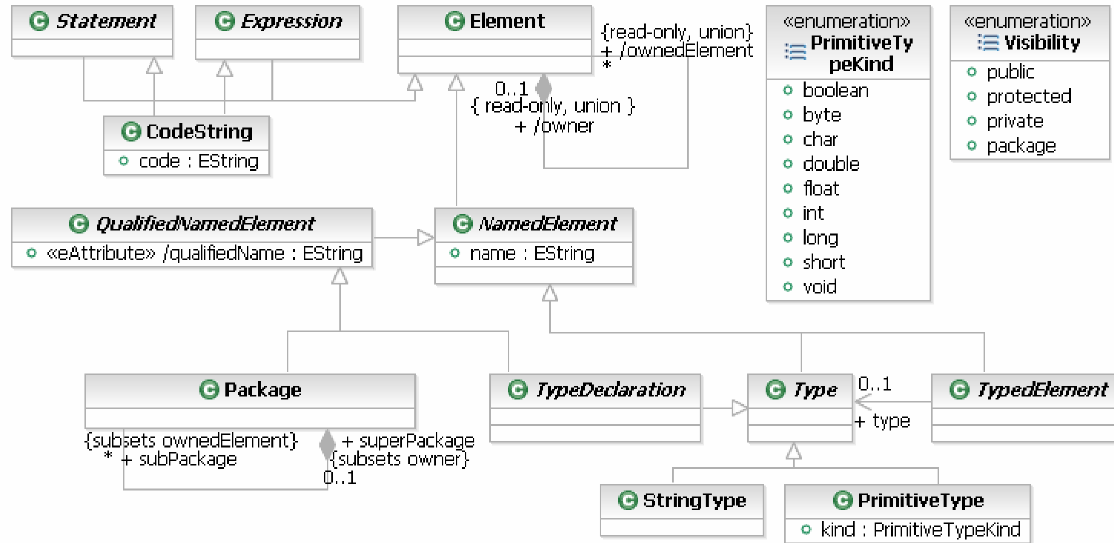

Figure 9 Java meta-model kernel

Element is the supertype of all classes in the model. This class has one composition association to itself. Both association end properties are derived unions and read-only. All composition association of subtypes of Element (i.e., all composition associations in the model) specialize this association and their association end properties subset owner or ownedElement. This is similar to how it is done in the UML 2.0 meta-model.

Element, NamedElement, TypedElement and Type are general abstract elements common to most meta-models.

One option for extension is to include a Model element as a root element of all models (like the UML meta-model). This could be used to contain java classes which should be part of the default (nameless) package. And it could have contained meta information about the model, for instance the path on the file system and maybe even the contents of the classpath. In addition a Namespace element would probably have been advantages.

The StringType element was needed to represent strings. In UML String is a primitive type, but in Java there is no equivalent such primitive type.

Figure 10 shows the rest of the Java meta-model. The two most central classes are CompilationUnit and Class. A compilationunit represents a java file and consists of a list of import statements and type declarations. In this model a type declaration can only be a class/interface, this could have been expanded to also include enumeration.

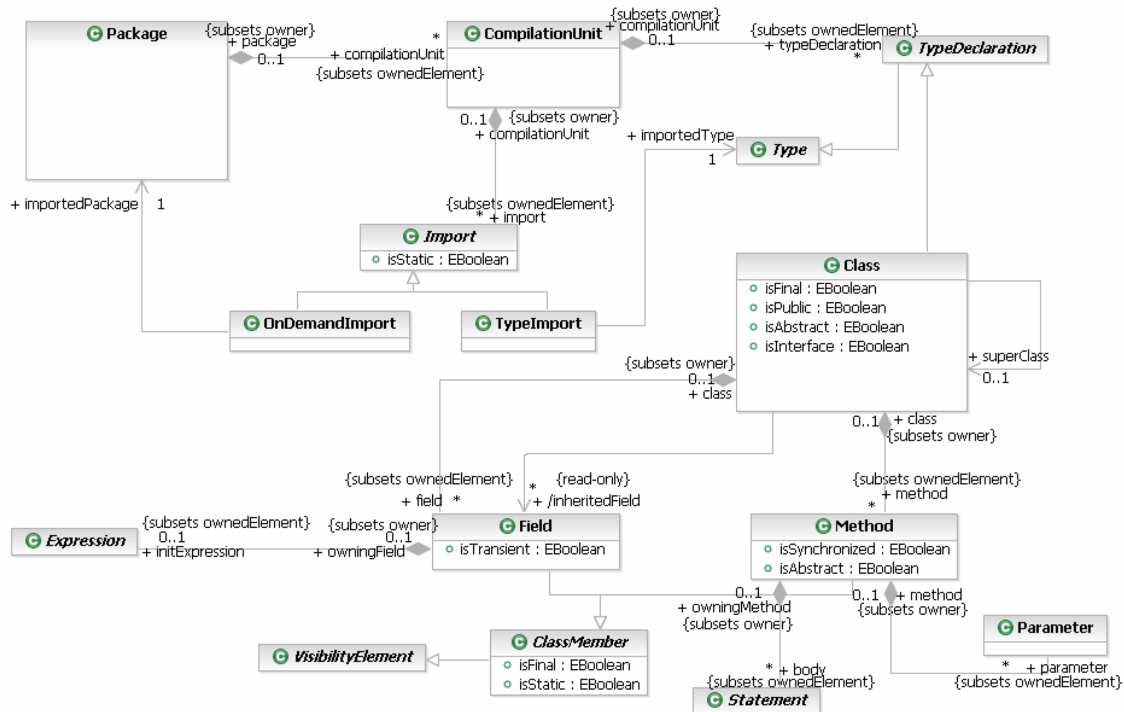

Figure 10 CompilationUnit, Class and related elements

When I was designing this meta-model I had a choice about how to structure the package and compilationunit elements. I could either use the BNF of the Java language as a starting point or use a more meta-model like approach.

The BNF rule for compilationunit looks like this:

CompilationUnit := PackageDecl Import* TypeDeclaration*

This would be the equivalent of ComilationUnit element with three composite associations to Import, TypeDeclaration and PackageDecl. PackageDecl would then be a new element with a reference to a Package element.

The other way, and the way I have chosen, is to have a composite association from Package to CompilationUnit. This leaves out the PackageDecl element and makes sure that all compilation units are contained within a package.

The reason the BNF for Java does not handle package structures very well is because it only describes the content of a single Java file, as opposed to the meta-model that must describe several files and packages.

Another omission is a Constructor element; this is because I let the model to text transformation generate constructors.

5.4.2 JavaFrame meta-model extension

The JavaFrame extension to the Java meta-model was created partly based on the UML 2.0 meta-model and partly from the structure of the JavaFrame implementation/code.

Figure 11 shows the main structure of the JavaFrame extension. Each class that has a special implementation in JavaFrame is represented by a class that extends the Class element from the Java meta-model.

Figure 11 The structure of the JavaFrame extension to the Java meta-model

The CompositeClass represents a UML class with internal structure. The composite association to the ActiveObjectField element represents UML parts, the composite association to MediatorField, which is inherited from ActiveObjectClass, represents UML ports and the composite association to MediatorConnection represents UML connector elements.

Both ActiveObjectField and MediatorField extend the Field element from the Java meta-model. However as ActiveObjectField represents a UML part it also needs to represent any actual ports on that part. This is achieved by a composite association to MediatorField. The effect of this is that each UML port of a part is represented by a MediatorField element in this meta-model. In UML a port is only one element, the parts themselves do not have port elements. This causes problems because connector elements connect to the ports of specific part not the general port of a type. A connector end has a ‘part with port’ property for determining which part the port belongs to. This is quite difficult to generate code directly from and the JavaFrame meta-model solves this problem by having each ActiveObjectField (uml: part) contain their own MediatorField (uml: port) elements. As shown the MediatorConnection class, which represents a UML connector, is directly connected to MediatorField with source and target associations, no ‘part with port’ construct is needed.

The JavaFrame meta-model need to represent the behavior, as well as the structure, of the implementation, this is done by introducing high level abstract elements. The behavior

that needs to be represented is the region of a statemachine. This is achieved by using the UML meta-model as a starting point and making it better suited for code generation to JavaFrame. As presented in chapter 3, a UML region is implemented by fields for states and a method called execTrans for the transitions. The execTrans method is a combined switch on current state and received signal type. All this is implemented by the JavaFrame CompositeStateClass which is shown in Figure 12 along with other related elements.

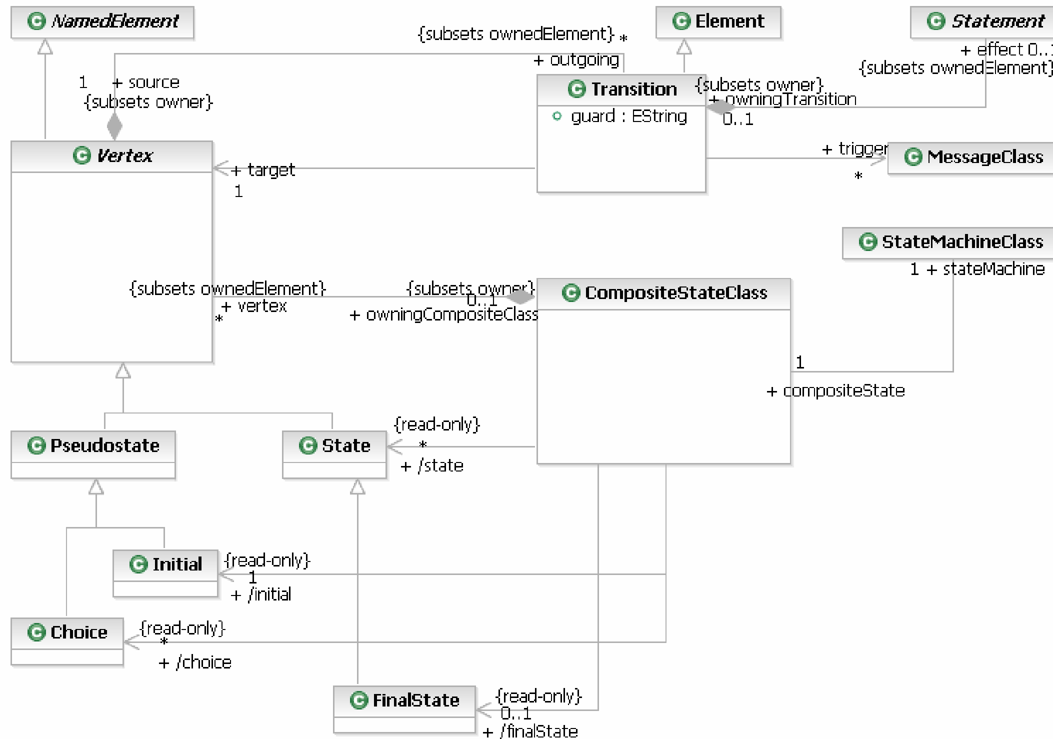

Figure 12 The CompositeStateClass element and its contents representing a UML region

Like in the UML meta-model, an abstract Vertex class is the supertype of all pseudostates and states. The CompositeStateClass has a composite association to the Vertex class and in addition several conveniences association to the concrete subtypes of Vertex. These are all derived, read-only, transient and volatile.

The UML region class has a composite association to the transition class. In this meta-model this is changed and Vertex is the class that owns transitions instead. This fits better with the execTrans method where first current state is checked then all outgoing transitions from that state is checked.

An advantage of keeping the intermediate meta-model on a relatively high abstraction level is that there is a choice about how to implement these abstract features. In this case there are at least two different ways to implement transitions. In the standard JavaFrame approach the code for transitions (guard, effect and new state entered) are inserted in the execTrans method. Another approach would be to generate a method for each transition.

Such a change might have been more difficult if the intermediate meta-model was closer to the code.

5.4.3 JfJava Transformations

The general transformation architecture used to implement the transformations from UML to JfJava and from JfJava to code. For this scenario I chose to use the RSM transformation framework as model-to-model technology. This allowed me to reuse part of the original UML compiler, as that was also written using the RSM transformation framework. The JfJava transformations are shown in Figure 13.

