Master’s Thesis

AUTOMATIC GENERATION OF PLATFORM SPECIFIC MODEL TRANSFORMATION

by

Ákos Horváth

May 19, 2006

Department of Measurement and Information Systems
Budapest University of Technology and Economics

Supervisors

Dr. Dániel Varró  Gergely Varró
assistant professor    PhD student
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of the PST transformer generation approach</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Metamodels for the object relational mapping</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>A sample model</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The entire object-relation mapping</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>The GT rule classR application on a sample model</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>VTML representation of RelDB</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>A sample graph pattern</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>the VTCL description of classR GT rule</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>ASM example</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>A sample graph transformation rule and its corresponding search graph</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Weighted search graph of classR and a possible search plan</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of meta-model Java generator</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Overview of ASM and GT Java generator</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>The relation2SGEdge GT rule</td>
<td>50</td>
</tr>
<tr>
<td>5.1</td>
<td>Initial model of the test case for the N=3 case</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Sequential and Parallel execution</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>The Ecore kernel</td>
<td>57</td>
</tr>
<tr>
<td>6.2</td>
<td>Overview of the Model integration</td>
<td>60</td>
</tr>
</tbody>
</table>
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2</td>
<td>GT pattern generator</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>Performance assessment</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td>Case study</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>EMF Model Integration</td>
<td>56</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Eclipse Modeling Framework</td>
<td>56</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Listeners in Viatra2</td>
<td>58</td>
</tr>
<tr>
<td>6.2</td>
<td>Connecting Viatra2 to EMF</td>
<td>59</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Overview of the Integration</td>
<td>59</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Notification handling</td>
<td>60</td>
</tr>
<tr>
<td>6.3</td>
<td>Graph Transformation in EMF</td>
<td>62</td>
</tr>
<tr>
<td>6.3.1</td>
<td>EMF based PSM</td>
<td>63</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Modifications in the generated PST</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>67</td>
</tr>
<tr>
<td>7.1</td>
<td>Related Work</td>
<td>67</td>
</tr>
<tr>
<td>7.2</td>
<td>Result Assessment</td>
<td>68</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Main Conclusion</td>
<td>68</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Future work</td>
<td>68</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Acknowledgement</td>
<td>69</td>
</tr>
<tr>
<td>A</td>
<td>Generated code</td>
<td>73</td>
</tr>
<tr>
<td>A.1</td>
<td>PSM model of Assoc entity in Java</td>
<td>73</td>
</tr>
<tr>
<td>A.2</td>
<td>PSM model of Assoc entity in EMF</td>
<td>76</td>
</tr>
<tr>
<td>A.3</td>
<td>The LHS pattern of classR, in choose mode without input parameters</td>
<td>81</td>
</tr>
<tr>
<td>B</td>
<td>Auxiliary classes and interfaces</td>
<td>82</td>
</tr>
<tr>
<td>B.1</td>
<td>Name and Entity interfaces</td>
<td>82</td>
</tr>
<tr>
<td>B.2</td>
<td>FunctionHolder, SourceHolder and ConHolder class</td>
<td>82</td>
</tr>
<tr>
<td>B.3</td>
<td>EMF and Viatra2 Adapters</td>
<td>83</td>
</tr>
<tr>
<td>C</td>
<td>PST generation in Viatra2</td>
<td>85</td>
</tr>
<tr>
<td>C.1</td>
<td>The makeRelation ASM rule</td>
<td>85</td>
</tr>
<tr>
<td>C.2</td>
<td>ASM rule nestedR</td>
<td>86</td>
</tr>
<tr>
<td>C.3</td>
<td>The GT rule relation2SGEdge</td>
<td>86</td>
</tr>
<tr>
<td>C.4</td>
<td>Meta-model of the context graph</td>
<td>87</td>
</tr>
<tr>
<td>C.5</td>
<td>Meta-model of the search graph</td>
<td>87</td>
</tr>
<tr>
<td>C.6</td>
<td>VTML description of the ASM and GT based model transformation engine</td>
<td>88</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 System Modeling Overview

Nowadays, the immense role of model transformation (MT) concepts and tools is unquestionable for the success of the model-driven system development (MDSD). The task of MDSD is to provide a solution for the challenge of ever accelerating business and technology change. Based on high-level model standards (e.g. UML), MDSD separates business and application logic from underlying platform technology. Platform-independent models (PIM) of an application are built using the high level model standards. These platform independent models describe the business functionality and behavior of an application separate from the platform-specific models (PSM) that implement it, separating the core of the application from technology while enabling interoperability across platforms. The success of the MDSD depends on how much of the transformation from PIM to PSMs can be automated, and that is the reason why MT concepts and tools are vital for the success.