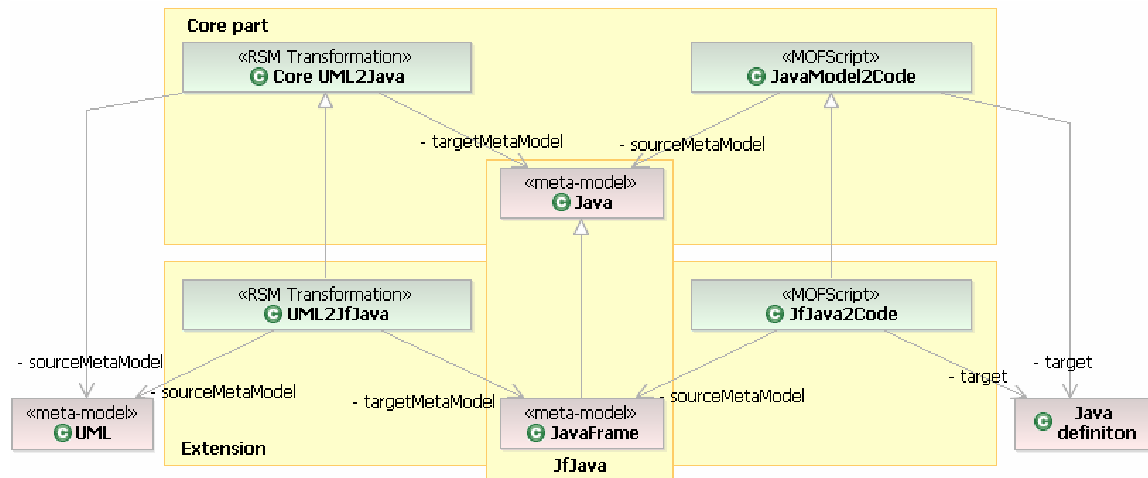

Figure 13 JfJava transformation architecture

Because of limited time the MOFScript transformation was not implemented completely. The transformation would have been very similar to the MOFScript transformation from JfUml (which is fully implemented). The RSM transformation was not implemented in two parts for the same reason, however how to extend RSM transformations is described in [25].

As explained in section 2.2.5 the RSM transformation framework is a Java framework for creating transformations. A transformation consists of a set of rules which is represented by Java classes that extends the AbstracRule framework class. To implement a rule the code directly manipulates the meta-model API generated by EMF, in this case this is the generated UML 2.0 API. This approach is much more verbose than using a dedicated transformation language such as MTF, but it also gives a larger degree of control and expressiveness to the transformation. An excerpt from the rule used to transform a UML property to a Java field is given below.

```
protected Object createTarget(ITransformContext ruleContext)
        throws Exception {
    Property source = (Property) ruleContext.getSource();
    Field target = createTargetObject();
    if (ruleContext.getTargetContainer() instanceof Class) {
        Class owner = (Class) ruleContext.getTargetContainer();
        setTargetContainer(target, owner);
    }
```

```
    if (source.getType() instanceof Classifier) {
        // adding import to owning compilation unit
        CompilationUnit cu = getCompilationUnit(target);
        TypeImport typeImport = JmmFactory.eINSTANCE.createTypeImport();
        typeImport.setCompilationUnit(cu);
        References.setFeatureUnknownValue(typeImport, JmmPackage.eINSTANCE
                .getTypeImport_ImportedType(), source.getType());
    }

    updateTarget(source, target);
    References.mapReference(source, target);
    return target;
}
```

Inheritance between rules is possible and I did implement a small inheritance hierarchy of rules, but this requires much manual preparation and is not nearly as convenient as extending MTF rules.

Some of the central classes in the RSM transformation are shown in Appendix B.

5.4.4 Generating meta-models from BNF

In [22] Fischer et al describes a method for developing meta-models from BNF [26]. Since I was making a meta-model for the Java language I examined the possibility of using this method to create the Java meta-model. In the end I decided against using this method for reasons I will explain later.

The method includes two steps. The first step is to generate a primitive meta-model from the BNF definition. In the second step advanced meta-model concepts like generalization, structural composition, and general abstract concepts are added manually. This manual step is needed because meta-models are more expressive than grammars and those concepts cannot be generated from the BNF definition.

Two things have to be manually provided for the second step of the process. The first is a model of abstract concepts used by the language and the second is information about which concrete language concept is a refinement of which abstract concept. Figure 14 shows a simplified activity diagram of the process.

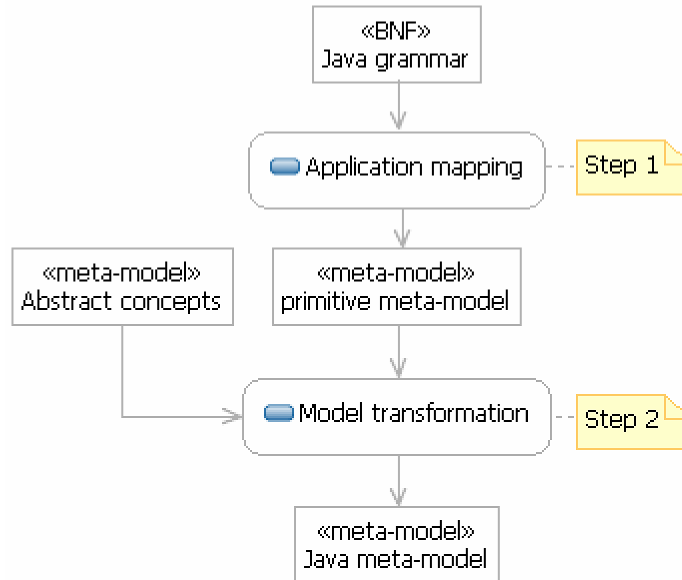

Figure 14 A simplified model showing the process of creating a meta-model from BNF

If this process is used on several programming languages, common abstract concepts may be identified and a shared base of concept definitions may be used for related languages. Such a language alignment would be helpful when creating transformations between different modeling languages and programming languages.

The application mapping (step 1) is done by a tool called *agramm* and the model transformation is implemented in Java using the *mmm* API, which is an API for model transformation based on JMI[*].

I will present an example of a very small BNF grammar, the transformation and the meta-model produced from it. The grammar is shown below. Note that this grammar does not describe any actual syntax (i.e., there are no terminals in this grammar).

```
Name :: TOKEN;

Class_name :: Name;
Import_name :: Name;
Package_name :: Name;
Field_name :: Name;

Identifier :: Package_name* Name;
Package_identifier :: Identifier;
Class_identifier :: Identifier;

CompilationUnit :: PackageDecl ImportDecl* ClassDecl;
PackageDecl :: Package_identifier;
ImportDecl :: Package_identifier | Class_identifier;
ClassDecl :: Class_name Field*;
Field :: Class_identifier Field_name;
```

[*] Java Metadata Interface. Sun's implementation of MOF.

The BNF is run through the *agramm* tool and a primitive meta-model is produced, this completes step 1. For step 2 I need a model of abstract concepts and I need to implement a transformation that links the abstract concepts to the concrete concepts of my language. I will reuse the common abstract concepts used by Fischer et al. The transformation was written manually, and an excerpt from it is presented below.

```
// CompilationUnit
compilationUnit.setContainer(structure);
compilationUnit.setMetaSupertype(getAbstractModelElements().getNamespace());
compilationUnit.addContainedType(packageDeclaration.getBase());
compilationUnit.addContainedType(importDeclaration.getBase());
compilationUnit.addContainedType(classDeclaration.getBase());

 // ClassDecl
 classDeclaration.setContainer(structure);
 classDeclaration.setMetaSupertype(getAbstractModelElements().getNamespace());
 classDeclaration.setMetaSupertype(getAbstractModelElements().getNamedElement());
 classDeclaration.setIsNamedElement();
 classDeclaration.addContainedType(field.getBase());
```

The final produced meta-model is shown in Figure 15. The four elements in the Common package are reused abstract elements.

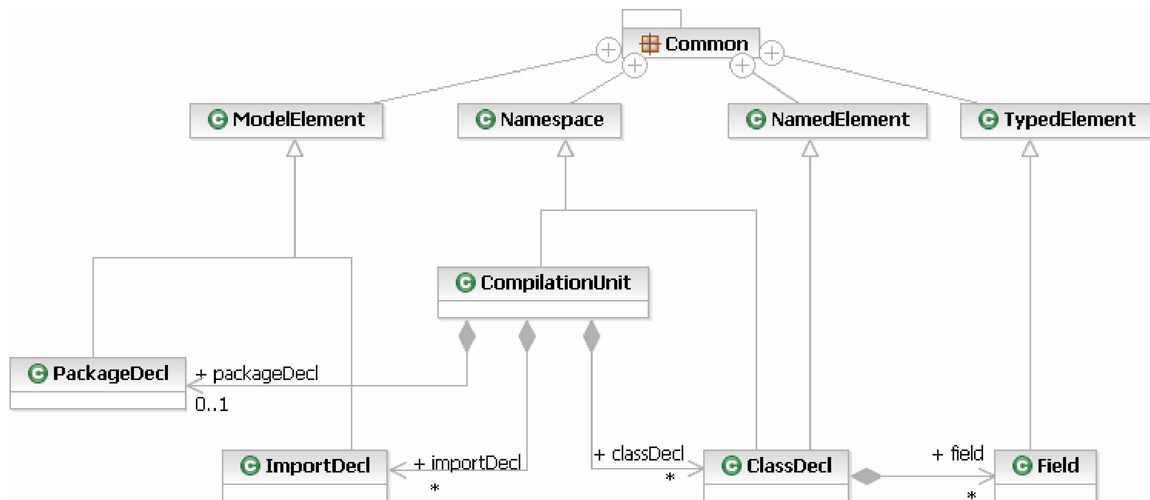

Figure 15 Meta-model produces from BNF

As I said in the beginning I chose not to use this method, the first reason for this is that I would have to modify the existing Java BNF. By modifying it I loose some of the advantages of this approach; that it should be more automatic and less error prone. There are several reasons I need to modify the BNF, the first is that the structure of the BNF and the desired structure of the meta-model do not always match. For example the CompilationUnit production of the grammar states that the CompilationUnit element should have a composite association to PackageDecl. A better meta-model structure would be to have a Package element with a composite association to CompilationUnit. This problem is explained in section 5.4.1. Two other less important reasons I needed to modify the BNF is that I was only creating a meta-model of a subset of the Java language

and I would have needed to remove superfluous parts of the BNF. And I would have needed to remove all terminals to make the grammar work with *agramm*.

The second reason is that the coding of the model transformation would be time consuming and error prone, partly because the BNF needed to be modified and partly because another difference between the grammar and the meta-model. The grammar needs to have several identifiers (e.g., Class_identifier) because textually this is the only way to identify elements. These identifiers are not needed in meta-models, direct associations to the correct element is used instead. This change needed to be manually implemented in the transformation. Fischer et al discuss this problem and conclude that mostly it can be solved by modeling concepts that would use identifier in the grammar as abstract concepts, thus removing the problem. However in Java some concrete elements use identifiers, for instance ImportDecl. Note that the grammar I have shown includes some identifiers, but these have not been implemented in the transformation.

The third reason is that the output meta-model was in the form of a MOF model. This was a problem because no good MOF to EMF transformation tool was found. In addition the generated MOF meta-model was structured in packages, the EMF code generator treats each package as a different meta-model, though this is a minor point.

For all those reasons and because I was only going to create a meta-model of a subset of the Java language, making the meta-model relatively simple, I chose to model it using UML 2.0 and RSM instead. If I was making a meta-model of the complete Java language I would not have to modify the BNF as much and the benefit of the generation would have been bigger. In that case this approach might have been worth using.

5.5 JfUml intermediate meta-model

The JfUml meta-model consists of a subset of the UML 2.0 meta-model and a JavaFrame extension to the UML 2.0 meta-model. In this section I explain the meta-model and the transformations implemented.

As explained in section 5.2.2 MOFScript does not support input meta-models in several files, this means that no UML 2.0 elements are available from the MOFScript transformations. The JfJava intermediate meta-model solved this by simply defining the two parts of the meta-model as one, but as this is not an option for this meta-model I solved it by creating elements that extend the needed UML 2.0 elements. These elements are shown in Figure 16. The metaclass stereotype shows that the element is from the UML 2.0 meta-model.

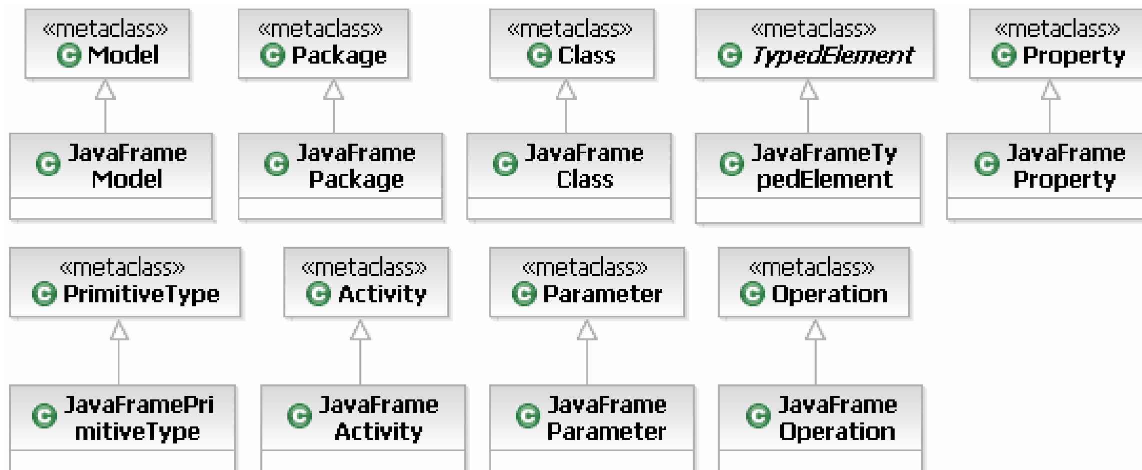

Figure 16 JfUml adaptations of UML 2.0 elements

Figure 17 shows a JavaFrame meta-model that extends the UML meta-model. It is very similar to the JavaFrame extension of JfJava. The differences are partly due to limitations of the MTF transformation language and partly due to just trying different approaches.

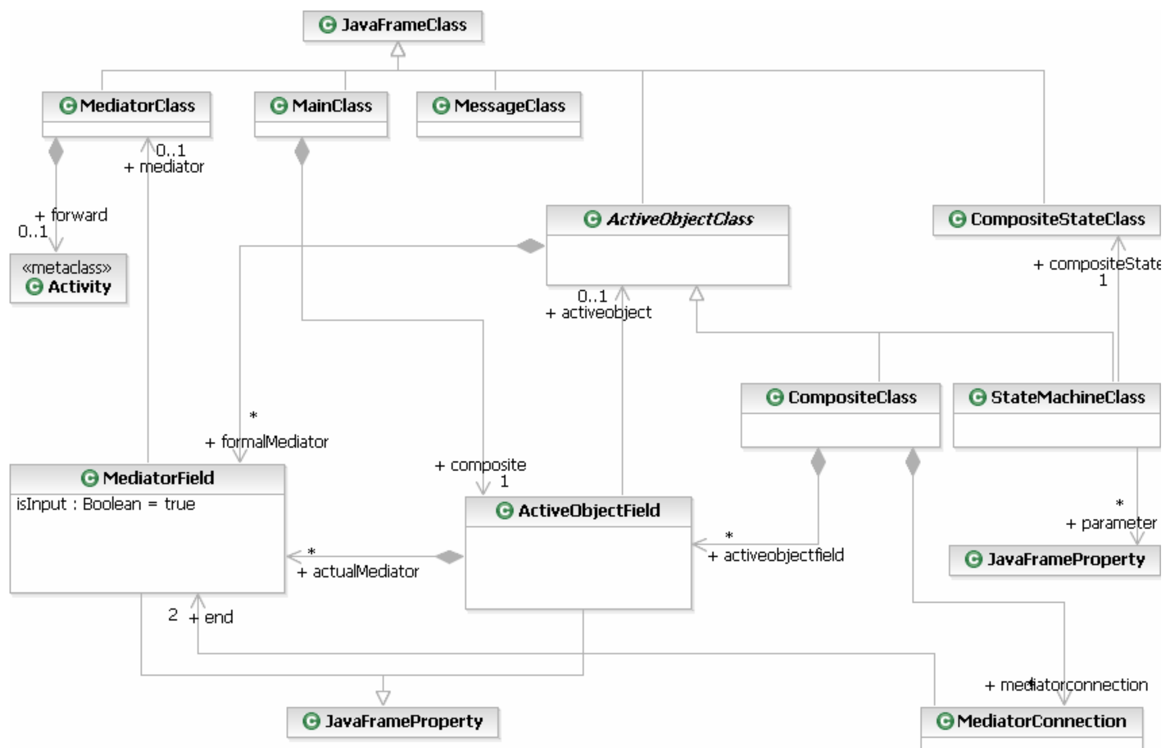

Figure 17 A JavaFrame meta-model that extends classes from the UML meta-model

An example of a difference is the MediatorConnection element and its association to MediatorField called end. This would have been better represented by two associations called source and target. This was not possible because this is represented as a single association in the UML meta-model and because MTF has no way of distinguishing between the order of elements in an association.

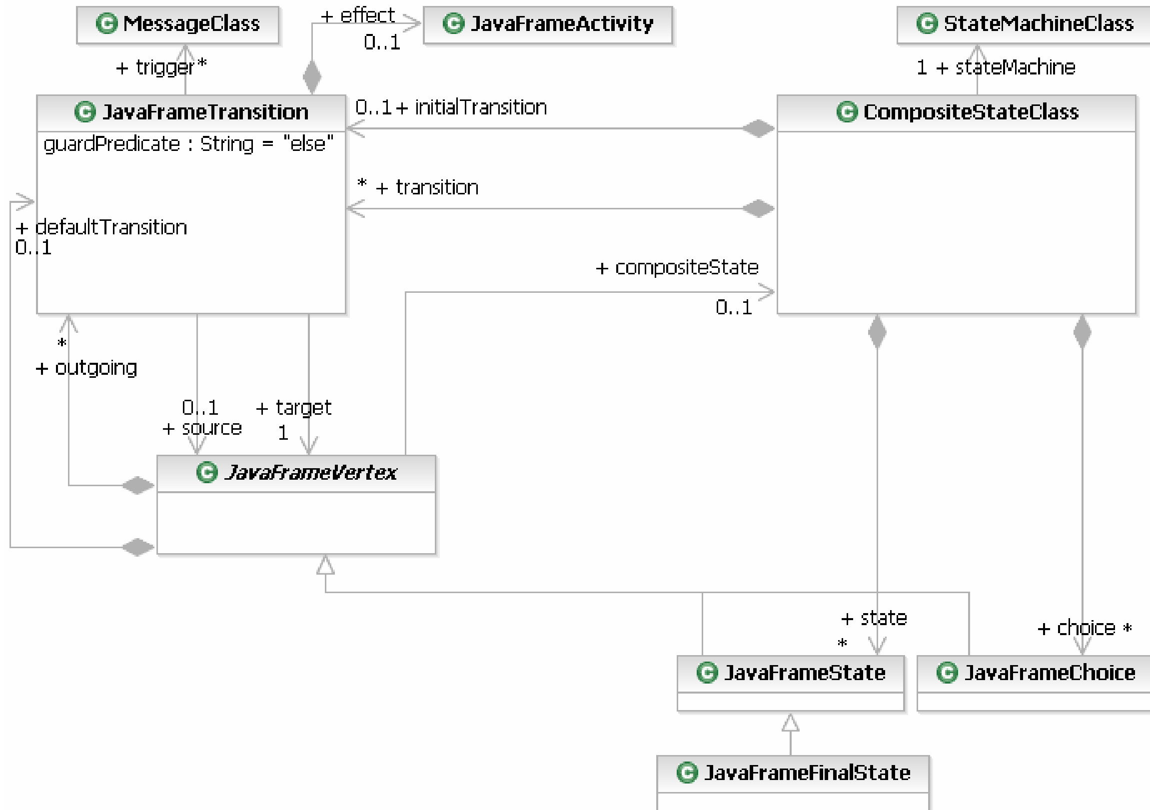

Figure 18 CompositeStateClass and related elements

Figure 18 shows the CompositeStateClass and related elements, this almost the same as the equivalent diagram in JfJava. The difference is that this meta-model do not have an InitialState class, instead an initialTransition composite association from CompositeStateClass to Transition represents this.

5.5.1 Transformations

The general transformation architecture was adapted to implement the transformations needed for this intermediate meta-model. A difference from the general scenario is that UML is both the source meta-model and the core part of the intermediate meta-model. This has the consequence that the model-to-model transformations use UML as both input and output and a UMLCopy transformation was needed. Figure 19 shows the adapted transformation architecture. For this scenario MTF was chosen as model-to-model transformation technology.

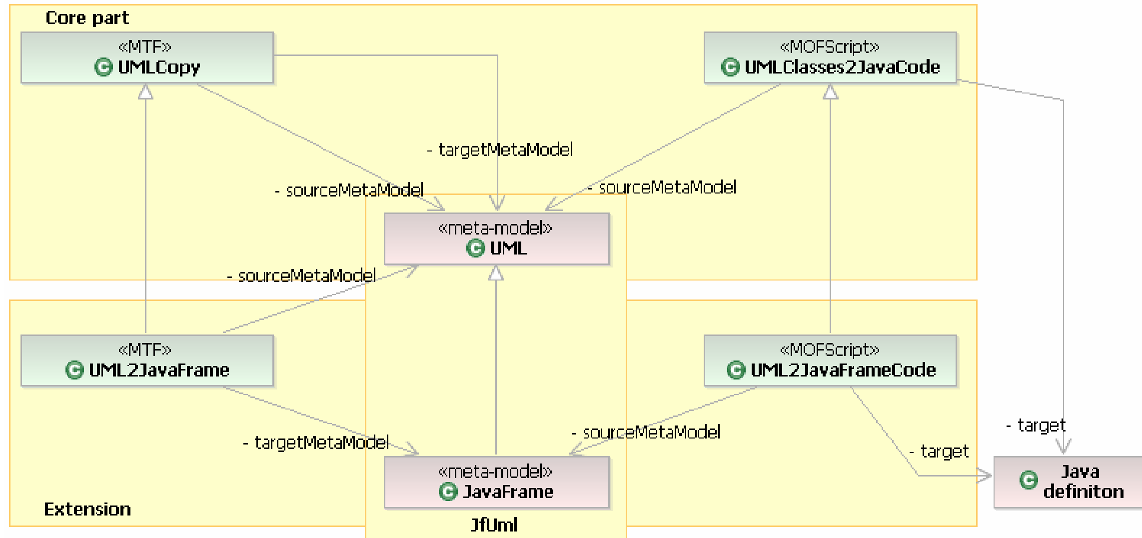

Figure 19 Transformations using a JavaFrame intermediate meta-model that extends the UML meta-model.

The UMLCopy transformation has rules for copying, among other things, classes, attributes and operations. This is not useful by itself, but it allows for these rules to be reused by any transformation that extend UMLCopy. Below is an example of a rule from UMLCopy, it copies UML Property elements. This rule is then reused by extending it in the UML2JavaFrame transformation.

.

```
relate mapProperty extends mapTypedElement(uml:Property e1, uml:Property e2) {
        equals(e1.visibility, e2.visibility)
        ,equals(e1.aggregation, e2.aggregation)
        ,equals(e1.default, e2.default)
        ,equals(e1.isStatic, e2.isStatic)
        ,equals(e1.isUnique, e2.isUnique)
        ,equals(e1.isLeaf, e2.isLeaf)
        ,mapLiteralSpecification(over e1.lowerValue, over e2.lowerValue)
        ,mapLiteralSpecification(over e1.upperValue, over e2.upperValue)
}
```

The next rule is a UML2JavaFrame rule that transforms ports to mediatorfield elements. Since both Port and MediatorField are subtypes of Property, this rule can extend mapProperty from UMLCopy.

```
relate mapMediatorField extends mapProperty(uml:Port e1, jf:MediatorField e2) {
        equals(e1.isBehavior, e2.isInput)
}
```

The UMLCopy transformation might have been generated by a transformation that takes any meta-model as input and outputs code for copying that meta-model. Such a transformation is defined with M3 level meta-models as source and is run with M2 level meta-models as input, as opposed to the normal transformations that are defined with M2 level meta-models as source and are run with M1 level models as input.

For example I could have created a MOFScript transformation with Ecore meta-models as source to produce a MTF transformation that copies any model that conforms to the

input meta-model. I would then need to run the transformation with a UML Ecore meta-model as input and a MTF transformation that copies UML models would have been generated. Dennis Wagelaar [27] has created such a transformation using ATL as a transformation language and MOF as source meta-model. This transformation generates copy transformations defined in ATL.

As explained in section 5.2.2 MOFScript does not currently support transformation inheritance sufficiently so UMLClasses2JavaCode and UML2JavaFrameCode was implemented in one transformation called JfUml2Java. The three transformation files are shown in Appendix A.

The core transformations may be reused to generate any Java code, not just code based on JavaFrame. Also not all code generation scenarios warrant the use of an intermediate meta-model, but even if no intermediate meta-model is used the UMLClasses2JavaCode transformation may be useful to extend.

5.6 Comparing the JfUml to JfJava approaches

Even though the JfUml and JfJava intermediate meta-models started from two different technologies, one from the Java programming language and the other from a subset of UML, in the end they turned out to be quite similar. This is mostly because the core parts only used structural concepts common to most object oriented systems. No behavior was represented there. That was added in the JavaFrame extension, and that was going to represent the same thing in both approaches.

One difference is the use of a CompilationUnit class in the JfJava meta-model, UML has no such concept. Tough this is a minor difference.

The bigger differences in the intermediate meta-models were how they were created. Creating an extension to the UML meta-model, and getting the code-generator to generate correct code proved a technical challenge. The reuse of existing UML elements made JfUml faster to create and gave it access to a lot of well thought out elements from UML meta-model. However if some of those elements are better suited to be represented differently in the intermediate meta-model, it can prove a problem.

The big difference regarding transformations is that JfUml can generate the UMLCopy transformation. This has a potential to save time and to create a transformation that is error free and well suited for extension.

The JfUml and JfJava intermediate meta-models are similar in level of abstraction. It would be interesting to see how they compared with intermediate meta-models that are closer to the code. Two good candidates for this are a complete Java meta-model and a subset of the UML meta-model with a platform independent action language.

6 Customizing the generated code

The previous chapter described the architecture and implementation of a transformation from UML to JavaFrame. The transformations were implemented using two different intermediate meta-models. In this chapter I look at how, given such a transformation, a user can customize the generated code.