Transformations are basically mathematical operators, but in the MDSD world models are usually captured by graph-based structures so model transformations can be described by graph transformations. Informally, a graph transformation (GT) rule performs local manipulation on graph models by finding a matching of the pattern prescribed by its left-hand side (LHS) graph in the model, and changing it according to the right-hand side (RHS) graph. GT rules are capable to manipulate the model locally, but every MT concepts which are based on GT rules connects additional control structure for complex MT constructs, so more and more complex MTs can be captured by a sequence of GT rules. But regardless of the approach of the model transformation, these concepts should enable a cost and time efficient design of (i) manipulations within a single modeling language (or domain) (ii) mappings and synchronization between different modeling languages and/or source code (iii) semantic translations into various mathematical domains to carry out formal model analysis.

As model transformation is becoming an engineer for discipline (transware), tool support is necessitated for the entire life-cycle, i.e., the specification, design, execution, validation and maintenance of transformations. However, different phases of transformation design frequently set up conflicting requirements, and it is difficult to find the best compromise. For instance, the main driver in the execution phase is performance, therefore,
CHAPTER 1. INTRODUCTION

a compiled MT approach (where a transformation is compiled directly into native source code) is advantageous. On the other hand, interpreted MT approaches (where transformations are available as models) have a clear advantage during the validation (e.g. by interactive simulation) or the maintenance phase due to their flexibility.

Advanced model transformation tools should support both interpreted and compiled approaches to separate the design (validation, maintenance) of a transformation from its execution. Interpreter-based platform-independent transformers \[5, 33\] (PIT) ease the development (and testing, debugging, validation) of model transformations within a single transformation framework without relying on a highly optimized target transformation technology. Platform-specific transformers (PST) are compiled (in a complex model transformation and/or code generation step) from an already validated transformation specification into various underlying tools or platform-dependent transformation technologies (e.g. Java, XSLT, etc.). Platforms here should be understood as the tools that allow the specification, design and execution of transformations (not to be confused with the “platform” term as used on the PIM to PSM application development life-cycle: both platforms are different ones!).

VIATRA2 \[3\] is a model transformation tool having been developed at the TUB-DMIS. VIATRA2 already supports interpreter-based PIT model transformations and metamorphisms \[33\] (i.e., a transformation having transformations as its input and output), which can be defined by the combination of graph transformation \[11\] (GT) and abstract state machines \[6\] (ASM). On the other hand, the generation of platform specific transformers constitutes a part of the ongoing research at the department. Java has been selected as a platform for implementing the PST as it has the advantage of speed, and a wide range support of off-the-shelf CASE tools being capable of integrating stand-alone plug-ins.

1.2 Research Objectives

My research objectives are:

- to specify an automated mapping of meta-models and models of the VIATRA2 tool to Java classes and objects (see Sec. \[3.1\]),
- to define a module that automatically generates a (native Java) platform specific transformer from the platform independent model transformation description (including both abstract state machine and graph transformation constructs) defined in the VIATRA2 framework (see Sec. \[3.2\] and \[3.3\]),
- to design a meta-transformation for the above-mentioned PIT-to-PST mapping itself by using only graph transformation and abstract state machine constructs (see Chapter \[4\]),
- to measure the run-time performance of the generated code (see Chapter \[5\]) in order to enable the participation of VIATRA2 in the benchmarking process \[34\] of graph transformation tools,
• to integrate the graph transformation capabilities to the Eclipse Modeling Framework [1] (see in Chapter [6]).

While the main guidelines of these transformations have been outlined in [31] (by my supervisors), the complete specification and the Viatra2 implementation of these transformations by means of meta-transformations are my own contributions.

1.3 Overview of the Approach

An overview of the overall approach is provided in Fig. 1.1.

Figure 1.1: Overview of the PST transformer generation approach

The upper and lower parts represent constructs and activities of the Viatra2 framework and of the Java platform, respectively.

• Chapter 2 gives an overview of the meta-modeling and model transformation concepts and notations, which can be used to define platform-independent transformations in the Viatra2 tool.

The general flow of developing and executing PITs in Viatra2 is the following.

1. **Meta-model design and model import.** Meta-models of the source and target modeling languages are designed according to the visual and precise metamodeling (VPM) approach [32], which is the built-in meta-modeling framework of Viatra2. Source models can be imported by using model importers and these models have an internal VPM representation. As VPM has a central role in Viatra2, Sec. 2.1 gives an introduction to the VPM approach.

2. **Transformation design.** Model transformations between source and target meta-models are captured by a seamless integration of GT and ASM rules [30].
Concepts of graph transformation and abstract state machines are presented in Sec. 2.2 and 2.3 respectively.