First different user roles involved in implementing and using transformations is identified. Second a classification of customization alternatives is given and examples of scenarios with partial implementations are shown. Finally how the transformation architecture can benefit maintenance is discussed.

6.1 User roles

The different user roles involved in the implementation of and use of transformations are:

- Transformation designer / implementer
- Expert user / transformation extender
- Transformation user

Transformation designer is the role I have been addressing so far in this thesis. The transformation designer creates the intermediate meta-models and implements transformations. He may also be involved in creating extensions to existing transformations.

The expert user is a user that not only uses the transformations to generate code, but also customizes the generated code.

The transformation user creates models and runs the transformation(s). This thesis does not discuss this user role.

This chapter describes how generated code can be customized and so the relevant user roles are transformation designer and expert user.

6.2 Classification of changes

This section gives a short description of different types of possible changes to existing transformations. A short outline of how the transformation architecture

6.2.1 Supporting more of UML

The existing transformation from UML to JavaFrame transforms composite structures, statemachines and classes to Java code. A way of extending this transformation is to support more of UML. Examples of this can be activity diagrams, deployment diagrams and interactions.

To implement such additions, the added source concepts must be represented in the intermediate meta-model. Either the existing intermediate meta-model may be reused or it must be extended to include the new concepts.

6.2.2 Supporting extensions to UML

The transformation can be extended to support extensions to UML. The UML extensions are usually in the form of a profile. The effect on the transformations of this is very similar to the last point about supporting more of UML.

6.2.3 Model checking

It may be desirable to not only generate code for running the model normally, but also to run model checks. There are several ways to do this; it is for example possible to check a running statemachine against a sequence diagram specification to verify that the statemachine behaves according to the specification.

This would probably require change in the input, to specify which statemachine to verify against which sequence diagram, and some change in the generated code, in other words the intermediate meta-model needs to be extended.

6.2.4 Quality of Service (QoS) change

QoS changes is a large group of changes, some may require an intermediate meta-model change, but a lot will probably only need a model-to-text extension. In section 6.3.3 I show an implementation of an example of a QoS change; adding memory optimization to signal classes.

6.2.5 Platform change

A possible change is to generate code for a different platform. Depending on the level of the intermediate meta-model this may require an intermediate meta-model change; if it does a lot needs to be changed. If not only the model-to-text transformations need to be replaced.

6.2.6 Different semantic variation point implementation

One or more semantic variation points may be implemented differently. An example of this is the choice of which transition to fire when outgoing transitions from a single state has conflicting triggers. If a state has two outgoing transitions, one triggered by SignalA and the other triggered by SignalB and SignalB is a subtype of SignalA, which transition should be triggered if a SignalA object is received? This is a semantic variation point in UML 2.0 and a conflict resolutions mechanism is needed. The default JavaFrame behavior is to select a consistent arbitrary transition (based on order). Another solution would be to choose randomly each time such a conflict occurs. It could be argued that this is better for fairness purposes, but it is also less deterministic. This change could be implemented by extending the model-to-text transformation. However UML 2.0 has

many semantic variation points and some of them would probably require an intermediate meta-model change to implement.

In [19] Chauvel and Jézéquel discusses in detail how semantic variation points can be modeled and how different code can be generated for different solutions.

6.3 Examples

This section explains some specific examples of customization of the generated code. I use the JfUml transformations as a starting point and show what changes need to be made to implement the customizations.

6.3.1 Generating Tests for JavaFrame models

The UML 2.0 Testing profile is used to specify tests for the model and JavaFrame specific tests are generated.

This is an example of adding support for an extension to UML and an extension that requires a change in the intermediate meta-model. The changes are shown in Figure 20 where a new extension layer is added to the JfUml transformations. The extension layer consists of an intermediate meta-model extension and two transformation extensions.

Whenever an extension to the intermediate meta-model is required such an extensions layer needs to be added.

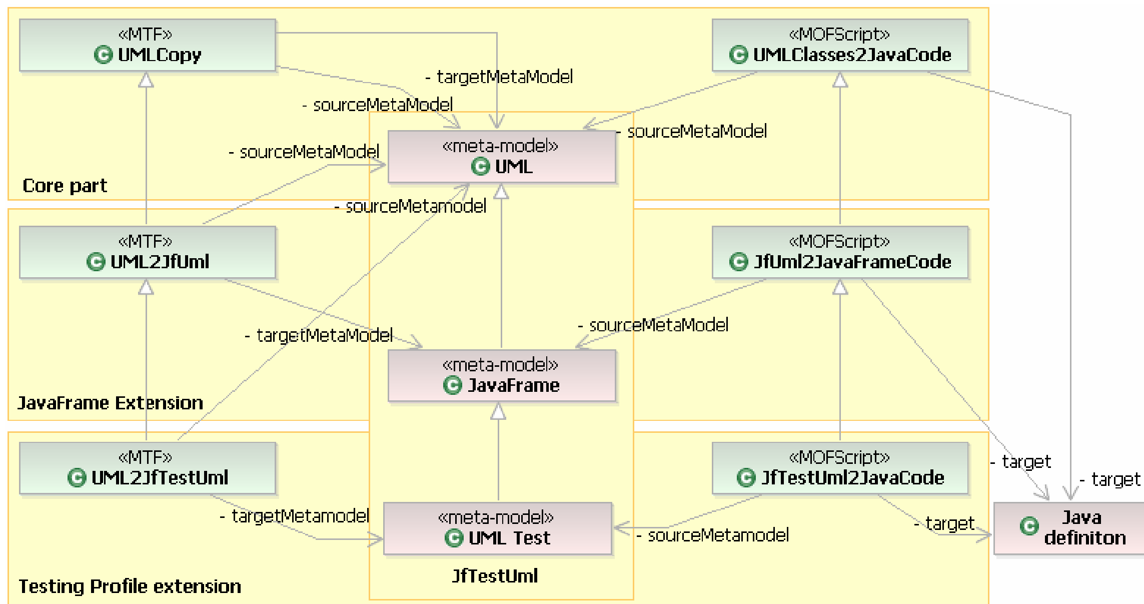

Figure 20 JfUml transformation architecture extended with support for UML Testing Profile

6.3.2 Generating C# code

If JavaFrame was ported to C#Frame there would be possible to generate C# code. This is an example of a platform change. The existing intermediate meta-model and model-to-model transformation could be reused but the model-to-text transformations would need

to be replaced. The modified transformation architecture of the JfUml transformations is shown in Figure 21.

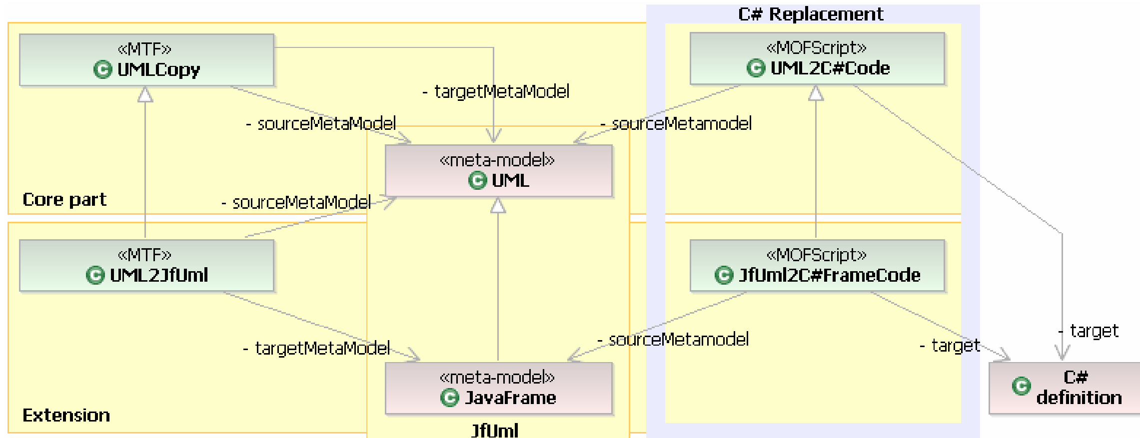

Figure 21 C#Frame Transformations

6.3.3 Adding memory optimization to signals

This change is a Quality of Service (QoS) change. It does not require an extension to the intermediate meta-model. Extending the model to text transformation is all that is required.

Figure 22 shows the JfUml transformation architecture with an additional MOFScript transformation. This transformation specializes the code generated from signal elements.

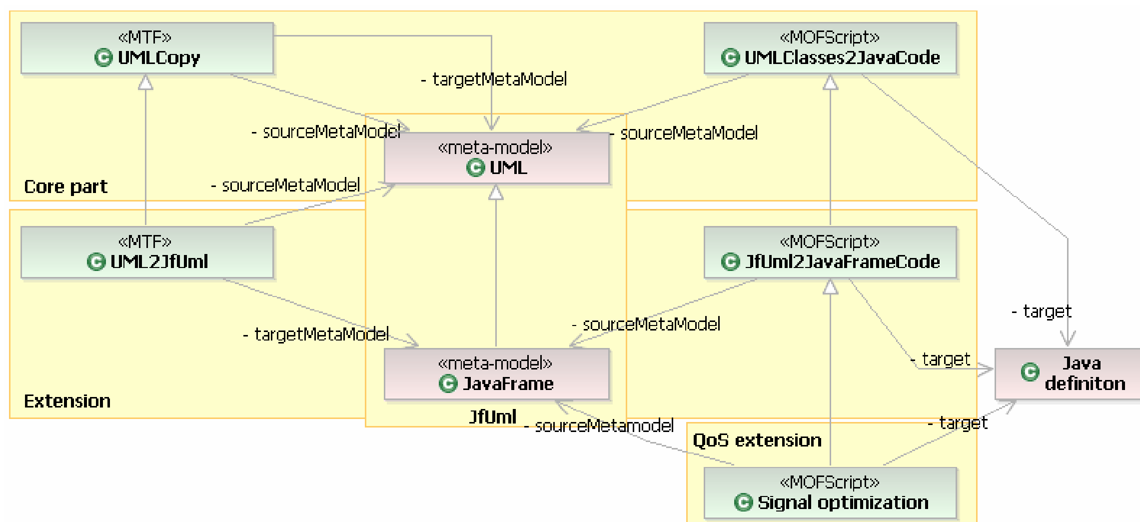

Figure 22 QoS Signal memory optimization

Methods for creation and destruction of signal objects is created and added to all signal classes. If these methods are used instead of the Java 'new' operator a lot of unnecessary object creation and garbage collection may be avoided.

To implement this change, four additional methods need to be generated for each signal class.

```
/* QoS extension: Add freelist to signals */
import "UmlJF2JavaFrame.m2t"

texttransformation SignalFreelist(in jf:"http:///JF.ecore") extends UmlJF2JavaFrame

jf.SignalClass::mapConstructor() {
	super.mapConstructor()
	var constructorPars = self.getConstructorParameterString()
	var constructorArgs = self.getConstructorArgString()
	newline
<%
	private static %>self.name<% %> self.name<%Freelist; // top of the freelist stack
	private %> self.name <% %>self.name<%Next; // the list pointer

	synchronized public static %>self.name<% New%>self.name<%() {
		// method contents …
	}

	synchronized public static %>self.name<%New%>self.name<%(%>constructorPars<%) {
		// method contents …
	}

	synchronized private static void %>self.name<%DelMsg (%>self.name<% sig) {
		// method contents …
	}

	public void del() {
		%>self.name<%DelMsg (this);
	}
%>
}
```

This transformation contains only one rule: mapConstructor. This rule overrides a rule defined in UmlJF2JavaFrame which generates a constructor for the class. This have the effect of replacing the original rule with this one, but because I still want to generate a constructor for the class I call super.mapConstructor() which will call the original rule.

Note that this extension does not currently work as it is shown here due to a bug in the inheritance mechanism of MOFScript. To run the transformation I implemented all the three MOFScript transformations as one transformation. This transformation is shown in Appendix A.

6.3.4 Reusing the core

If a transformation from UML to Java for something that is unrelated to JavaFrame needs to be implemented, the core part of the transformation architecture may still be reused. Figure 23 shows a transformation were the JavaFrame extension layer has been replaced by a GWT[*] extension layer.

[*] Google Web Toolkit, a framework for developing web-applications in Java.

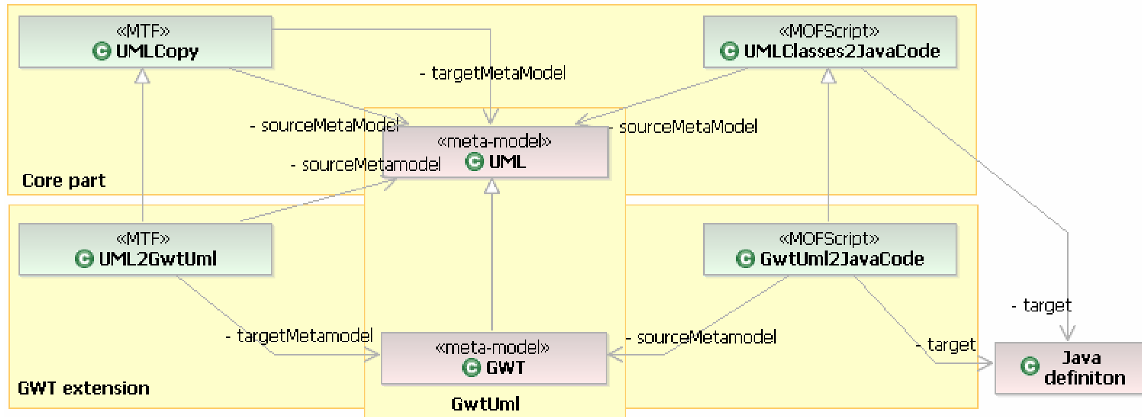

Figure 23 GWT extension, reusing the core transformation layer

6.4 Maintenance

Change is inevitable and transformations must prepare for it. The reasons for changing transformations may be new demands, bugs, and changes in the source or target language.

By separating the source from the target with an intermediate meta-model the consequences of a change in the source or target language is limited. In the case of a change from UML version 2.0 to 2.1, only the model-to-model transformations need changing. Although the JfUml meta-model uses a subset of UML this is probably unlikely to change much, as this subset (classes, attributes, etc.) are well established concepts.

Changes in the transformation that do not require an intermediate meta-model change can probably be limited to changing one of the four transformations. However if a change of the intermediate meta-model is needed, at least two transformations must be changed (one model-to-model and one model-to-text).

Changes performed in a super-transformation may require changes in any sub-transformations, although if none of the signatures of the rules in the super-transformations are changed, the change can be limited to only that transformation.

7 Summary and conclusions

The background for this thesis was an existing UML compiler producing JavaFrame code from UML 2.0 models annotated with JavaFrame-specific stereotypes. The compiler was written as a single transformation using Java and the RSM transformation framework.

The main problem was how transformations could be made more user-friendly in the sense that there would be easier to customize the generated code.

To answer this I developed a general transformation architecture that uses an intermediate meta-model in two parts: one core meta-model and one extension. The use of an intermediate meta-model allowed the transformation to be split in a model-to-model transformation and a model-to-text transformation. And because the intermediate meta-model was defined in two parts, both of these transformations could be split in a core and an extended part.

Different ways of creating intermediate meta-models were presented and analyzed, and two intermediate meta-models were created. The first intermediate meta-model used a Java meta-model as a core part and the second used a subset of the UML meta-model. Both used a JavaFrame meta-model as an extension. Even though the two intermediate meta-models had different starting points they turned out to be quite similar.

The general transformation architecture was adapted to the specific scenarios and transformations were implemented using MTF, the RSM transformation framework, and MOFScript.

Using the transformation architecture I showed how the generated code could be customized either by extending the transformation or by replacing parts of it. The fact that the transformation architecture consists of several small transformation, that transformation inheritance is used, and that the intermediate meta-model provide a separation layer between the source and the target allow for these customization and replacements to be implemented while reusing large parts of the existing transformation.

7.1 Future work

When choosing intermediate meta-models I discussed using a subset of UML or a Java meta-model as an intermediate meta-model and I identified problems with intermediate meta-models this close to the code. Trying to overcome these problems and implementing transformations for such an intermediate meta-model would be an interesting experiment.

The major obstacle for developing large transformations is the immaturity of current transformation languages. The developed transformation architecture had a set of requirements for model transformation languages, these requirements were to demanding for most transformation languages. A lot of work remains implementing transformation languages that are suited for modular development.

8 References

1. S. Savenko, *Combined PIM-PSM*, in *Institute of Informatics*. 2004, University of Oslo. p. 135.
2. OMG, *Meta Object Facility 1.4, OMG Document (formal/02-04-03)*. 2005.
3. *Introducing EMF, Eclipse Foundation*. [Slideshow presentation] 2004 [cited 14.3.2006]; Available from: http://www.awprofessional.com/content/images/0131425420/samplechapter/budinskych02.pdf.
4. M. Paternostro and K. Hussey. *Advanced Features of the Eclipse Modeling Framework*. EclipseCon [Slideshow presentation] 2006 [cited 10.5.2006]; Available from: http://eclipse.org/emf/docs/presentations/EclipseCon.
5. OMG. *Unified Modeling Language (UML)*. [Webpage] [cited 13.5.2006]; Available from: http://www.uml.org/.
6. OMG, *UML Superstructure Specification, OMG Document (formal/05-07-04)*. 2005.
7. Wikipedia. *Unified Modeling Language*. [Webpage] [cited 13.5.2006]; Available from: http://en.wikipedia.org/wiki/Unified_Modeling_Language.
8. T. Mens, K. Czarnecki, and P.V. Gorp, *A Taxonomy of Model Transformations.* Dagstuhl Seminar on Language Engineering for Model-Driven Software Development, 2005.
9. K. Czarnecki and S. Helson, *Classification of Model Tranformation Approaches.* OOPSLA’03 Workshop on Generative Techniques in the Context of MDA, 2003.
10. OMG, *MOF QVT Final Adopted Specification (ptc/05-11-01).* 2005.
11. F. Jouault and I. Kurtev. *Transforming models with ATL*. in *Proceedings of the Model Transformations in Practice Workshop*. 2005. Jamaica.
12. IBM. *Model Transformation Framework*. [Application] [cited 21.5.2006]; Available from: http://www.alphaworks.ibm.com/tech/mtf.
13. *MOFScript*. [Transformation language] [cited 21.5.2006]; Available from: http://www.eclipse.org/gmt/mofscript/.
14. G.v.E. Boas, *Template Programming for Model-Driven Code Generation.* 2004: p. 17.

15. I. Kurtev, K.v.d. Berg, and F. Jouault, *Evaluation of Rule-based Modularization in Model Transformation Languages illustrated with ATL (preliminary version).* 2006.

16. J. Oldevik, *Transformation Composition Modeling Framework.* 2005: p. 7.

17. K.C. Louden, *Compiler Construction: Principles and Practice*. 1997: PWS Publishing Company.

18. B. Vanhooff and Y. Berbers, *Breaking Up the Transformation Chain.* 2005: p. 5.

19. F. Chauvel and J.-M. Jézéquel, *Code generation from UML Models with semantic variation points.* 2005: p. 15.

20. J. Rumbaugh, I. Jacobsen, and G. Booch, *The Unified Modelling Language Reference Manual, Second edition*. 2004: Addison-Wesley.

21. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, *Weaving Executability into Object-Oriented Meta-Languages*, in *Proceedings of UML MoDELs*. 2005, Springer Verlag: Jamaica.

22. J. Fischer, M. Piefel, and M. Scheidgen, *A Metamodel for SDL-2000 in the Context of Metamodeling ULF.* 4th SDL and MSC Workshop, 2004.

23. QVT-Merge_Group, *Revised submission for MOF 2.0 Query/view/Transformation RFP (ad/05-03-02)*. 2005, OMG.

24. Sun. *NetBeans Source*. [Webpage] [cited 16.5.2006]; Available from: http://www.netbeans.org/community/sources/.

25. D. Ruest. *Extending the UML to Java transformation with Rational Software Architect: An example explained*. [cited 21.5.2006]; Available from: http://www-128.ibm.com/developerworks/rational/library/05/802_uml/.

26. L.M. Garshol. *BNF and EBNF: What Are They and How Do They Work?* [cited 20.5.2006]; Available from: http://www.garshol.priv.no/download/text/bnf.html.

27. D. Wagelaar. *MDE Case Studies*. [Webpage] [cited 15.5.2006]; [Vrije Universiteit Brusse, System and Software Engineering Lab.] Available from: http://ssel.vub.ac.be/ssel/research:mdd:casestudies.

Appendix A - JfUml Transformations

UMLCopy.rdl (MTF v 1.1)

```
import uml "http:///com/ibm/mtf/uml.ecore"
import uml "http://www.eclipse.org/uml2/1.0.0/UML"
import util "http:///com/ibm/mtf/util.ecore"
import ecore "http://www.eclipse.org/emf/2002/Ecore"
import emf "http:///com/ibm/mtf/model/emf.ecore"
import ws "http:///com/ibm/mtf/model/workspace.ecore"

relate File(ws:IFile file1, ws:IFile file2){
    mapModel(
        over file1.resource.contents,
        over file2.resource.contents)
}

relate mapModel (uml:Model m1, uml:Model m2) {
    equals(m1.name, m2.name)
    ,equals(over m1.profiles, over m2.profiles)
    ,ordered mapElement(over m1.ownedMember, over m2.ownedMember)
}

relate mapElement(uml:Element e1, uml:Element e2) {
    //equals(over e1.stereotypes, over e2.stereotypes)
}

abstract relate mapNamedElement extends mapElement(
    uml:NamedElement e1,
    uml:NamedElement e2)
    when equals(e1.name, e2.name) {}

relate mapPackageImport(uml:PackageImport e1, uml:PackageImport e2)
{
    mapLocalPackageImport(e1, e2),
    mapReferencePackageImport(e1, e2)
}

relate mapLocalPackageImport /*extends
mapPackageImport*/(uml:PackageImport e1, uml:PackageImport e2)
    when ref mapPackage(e1.importedPackage, e2.importedPackage)
{
    ref mapPackage(e1.importedPackage, e2.importedPackage)
}

relate mapReferencePackageImport /*extends
mapPackageImport*/(uml:PackageImport e1, uml:PackageImport e2)
    when !equals(e1.importedPackage.model, e1.model)
{
    equals(e1.importedPackage, e2.importedPackage)
}
```

```
relate stereotypeByName(uml:Stereotype e1, ecore:EString name)
    when equals(e1.name, name)

/*
* Should seperate between types that are part of the model and types
that
* are inter-model references. Types that are part of the model should
* link to the new element in the target model, types that are
* inter-model references should link to the same element in the
* referenced model.
*/
abstract relate mapTypedElement extends mapNamedElement(
    uml:TypedElement e1,
    uml:TypedElement e2) {
    mapTypedElementType(e1,e2)
}


abstract relate mapTypedElementType(
    uml:TypedElement e1, uml:TypedElement e2)

relate mapTypedElementPrimitiveType extends mapTypedElementType(
    uml:TypedElement e1 ,
    uml:TypedElement e2)
        when util:InstanceOf "uml:PrimitiveType" (e1.type) {
    equals(e1.type, e2.type)
}

relate mapTypedElementClassType extends mapTypedElementType(
    uml:TypedElement e1 when util:InstanceOf"uml:Class" (e1.type),
    uml:TypedElement e2) {
    ref mapClass(e1.type, e2.type)
}

relate mapTypedElementNullType extends mapTypedElementType(
    uml:TypedElement e1,
    uml:TypedElement e2) when !util:InstanceOf "uml:Element" (e1.type)

relate mapPackage extends mapNamedElement(
    uml:Package e1, uml:Package e2) {
    mappedPackage(e1),
    ordered mapClassifier(over e1.ownedMember, over e2.ownedMember)
    ,ordered mapPackage(over e1.ownedMember, over e2.ownedMember)
}

relate mappedPackage(uml:Package e1)

abstract relate mapClassifier extends mapNamedElement(
        uml:Classifier e1,
        uml:Classifier e2) {
    mapGeneralization [0..1](
        over e1.generalization,
        over e2.generalization)
    ,ordered mapPackageImport(over e1.packageImport, over
e2.packageImport)
}
```

```
relate mapClass extends mapClassifier(uml:Class e1, uml:Class e2) {
    ordered mapProperty(over e1.ownedAttribute, over e2.ownedAttribute),
    ordered mapOperation(
        over e1.ownedOperation,
        over e2.ownedOperation)
    ,ordered mapActivity(over e1.ownedBehavior, over e2.ownedBehavior)
}

relate mapGeneralization extends mapElement(
    uml:Generalization e1,
    uml:Generalization e2) {
    ref mapClassifier [1](over e1.general, over e2.general)
}

relate mapProperty extends mapTypedElement(uml:Property e1,
uml:Property e2) {
    equals(e1.visibility, e2.visibility)
    ,equals(e1.aggregation, e2.aggregation)
    ,equals(e1.default, e2.default)
    ,equals(e1.isStatic, e2.isStatic)
    ,equals(e1.isUnique, e2.isUnique)
    ,equals(e1.isLeaf, e2.isLeaf)
    ,mapLiteralSpecification(  over e1.lowerValue, over e2.lowerValue)
    ,mapLiteralSpecification(  over e1.upperValue, over e2.upperValue)
}

/*
relate mapMultiplicityElement extends mapElement(
    uml:MultiplicityElement e1, uml:MultiplicityElement e2) {
    mapLiteralSpecification(
        over e1.lowerValue, over e2.lowerValue)
    ,mapLiteralSpecification(
        over e1.upperValue, over e2.upperValue)
}
*/

abstract relate mapLiteralSpecification extends mapElement (
    uml:LiteralSpecification e1, uml:LiteralSpecification e2)

relate mapLiteralInteger extends mapLiteralSpecification(
    uml:LiteralInteger e1, uml:LiteralInteger e2)
{
    equals(e1.value, e2.value)
}

relate mapLiteralUnlimitedNatural extends mapLiteralSpecification(
    uml:LiteralUnlimitedNatural e1, uml:LiteralUnlimitedNatural e2)
{
    equals(e1.value, e2.value)
}

relate mapLiteralString extends mapLiteralSpecification(
    uml:LiteralString e1, uml:LiteralString e2)
{
    equals(e1.value, e2.value)
}
```

```
relate mapLiteralBoolean extends mapLiteralSpecification(
    uml:LiteralBoolean e1, uml:LiteralBoolean e2)
{
    equals(e1.value, e2.value)
}


relate mapOperation extends mapTypedElement(
    uml:Operation e1,
    uml:Operation e2) {
    equals(e1.isLeaf, e2.isLeaf)
    ,equals(e1.isStatic, e2.isStatic)
    ,mapParameter [0..1] (over e1.returnResult, over e2.returnResult)
    ,ordered mapParameter (over e1.ownedParameter, over
e2.ownedParameter)
    ,ref mapActivity [0..1] (over e1.method, over e2.method)
}

relate mapParameter extends mapTypedElement (
    uml:Parameter e1,
    uml:Parameter e2) {
    equals(e1.visibility, e2.visibility)
    ,equals(e1.default, e2.default)
    ,equals(e1.isUnique, e2.isUnique)
    ,equals(e1.lower, e2.lower)
    ,equals(e1.upper, e2.upper)
    ,equals(e1.direction, e2.direction)
    ,ref mapActivity(e1.effect, e2.effect)
}

relate mapActivity extends mapNamedElement(
    uml:Activity e1,
    uml:Activity e2) {
    mapActivityEdge(over e1.edge, over e2.edge)
    ,mapNode(over e1.node, over e2.node)
}

abstract relate mapNode extends mapNamedElement(uml:ActivityNode e1,
        uml:ActivityNode e2) {}

relate mapInitial extends mapNode(
    uml:InitialNode e1, uml:InitialNode e2)
relate mapAction extends mapNode(uml:Action e1, uml:Action e2)
relate mapFinal extends mapNode(
    uml:ActivityFinalNode e1, uml:ActivityFinalNode e2)

abstract relate mapActivityEdge extends mapNamedElement(
    uml:ActivityEdge e1, uml:ActivityEdge e2)
    when ref mapNode(e1.source, e2.source)
         & ref mapNode(e1.target, e2.target)

relate mapControlFlow extends mapActivityEdge(
    uml:ControlFlow e1, uml:ControlFlow e2)

relate mapObjectFlow extends mapActivityEdge(
    uml:ObjectFlow e1, uml:ObjectFlow e2)
```