3. **Transformation execution.** These transformations are executed on the source model by a transformation engine to derive the target model. Since my research focuses on Java specific transformations, the interpreted version [3] of transformation execution is not discussed here.

4. **Model export.** The target model can be post-processed by special code generation transformations and code formatters in order to be printed in the required output format. Model export is again out of scope of this thesis.

Beyond the graphical representation, VIATRA2 has two textual languages (presented in Sec. 2.4) for defining VPM models (i.e., meta-models and models) and model transformations, respectively.

- Chapter 3 presents the mapping of VPM models (Sec. 3.1) and model transformation descriptions (Sec. 3.2 and 3.3) to native Java. Implementation of these mappings is in a dominating way the results of my work.

- Chapter 4 introduces the design of a meta-transformation that captures the PIT-to-PST transformation of Chapter 3 by using only GT and ASM constructs. In other words, I used the VIATRA2 framework for developing a meta-transformation, which can generate Java specific transformers from the platform-independent model transformations, which could also be designed in VIATRA2. The approach described in this chapter is a result of my work.

- Chapter 5 presents the results of the measurements I performed for assessing the run-time performance of the generated code.

- Chapter 6 proposes how to integrate the plugin into Eclipse Modeling Framework. The presented integration is again a result of my work.

- Chapter 7 gives an overview of the related work, concludes and evaluates my work. Finally, it presents some research directions for the future.
Chapter 2

Models and Transformations

This chapter briefly introduces the main notions of models and meta-models (used in Viatra2), and it also discusses how these models can be manipulated by using graph transformation (GT) and abstract state machine (ASM) rules.

2.1 Visual and Precise Meta-modeling framework

Visual and precise meta-modeling framework (VPM) (the default underlying framework of Viatra2) is a meta-modeling language, which is capable of defining models and meta-models in a single model space. In order to define the input and the output of a model transformation, we need meta-models and models for both the source and the target language.

The VPM framework has a mathematically precise notation, which eases model checking and validation. A VPM model has a graph-based structure facilitating the design and maintenance with visual tools. VPM maintains explicit instance-of relationship supporting the storage of meta-models and models in the same model space. It also has the advantage over Meta-Object Facility [22] (MOF) to be able to handle multi-level meta-modeling.

VPM model elements. The VPM language has three basic kinds of model elements:

- A **VPM entity** represents basic concepts of a (modeling) domain, it can be considered as a common term for MOF packages, classes or objects. A VPM entity may also have an associated *value*, which is a string that contains application-specific data.

- A **VPM relation** represents a general relationship between other VPM entities. It can be considered as a common term for MOF associations, basically *one-to-many* relations.

- A **VPM function** represents a general relationship between other VPM entities. It can be considered as a common term of MOF attribute.

There are three special relationships defined between model elements:
The generalization relation represents binary superclass-subclass relationships (like the UML generalization concept).

The instance-of relation expresses instantiation between model elements. This relation explicitly connects model level constructs to meta-model level constructs.

The containment relation arranges the model elements into a strict containment hierarchy with a tree structure.

Meta-models. A meta-model describes the abstract syntax of a modeling language (or domain). Nodes of the meta-model are VPM entities, while the edges of the meta-model will be VPM functions or relations. There are four types of edges based on the multiplicity of the source and target elements:

- **one-to-one** edges will be represented by a single VPM function from the source to the target entity.
- **one-to-many** edges will be represented by a single VPM relation from the source to the target entity.
- **many-to-one** edges will be represented by a single VPM function from the source to the target entity.
- **many-to-many** edges will be represented by a VPM relation from the source to the target entity, and a VPM relation from the target to the source.

Example 1 In order to present all the concepts and techniques, a standard object relational mapping [29] will be used throughout this thesis as a running example, which generates a relational database schema from a UML class diagram.

Meta-models of UML class diagrams and relational database schemas (following the CWM standard [23]) are depicted in Fig. 2.1. For the sake of simplicity, the traditional MOF notation (with aggregations and multiplicities) is used in the figure.

![Figure 2.1: Metamodels for the object relational mapping](image)

Nodes (e.g. Schema) of a VPM model are called VPM entities. Traditional MOF attributes and many-to-one associations are generalized by the concept of VPM function that lead between a source and a target model element. One-to-many associations appear
as VPM relation (like EO element ownership, CF classifier feature). Many-to-many associations (such as KRF key relation feature) between two model elements of MOF are denoted by two VPM relations.

The object-relational mapping additionally requires the maintenance of one-to-one reference edges between source and target model elements. These edges are not shown in Fig. 2.1 however they are denoted by dashed lines in subsequent figures.