UML2JavaFrame.rdl (MTF v 1.1)

```
import "UMLCopy.rdl"

import jf "http:///JF.ecore"

/*
 * The rules in this transformation extend rules defined in
 * UMLCopy.rdl
 */
abstract relate mapActiveObjectClass extends mapJavaFrameClass(
        uml:Class e1 when util:InstanceOf "uml:StateMachine" (e1)
        | stereotypeByName(match over e1.stereotypes, "Composite") ,
        jf:ActiveObjectClass e2) {
        //createSchedulerProperty [1] (match over e2.ownedAttribute),
        ordered mapMediatorField(over e1.ownedPort, over e2.formalMediator)
}

relate createSchedulerProperty(jf:JavaFrameProperty e1 when equals(e1.name, "sched")) {
        //equals(e1.type, "jf:Scheduler"),
        //equals(e1.name, "sched")
}

/* StateMachine and CompositeState */
relate mapStateMachine extends mapActiveObjectClass(
        uml:StateMachine e1,
        jf:StateMachineClass e2) {
        mapCompositeStateClass [1](over e1.region, over e2.package.ownedMember),
        ref mapCompositeStateClass(over e1.region, e2.compositeState)
}

relate mapCompositeStateClass(uml:Region e1, jf:CompositeStateClass e2)
        when util:MatchString "{0}_{1}" (e1.stateMachine.name, e1.name, e2.name) {
        ref mapStateMachine(e1.stateMachine, e2.stateMachine)
        ,ordered mapState(over e1.subvertex, over e2.state)
        ,ordered mapChoice(over e1.subvertex, over e2.choice)
        ,mapInitialTransition[0..1](over e1.subvertex, e2.initialTransition)
        //,ordered mapTransition(over e1.transition, over e2.transition)
}


/*
relate createCompositeStateClass(
        jf:CompositeStateClass e1,
        uml:Region e3,
        ecore:EString e2)
        when util:MatchString "{0}States" (e2, e1.name) {
        equals(e1.region, e3),
        ref mapStateMachine(e3.stateMachine, e1.stateMachine)
}
*/
```

```
abstract relate mapVertex(uml:Vertex e1, jf:JavaFrameVertex e2) {
	//ordered mapTransition(over e1.owner.ownedElement, over e2.outgoing, e1)
	ordered mapTransition(over e1.outgoing, over e2.outgoing)
}

relate mapState extends mapVertex(
		uml:State e1,
		jf:JavaFrameState e2) {
	equals(e2.visibility, "public")
	//,equals(e1.default, e2.default)
	//,equals(e1.isComposite, e2.isComposite)
	,equals(e2.isStatic, "true")
	//,equals(e2.isUnique, "false")
	,equals(e2.isLeaf, "false")
	,equals(e1.name, e2.name)
	//,mapDefaultTransition [0..1](over e1.owner.ownedElement, e2.defaultTransition, e1)
}


relate mapFinalState extends mapState(uml:FinalState e1, jf:JavaFrameFinalState e2) {
	//equals(e2.name, "finalState")
}

relate mapChoice extends mapVertex(
		uml:Pseudostate e1 when equals(e1.kind, "choice"),
		jf:JavaFrameChoice e2) {
	//ordered mapTransition(over e1.owner.ownedElement, over e2.outgoing, e1)
}

relate mapTransition(
	uml:Transition e1, jf:JavaFrameTransition e2) {
	//ref mapVertex(e3, e2.source),
	mapGuard(e1.guard.specification, e2),
	mapActivity(e1.effect, e2.effect),
	mapSignalTrigger(over e1.trigger, e2),
	ref mapVertex(e1.target, e2.target),
	ref mapVertex(e1.source, e2.source)
}

relate mapInitialTransition(
	uml:Pseudostate e1 when equals(e1.kind, "initial"),
	jf:JavaFrameTransition e2) {
	mapTransition[1](over e1.outgoing, e2)
}

/*
relate mapDefaultTransition(
	uml:Transition e1, jf:JavaFrameTransition e2, uml:State e3)
	when !ref checkNotDefaultGuard(e1.guard.specification) {
	mapActivity(e1.effect, e2.effect),
	mapSignalTrigger(over e1.trigger, e2),
	ref mapVertex(e1.target, e2.target)
}

relate checkNotDefaultGuard(uml:OpaqueExpression e1)
	when !(equals(e1.body, "else") | equals(e1.body, ""))
```

```
*/

relate mapGuard(uml:OpaqueExpression e1, jf:JavaFrameTransition e2) {
	equals(e1.body, e2.guardPredicate)
}

relate mapSignalTrigger(uml:SignalTrigger e1, jf:JavaFrameTransition e2) {
	mapTrigger(over e1.signal, e2)
}

relate mapTrigger(uml:Signal e1, jf:JavaFrameTransition e2) {
	ref mapSignal(e1, over e2.trigger)
}

/* END StateMachine and CompositeState section */

relate mapMediator extends mapJavaFrameClass(
	uml:Class e1 when stereotypeByName(
		match over e1.stereotypes, "Mediator"),
	jf:MediatorClass e2) {
	//ref mapForwardAcitivity [0..1](e1.activity, e2.forward)
}

relate mapComposite extends mapActiveObjectClass(
	uml:Class e1, jf:CompositeClass e2)
	when stereotypeByName(match over e1.stereotypes,"Composite"){
	//ordered mapActiveObjectField(over e1.ownedAttribute, over e2.activeobjectfield),
	ordered mapConnector(
		over e1.ownedConnector,
		over e2.mediatorconnection)
	,createMainClass[1](e1, over e2.package.ownedMember)
}

relate createMainClass (uml:Class e1, jf:MainClass e2)  {
	util:MatchString "{0}_Main" (e1.name, e2.name),
	createMainActiveObjectField(e1, e2.composite)
}

relate createMainActiveObjectField(uml:Class e1, jf:ActiveObjectField e2) {
	equals(e2.name, "Main"),
	ref mapComposite(e1, e2.type),
	equals(e2.aggregation, "composite"),
	ordered mapMediatorField(over e1.ownedPort, over e2.actualMediator)
}


relate mapSignal extends mapClassifier(uml:Signal e1, jf:SignalClass e2) {
	ordered mapProperty(over e1.ownedAttribute, over e2.ownedAttribute)
}

relate mapActiveObjectField extends mapJavaFrameProperty  (
	uml:Property e1 when equals(e1.isComposite, "true"),
	jf:ActiveObjectField e2) {
	mapActualMediatorField(over e1.type, e2,  e1)
}
```

```
relate mapActualMediatorField(
	uml:Class e1,
	jf:ActiveObjectField e2,
	uml:Property e3) {
	mapActiveObjectMediatorField(
		over e1.ownedPort,
		over e2.actualMediator, e3)
}

relate mapActiveObjectMediatorField(
	uml:Port e1,
	jf:MediatorField e2,
	uml:Property e3)
	when util:MatchString "{0}_{1}" (e3.name, e1.name, e2.name){
	equals(e1.visibility, e2.visibility)
	,equals(e1.default, e2.default)
	//,equals(e1.isComposite, e2.isComposite)
	,equals(e1.isStatic, e2.isStatic)
	,equals(e1.isUnique, e2.isUnique)
	,equals(e1.isLeaf, e2.isLeaf)
	,equals(e1.lower, e2.lower)
	,equals(e1.upper, e2.upper)
	,mapTypedElementType(e1, e2)
}

relate mapMediatorField extends mapJavaFrameProperty(
	uml:Port e1, jf:MediatorField e2) {
	equals(e1.isBehavior, e2.isInput)
}

relate mapConnector(uml:Connector e1, jf:MediatorConnection e2) {
	ordered mapConnectorEnd(over e1.end, e2)
}


abstract relate mapConnectorEnd(
	uml:ConnectorEnd e1, jf:MediatorConnection e2) {}

relate mapActualPortConnectorEnd extends mapConnectorEnd(
	uml:ConnectorEnd e1, jf:MediatorConnection e2)
		when util:InstanceOf "uml:Property" (e1.partWithPort) {
	ref mapActiveObjectMediatorField (
		e1.role, e2.end, e1.partWithPort)
}

relate mapFormalPortConnectorEnd extends mapConnectorEnd(
	uml:ConnectorEnd e1, jf:MediatorConnection e2)
		when !util:InstanceOf "uml:Property" (e1.partWithPort){
	ref mapMediatorField (over e1.role, over e2.end)
}
/*
 * These to compensate for not being able to access uml element
 * in MOFScript
 */
relate mapJavaFrameModel extends mapModel(
	uml:Model e1, jf:JavaFrameModel e2)
```

```
relate mapJavaFramePackage extends mapPackage(
	uml:Package e1, jf:JavaFramePackage e2)

relate mapJavaFrameProperty extends mapProperty(
	uml:Property e1, jf:JavaFrameProperty e2)

relate mapJavaFrameParameter extends mapParameter(
	uml:Parameter e1, jf:JavaFrameParameter e2)

relate mapJavaFrameClass extends mapClass(
	uml:Class e1, jf:JavaFrameClass e2)

relate mapJavaFrameOperation extends mapOperation(
	uml:Operation e1, jf:JavaFrameOperation e2)

relate mapJavaFrameActivity extends mapActivity(
	uml:Activity e1, jf:JavaFrameActivity e2)
```

JFUml2Java.m2t (MOFScript v 1.1.4)

```
texttransformation JfUml2Java (in jfuml:"http:///JF.ecore")

property OBJECT = "Object"
property file_extension = ".java"

property targetDir = "gen-src/"

jfuml.JavaFrameModel::main() {
	self.mapModel()
}

jfuml.JavaFrameModel::mapModel() {
	self.ownedMember->forEach(c:jfuml.JavaFramePackage) {
		c.mapPackage()
	}
}

jfuml.JavaFramePackage::mapPackage() {
	self.ownedMember->forEach(c : jfuml.JavaFrameClass) {
		c.mapClass()
	}
	self.ownedMember->forEach(c:jfuml.JavaFramePackage) {
		c.mapPackage()
	}
}

jfuml.JavaFrameClass::mapClass() {
	file (targetDir + self.owner.getFolderName() + "/" + self.name + file_extension)

	self.mapPackageDeclaration()
	nl
	self.mapImportDeclarations()
```

```
	nl
	self.mapClassDeclarationStart()
	nl
	self.mapFields()
	nl
	self.mapConstructor()
	nl
	self.ownedOperation->forEach(o:jfuml.JavaFrameOperation) {
		o.mapOperation()
	}
	nl
	self.mapClassDeclarationEnd()
}

jfuml.JavaFrameClass::mapClassDeclarationStart() {
	<%public class %> self.name <% extends %> self.getGeneralLiteral(OBJECT) <% { %>
	nl
}

jfuml.JavaFrameClass::mapClassDeclarationEnd() {
<%}%>nl
}

jfuml.JavaFrameClass::mapPackageDeclaration() {
	println("package "+self.owner.getQualifiedName()+ ";")
}

jfuml.JavaFrameClass::mapImportDeclarations() {
	 var imports:Hashtable // use hashtable to avoid duplicates
	 self.ownedAttribute->forEach(c:jfuml.JavaFrameProperty) {
		if (c.type != null) {
			if (c.type.oclIsKindOf(jfuml.JavaFrameClass)) {
				if (c.type.owner != self.owner) { // dont import types from same package
					imports.put(c.type, c.type)
				}
			}
		}
	 }
	 imports->forEach(c : jfuml.JavaFrameClass) {
		println("import " + c.getQualifiedName() + ";")
	 }

	 self.packageImport->forEach(c) {
		println("import " + c.importedPackage.getQualifiedName() + ".*;")
	 }
	 self.mapImportDeclarationsMergeExtension()
}

jfuml.JavaFrameClass::mapFields() {
	self.mergesMapFields()
	self.ownedAttribute->forEach(p:jfuml.JavaFrameProperty) {
		p.mapProperty()
	}

}
```

```
jfuml.JavaFrameClass::mergesMapFields() {}

jfuml.JavaFrameProperty::mapProperty() {
	tab<%%>self.getVisibilityLiteral() <% %> self.getTypeLiteral() <% %> self.name <%;%>nl
}

jfuml.JavaFrameClass::mapConstructor() {
	self.mapDefaultConstructor()
	nl
	if (!(self.ownedAttribute.isEmpty() && self.inheritedMember.isEmpty()))
		self.mapParameterizedConstructor()
	nl
	self.inheritsMapConstructor()
}

jfuml.JavaFrameClass::inheritsMapConstructor() {}

jfuml.JavaFrameClass::mapParameterizedConstructor() {
<%	public %> self.name <%(%> self.getConstructorParameterString() <%) {
		super(%> self.getConstructorSuperArgString() <%);
%>
	self.ownedAttribute->forEach(c) {
		tab(2)<%this.%> c.name <% = %> c.name <%;%>nl
	}
<%	}
%>
}

jfuml.JavaFrameClass::mapDefaultConstructor() {
<%	public %> self.name <%() {
		super();
	}
%>
}

jfuml.JavaFrameOperation::mapOperation() {
	var return:String
	var parameters:String
	if (self.returnResult.isEmpty()) {
		return = "void"
	} else {
		return = self.returnResult.first().getTypeLiteral()
	}
	self.ownedParameter->forEach(p:jfuml.JavaFrameParameter) {
		parameters += p.getTypeLiteral() + " " + p.name + ", "
	}
	if (parameters.size() > 0) {
		parameters = parameters.substring(0, parameters.size() - 2)
	}
	tab<%%> self.visibility <% %> return <% %> self.name <%(%> parameters <%) {
%>
	self.method->forEach(a:jfuml.JavaFrameActivity) {
		a.mapActivity()
	}
<%	}
%>
```

```
}

jfuml.JavaFrameActivity::mapActivity() {
	if (self.action.isEmpty()) {
		println(self.name)
	} else {
		self.action->forEach(a) {
			println(a.name)
		}
	}
}

/* HELPERS */
jfuml.JavaFrameProperty::getTypeLiteral() : String {
	if (self.type = null) {
		 result = OBJECT
	} else {
		result = self.type.name
	}
}

jfuml.JavaFrameParameter::getTypeLiteral() : String {
	if (self.type = null) {
		 result = OBJECT
	} else {
		result = self.type.name
	}
}

jfuml.JavaFrameProperty::getVisibilityLiteral() : String {
	if (self.visibility = "package")
		result = ""
	else
		result = self.visibility
}

jfuml.JavaFrameClass::getGeneralLiteral(default:String) : String {
	result = default
	if (!self.general.isEmpty()) {
		result = self.general.first().name
	}
}

jfuml.JavaFrameClass::getConstructorParameterString() : String {
	var cp:String
	self.ownedAttribute->forEach(c:jfuml.JavaFrameProperty) {
	  cp += c.getTypeLiteral() + " " + c.name + ", "
	}
	self.inheritedMember->forEach(c:jfuml.JavaFrameProperty) {
	  cp += c.getTypeLiteral() + " " + c.name + ", "
	}
	// Remove trailing comma
	if (cp.size() > 0) { cp = cp.substring(0, cp.size() - 2) }
	result = cp
}
```

```
jfuml.JavaFrameClass::getConstructorArgString() : String {
	var args:String
	args = self.mergesGetConstructorArgString()
	self.ownedAttribute->forEach(c:jfuml.JavaFrameProperty) {
		args += c.name + ", "
	}
	self.inheritedMember->forEach(c:jfuml.JavaFrameProperty) {
		args += c.name + ", "
	}
	// Remove trailing comma
	if (args.size() > 0) { args = args.substring(0, args.size() - 2) }
	result = args
}

jfuml.JavaFrameClass::mergesGetConstructorArgString() : String {
  result = ""
}

jfuml.JavaFrameClass::getConstructorSuperArgString() {
	var args:String
	self.inheritedMember->forEach(c:jfuml.JavaFrameProperty) {
		args += c.name + ", "
	}
	// Remove trailing comma
	if (args.size() > 0) { args = args.substring(0, args.size() - 2) }
	result = args
}

jfuml.JavaFramePackage::getQualifiedName() : String {
	result = self.qualifiedName.replace("::", ".")
}

jfuml.JavaFramePackage::getFolderName() : String {
	result = self.qualifiedName.replace("::", "/")
}

jfuml.JavaFrameClass::getQualifiedName() : String {
	if (self.owner.oclIsKindOf(jfuml.JavaFramePackage)) {
		result = self.owner.getQualifiedName() + "." + self.name
	} else {
		result = self.name
	}
}

jfuml.JavaFramePrimitiveType::getQualifiedName() : String {
	result = self.name
}


/* JAVAFRAME SPECIFIC (MOFScript help) */
jfuml.JavaFrameClass::mapImportDeclarationsMergeExtension() {
	println("import se.ericsson.eto.norarc.javaframe.*;")
}

/* ActiveObject */
jfuml.ActiveObjectClass::mergesMapFields() {
```

```
<%	Scheduler sched = null;

	/* Formal mediators */
%>
	self.formalMediator->forEach(c) {
		c.mapProperty();
	}
}

/* ActiveObjectField */
jfuml.ActiveObjectField::mapProperty() {
nl
<%	/* Part %> self.name <% */%>nl
	self.actualMediator->forEach(c) {
		tab<%%> c.getTypeLiteral() <% %> c.name <%;%>nl
	}
	tab<%%> self.getTypeLiteral() <% %> self.name <%;%>nl
}

jfuml.ActiveObjectField::mapInit() {
	self.actualMediator->forEach(c) {
		tab(2)<%%>c.name<% = new %>c.getTypeLiteral()<%(/*DefaultValue*/);%>nl
	}
	tab(2)<%%>self.name<% = new
%>self.getTypeLiteral()<%(%>self.getConstructorArgString()<%);%>nl
}

jfuml.ActiveObjectClass::getConstructorParameterString() {
	var r = "Scheduler sched, "
	self.formalMediator->forEach(c) {
		r += c.getTypeLiteral() + " " + c.name + ", "
	}
	r = r.substring(0, r.size() -2 )
	result = r
}

jfuml.ActiveObjectField::getConstructorArgString() {
	var r = "sched, "
	self.actualMediator->forEach(c) {
		r += c.name + ", "
	}
	r = r.substring(0, r.size() -2 )
	result = r
}

/* Composite */
jfuml.CompositeClass::mapClassDeclarationStart() {
	<% public class %> self.name <% extends Composite { %>
	nl
}

jfuml.CompositeClass::mapConstructor() {

<%	public %>self.name<%(%>self.getConstructorParameterString()<%) {
		this.sched = sched;
%>
```

```
        self.formalMediator->forEach(m) {
<%              this.%> m.name <% = %> m.name <%;%>nl
        }
        nl
        self.ownedAttribute->forEach(a : jfuml.ActiveObjectField) {
                a.mapInit()
        }
        nl
        self.mediatorconnection->forEach(c) {
<%              %>c.end.first().name<%.addAddress(%>c.end.last().name<%);%>nl
        }
<%      }
%>
}

/* StateMachine */
jfuml.StateMachineClass::mapClassDeclarationStart() {
<%public class %> self.name <% extends StateMachine {

        static CompositeState states = new %>self.compositeState.name<%("outermostState");
%>
        nl
}

jfuml.StateMachineClass::mapConstructor() { // TODO seperate parameter/attribute for SM
        var cp = self.getConstructorParameterString()
<%      public %> self.name <%(%> cp <%) {
                super(sched);
                this.sched = sched;
%>
        self.formalMediator->forEach(c) {
                tab(2)<%this.%> c.name <% = %> c.name <%;%>nl
        }
<%      }

%>
        self.mapExecStartTransition()
}

jfuml.StateMachineClass::mapExecStartTransition() {
<%
        protected void execStartTransition() {
                states.enterState(this);
        }
%>
}

/* CompositeStates */
jfuml.CompositeStateClass::mapClassDeclarationStart() {
<%public class %> self.name <% extends CompositeState {
%>
}

jfuml.CompositeStateClass::mergesMapFields() {
<%      /* States declared as static fields */%>nl
        self.state->forEach(c) {
```

```
		c.mapProperty()
	}
	nl
}

jfuml.CompositeStateClass::mapConstructor() {
<%	public %>self.name<%(String name) {
		super(name);
%>
	self.state->forEach(c) {
		tab(2)<%%>c.getName()<%.enclosingState = this;%>nl
	}
<%
	}
%>
	self.inheritsMapConstructor()
}

jfuml.CompositeStateClass::inheritsMapConstructor() {
	self.mapEnterState()
	self.mapExecTrans()
}

jfuml.CompositeStateClass::mapEnterState() {
<%
	public void enterState(StateMachine curfsm) {
		%>self.stateMachine.name<% csm = (%>self.stateMachine.name<%)curfsm;
%>
	self.initialTransition.mapFireTransition("if")
<%
	}
%>
}

jfuml.CompositeStateClass::mapExecTrans() {
<%
	protected boolean execTrans(Message signal, State st, StateMachine curfsm) {
		%>self.stateMachine.name<% csm = (%>self.stateMachine.name<%)curfsm;
%>
	self.state->forEach(s | !s.oclIsKindOf(jfuml.JavaFrameFinalState)) {
<%		if (st == %>s.getName()<%) {
%>		s.outgoing->forEach(t) {
			t.mapTransition()
		}
<%		}
%>
	}
<%
		return false;
	}
%>
}

/* State */
jfuml.JavaFrameState::mapProperty() {
<%	static %> self.getTypeLiteral() <% %> self.name <% = new %>
```

```
	self.getTypeLiteral() <%("%>self.name<%");%>nl
}

jfuml.JavaFrameFinalState::mapProperty() {
<%	static State finalState = new State("finalState");%>nl
}

jfuml.JavaFrameState::getTypeLiteral() : String {
	if (self.type = null)
		result = "State"
	else
		result = self.type.name
}

jfuml.JavaFrameState::mapStateChangeCode() {
	tab(2)<%%>self.getName()<%.enterState(csm);%>nl
}

jfuml.JavaFrameFinalState::mapStateChangeCode() {
<%			csm.moveStateMachine(null);
			finalState.enterState(csm);
			csm.owner.removeActiveObject(csm);
%>
}

jfuml.JavaFrameChoice::mapStateChangeCode() {
	var test:String = "if"
<%	/* Enter choice */
%>
	self.outgoing->forEach(t | !t.isDefault()) {
		t.mapFireTransition(test)
		test = "else if"
	}
	self.outgoing->forEach(t | t.isDefault()) {
		t.mapFireTransition(test)
	}
}

jfuml.JavaFrameState::getName() : String { result = self.name }
jfuml.JavaFrameFinalState::getName() : String { result = "finalState" }

/* Transition */
jfuml.JavaFrameTransition::isDefault() : Boolean {
	var g:String = self.guardPredicate
	if (self.guardPredicate = null) {
		result = "true"
	}
	if (g.equals("else") or g.equals("")) {
		result = true
	} else {
		result = false
	}
}

jfuml.JavaFrameTransition::getGuardLiteral() : String {
	if (self.guardPredicate = null)
```

```
            result = "true"
      else {
            if (self.guardPredicate.equals("") or self.guardPredicate.equals("else"))
                  result = "true"
            else
                  result = self.guardPredicate
      }
}

jfuml.JavaFrameTransition::mapFireTransition(test:String) {
      tab(2)<%%>test<% (%>self.getGuardLiteral()<%) {
%>
      if (!(self.effect = null)) {
            tab(3)<%%>self.effect.mapActivity() nl
      }
      self.target.mapStateChangeCode()
      tab(2)<%}
%>
}

jfuml.JavaFrameTransition::mapTransition() : String {
      var r:String = "\t\tif ("
      self.trigger->forEach(c) {
            r += "signal instanceof " + c.getQualifiedName() + " || "
      }
      r = r.substring(0, r.size() - 3)
      r += ") {\n"
      if (!self.trigger.size() > 1)
            r += "\t\t\t" + self.trigger.first().getQualifiedName() + " sig = ("
                  +self.trigger.first().getQualifiedName()+")signal;\n"
      else
            r += "\t\t\tMessage sig = signal;\n"
      print(r)
      self.mapFireTransition("if") // TODO include opt return stmt in mapFire..
<%                return true;
            }
%>
}

/* Mediator */
jfuml.MediatorClass::mapClassDeclarationStart() {
      <% public class %> self.name <% extends %> self.getGeneralLiteral("Mediator") <% { %>
      nl
}

/* Signal */
jfuml.SignalClass::mapClassDeclarationStart() {
      <% public class %> self.name <% extends %> self.getGeneralLiteral("Message") <% { %>
      nl
}

/* Overridden helpers */
jfuml.MediatorField::getTypeLiteral() : String {
      if (self.type = null) {
            result = "Mediator"
      } else {
```

```
            result = self.type.name
        }
}

/* Main class */
jfuml.MainClass::mapFields() {
        self.composite.mapProperty()
}

jfuml.MainClass::mapConstructor() {
<%    public %>self.name<%(String hostname, int portnumber) {
                Trace trc = null;
                // Create Trace object with socket communication
                // Establish socket communication with JFTrace
                if (hostname.length() != 0) { // JFTrace connection specified?
                        try {
                                trc = new Trace(true, hostname, portnumber);
                        } catch (Exception e) {
                                System.err.println("Socket connection to JFTrace failed to establish: "+ e);
                        }
                }
                Scheduler sched;
                if (trc != null)
                        sched = new Scheduler(trc); // with socket connected Trace object
                else
                        sched = new Scheduler(); // with default Trace object

                Thread t1 = new Thread(sched); // associating the Scheduler with a Thread

%>
        self.composite.mapInit()
<%
                // Required: start the JavaFrame machinery when the whole system is initialized
                t1.start();
                System.out.println("*********** JavaFrame starting ***************");
        }

        public static void main(String[] args) {
                // Process command options
                // Necessary if the JavaFrame application will use socket communication
                // with JFTrace
                String hostname = "";
                int portnumber = 0;
                for (int i = 0; i < args.length; i++) {
                        String token = args[i];
                        if (token.equals("-remote")) {
                                if (i == (args.length - 1)) {
                                        System.out.println("No remote specified.");
                                        usage();
                                        System.exit(1);
                                }
                                int index = args[++i].indexOf(':');
                                if (index != -1) {
                                        hostname = args[i].substring(0, index);
                                        Integer portnumberInt = new Integer(args[i]
                                                        .substring(index + 1));
```

```
                        portnumber = portnumberInt.intValue();
                    } else {
                        usage();
                        System.exit(1);
                    }
                } // process other options here
            }

            new %>self.name<%(hostname, portnumber);
        }

        private static void usage() {
            // Writes command line options
            System.out.println("Usage: " + "%>self.name<%"
                    + " [ -remote <hostname>:<port-number> ]");
            System.out.println();
        }
%>
}

/*******************************************/
/* QoS extension: Add freelist to signals */
jfuml.SignalClass::inheritsMapConstructor() {
        var constructorPars = self.getConstructorParameterString()
        var constructorArgs = self.getConstructorArgString()
        nl
<%      private static %>self.name<% %> self.name<%Freelist; // top of the freelist stack
        private %> self.name <% %>self.name<%Next; // the list pointer

        synchronized public static %>self.name<% New%>self.name<%() {
            %>self.name<% returnmessage;
            if (%>self.name<%Freelist != null) {// take a message from the freelist
                returnmessage = %>self.name<%Freelist;
                %>self.name<%Freelist = %>self.name<%Freelist.%>self.name<%Next;

            } else // freelist empty, new message must be generated
                returnmessage = new %>self.name<%();

            returnmessage.%>self.name<%Next = returnmessage; /* designating live object*/
            return returnmessage;
        }

%>      if (constructorPars.size() > 0) {<%
        synchronized public static %>self.name<% New%>self.name<%(%>constructorPars<%) {
            %>self.name<% returnmessage;
            if (%>self.name<%Freelist != null) {// take a message from the freelist
                returnmessage = %>self.name<%Freelist;
                %>self.name<%Freelist = %>self.name<%Freelist.%>self.name<%Next;
%>
            self.ownedAttribute->forEach(c) {
                tab(3)<%returnmessage.%>c.name <% = %> c.name <%;%>nl
            }
            self.inheritedMember->forEach(c) {
                tab(3)<%returnmessage.%>c.name <% = %> c.name <%;%>nl
            }
<%
```

```
            } else // freelist empty, new message must be generated
                returnmessage = new %>self.name<%(%>constructorArgs<%);

            returnmessage.%>self.name<%Next = returnmessage; /* designating live object*/
            return returnmessage;
        }

%>   }<%
        synchronized private static void delMsg(%>self.name<% sig) {
            if (sig.%>self.name<%Next==sig) { // check that it is live
                sig.%>self.name<%Next = %>self.name<%Freelist;
            %>self.name<%Freelist = sig;
            } else {
                System.err.println("**%>self.name<%.del: Object not live");
            }
    }

        public void del() {
        delMsg(this);
        }
%>
}
```

Appendix B - JfJava Transformations

UML2JavaMetamodel (RSM Transformation)

ClassRule.java

```
package no.uio.ifi.javaframetransformation.rules;

import jmm.Class;
import jmm.CompilationUnit;
import jmm.JmmFactory;
import jmm.JmmPackage;
import jmm.Package;
import no.uio.ifi.javaframetransformation.tools.ClassBuilder;
import no.uio.ifi.javaframetransformation.tools.JavaNameTool;
import no.uio.ifi.javaframetransformation.tools.References;

import org.eclipse.uml2.Classifier;

import com.ibm.xtools.transform.core.AbstractRule;
import com.ibm.xtools.transform.core.ITransformContext;

public class ClassRule extends AbstractRule {

   public ClassRule() {
      super();
   }

   public ClassRule(String id, String name) {
```

```
      super(id, name);
   }

   protected Class createTargetObject() {
      return JmmFactory.eINSTANCE.createClass();
   }

   protected Object createTarget(ITransformContext ruleContext)
         throws Exception {
      Classifier source = (Classifier) ruleContext.getSource();
      CompilationUnit cu = JmmFactory.eINSTANCE.createCompilationUnit();
      Class target = createTargetObject();
      target.setName(JavaNameTool.asClass(source.getName()));
      target.setIsAbstract(source.isAbstract());
      target.setIsPublic(true);
      cu.setName(JavaNameTool.asClass(source.getName()));

      Classifier superClass = ClassBuilder.getSuperType(source);
      if (superClass != null)
         References.setFeatureUnknownValue(target, JmmPackage.eINSTANCE.getClass_SuperClass(),
superClass);

      if (ruleContext.getTargetContainer() instanceof Package) {
         Package owner = (Package) ruleContext.getTargetContainer();
         cu.setPackage(owner);
         /* Add class as type decl to cu */
         target.setCompilationUnit(cu);
         /* Add cu to package */
         cu.setPackage(owner);
         /* Map reference */
         References.mapReference(source, target);
      }
      return target;
   }

   public boolean canAccept(ITransformContext context) {
      Classifier source = (Classifier) context.getSource();
      return source.getAppliedStereotypes().isEmpty();
   }
}
```

PackageRule.java

```
package no.uio.ifi.javaframetransformation.rules;

import jmm.JmmFactory;
import jmm.Package;
import no.uio.ifi.javaframetransformation.tools.JavaNameTool;
import no.uio.ifi.javaframetransformation.tools.References;

import com.ibm.xtools.transform.core.AbstractRule;
import com.ibm.xtools.transform.core.ITransformContext;

public class PackageRule extends AbstractRule {

   public PackageRule(String id, String name) {
```

```
        super(id, name);
    }

    protected Object createTarget(ITransformContext ruleContext) throws Exception {
        org.eclipse.uml2.Package source = (org.eclipse.uml2.Package)ruleContext.getSource();

        jmm.Package target = JmmFactory.eINSTANCE.createPackage();
        target.setName(JavaNameTool.asPackage(source.getName()));

        if (ruleContext.getTargetContainer() instanceof jmm.Package) {
            Package owner = (jmm.Package)ruleContext.getTargetContainer();
            target.setSuperPackage(owner);
            References.mapReference(source, target);
                return target;
             }
        return null;
    }
}
```

References.java

```
package no.uio.ifi.javaframetransformation.tools;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EReference;

public class References {

    protected static HashMap references = new HashMap();

    protected static List unknownTargetDeferedTargets = new ArrayList();
    protected static List unknownTargetDeferedFeatures = new ArrayList();
    protected static List unknownTargetDeferedValues = new ArrayList();

    protected static List unknownValueDeferedTargets = new ArrayList();
    protected static List unknownValueDeferedFeatures = new ArrayList();
    protected static List unknownValueDeferedValue = new ArrayList();

    protected static List unknownBothDeferedTargets = new ArrayList();
    protected static List unknownBothDeferedFeatures = new ArrayList();
    protected static List unknownBothDeferedValue = new ArrayList();

    public static void mapReference(EObject source, EObject target) {
        references.put(source, target);
    }

    protected static EObject getMappedReference(Object source) {
        return (EObject)references.get(source);
    }
```

```
public static void setFeatureUnknownTarget(EObject target, EReference feature, EObject value) {
   EObject ref = getMappedReference(target);
   if (ref != null) {
      setFeature(ref, feature, value);
   } else {
      deferFeatureUnknownTarget(target, feature, value);
   }
}

public static void setFeatureUnknownValue(EObject target, EReference feature, EObject value) {
   EObject ref = getMappedReference(value);
   if (ref != null) {
      setFeature(target, feature, ref);
   } else {
      deferFeatureUnknownValue(target, feature, value);
   }
}

public static void setFeatureUnknownBoth(EObject target, EReference feature, EObject value) {
   EObject value_ref = getMappedReference(value);
   EObject target_ref = getMappedReference(target);
   if (value_ref != null && target_ref != null) {
      setFeature(target_ref, feature, value_ref);
   } else {
      deferFeatureUnknownBoth(target, feature, value);
   }
}

protected static void deferFeatureUnknownTarget(EObject target, EReference feature, EObject value) {
   unknownTargetDeferedTargets.add(target);
   unknownTargetDeferedFeatures.add(feature);
   unknownTargetDeferedValues.add(value);
}

protected static void deferFeatureUnknownValue(EObject target, EReference feature, EObject value) {
   unknownValueDeferedTargets.add(target);
   unknownValueDeferedFeatures.add(feature);
   unknownValueDeferedValue.add(value);
}

protected static void deferFeatureUnknownBoth(EObject target, EReference feature, EObject value) {
   unknownBothDeferedTargets.add(target);
   unknownBothDeferedFeatures.add(feature);
   unknownBothDeferedValue.add(value);
}

public static void setDeferedFeatures() {
   for (int i=0; i<unknownTargetDeferedTargets.size();i++) {
      EObject target = (EObject)unknownTargetDeferedTargets.get(i);
      EReference feature = (EReference)unknownTargetDeferedFeatures.get(i);
      EObject value = (EObject)unknownTargetDeferedValues.get(i);

      EObject refTarget = getMappedReference(target);
      if (refTarget != null)
         setFeature(refTarget, feature, value);
```

```
        }
        for (int i=0; i<unknownValueDeferedTargets.size();i++) {
            EObject target = (EObject)unknownValueDeferedTargets.get(i);
            EReference feature = (EReference)unknownValueDeferedFeatures.get(i);
            EObject value = (EObject)unknownValueDeferedValue.get(i);
            setFeature(target, feature, getMappedReference(value));
        }
        for (int i=0; i<unknownBothDeferedTargets.size(); i++) {
            EObject target = (EObject)unknownValueDeferedTargets.get(i);
            EReference feature = (EReference)unknownValueDeferedFeatures.get(i);
            EObject value = (EObject)unknownValueDeferedValue.get(i);
            EObject target_ref = getMappedReference(target);
            EObject value_ref = getMappedReference(value);
            if (target_ref != null && value_ref != null)
                setFeature(target_ref, feature, value_ref);
            // TODO - add warning if null in reference
        }
    }

    public static void setFeature(EObject target, EReference feature, EObject newValue) {
        if (feature.isMany()) {
            EList featureList = (EList)target.eGet(feature);
            featureList.add(newValue);
            //target.eSet(feature, featureList);
        } else {
            target.eSet(feature, newValue);
        }
    }

}
```