Instance models. The instance model describes concrete systems defined in a modeling language. The instance model should be a well-formed instance of the meta-model, which means that each VPM entity, relation and function of the instance model must be connected to a corresponding VPM entity, relation and function in the meta-model by an explicit instance-of relationship edge, respectively. For the sake of easier readability, we use an object diagram like notation in the figures to denote the type of a model-level VPM entity instead of explicit instance-of relationship edges required by VPM.

Example 2 A sample instance model is presented in Fig. 2.2.

This instance model has four classes (c1, c2, c3, c4), two packages (p1, p2) and a schema (s1). Package p1 contains three classes named c1, c2 and c3 and a reference to the s1 schema element. Package p2 has only a single class c4.

2.2 Graph transformation

Graph transformation provides a pattern and rule based manipulation of graph models. Each rule application transforms a graph by replacing a part of it by another graph.

Graph transformation rule. A graph transformation rule (GT rule) contains a left-hand side graph LHS, a right-hand side graph RHS, and a negative application condition graph NAC. The LHS and the NAC graphs are together called the precondition PRE of the rule.

In the thesis, we use the graphical representation initially introduced in [14], where the union of these (PRE,RHS) graphs is presented. Elements to be deleted are marked by the
del keyword, elements to be created are labelled by the add, while elements in the NAC graph are denoted by the neg keyword.

Example 3 The entire object-relational mapping can be formalized by six graph transformation rules [34] as presented in Fig. 2.3.

In the precondition of GT Rule classR of Fig. 2.3(c), the LHS prescribes that there exists a class \( c \) located in such a package \( p \), which is already transformed into a corresponding schema \( s \), while NAC prescribes that this class \( c \) is not allowed to be related to any table \( t \). The rule manipulation part prescribes that a new table \( t \) is generated with a column \( \text{tid} \) and a primary key constraint \( \text{tpk} \) as the result of the rule application.

Rule application. The application of a GT rule to a host model \( M \) replaces a matching of the LHS in \( M \) by an image of the RHS. This is performed by (i) finding a matching of
LHS in \( M \) (by graph pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain objects and links) (iii) removing a part of the model \( M \) that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the \textit{derived model} \( M' \). The \textit{pattern matching phase} consists of steps (i) and (ii), while the \textit{manipulation phase} is constituted by steps (iii) and (iv).

![A sample model](a) A sample model

![Model with selected nodes](b) Model with selected nodes

![The resulting model after the GT rule application](c) The resulting model after the GT rule application

Figure 2.4: The GT rule \texttt{classR} application on a sample model

\textbf{Example 4} An application of GT rule \texttt{classR} on a sample model is shown in Fig. 2.4.

The host model before the rule application is depicted in Fig. 2.4(a). It holds packages, classes and schemas. In Fig. 2.4(b) the host model elements having been selected by the precondition pattern match of the \texttt{classR} GT rule are marked by dotted lines. The derived model is depicted in Fig. 2.4(c), in which new model elements are marked by thick lines.
2.3 Abstract state machines

While the elementary steps of complex model transformation are captured by GT rules, these GT rules are assembled into a complex transformation program by abstract state machine rules [6]. In order to semantically integrate the two paradigms, GT rules are treated as traditional ASM rules (called by the apply construct), and graph patterns (used in the LHS and NAC graphs) can be used as existentially quantified Boolean formulae in ASM conditions (find construct). More information on the semantic integration between graph transformation and abstract state machines can be found in [30].

The basic elements of an ASM program are:

- **ASM rules** are analogous with methods in OO languages. They have input parameters and hold a set of operations. There are two types: built-in, and user defined. Built-in rules are such as iterate rule and update and all other basic language elements, while all the rest are user defined.

- **ASM variables** are the same as variables in OO languages, they hold values (model element references, constants, etc.).

- **ASM functions** are special mathematical functions, which store values in arrays. These values can be updated from the ASM program. These functions are called dynamic. There are also static functions, which means that they cannot change their values. For example, the basic mathematical functions (+,-,*,/) are static.

Now we give a brief overview of the control structure used in ASM.

- **Random rule:**
  Random rule is used to execute one rule that has been randomly selected from the nested rules.

- **Parallel rule:**
  Parallel rule is used to execute several rules simultaneously.

- **Sequential rule:**
  The sequential rule specifies the execution order of the nested rules.

- **Forall rule:**
  The forall rule is used to execute a rule for every element in the model space that has a specific property. The property checker is an incoming parameter of the forall rule.

- **Choose rule:**
  The choose rule is used to execute a rule for an element in the model space that has a specific property. As in case of the forall rule, the property checker is again an incoming parameter.
• **Iterate rule:**
  This rule specifies an iterative execution of a given rule as long as possible. It terminates when a rule fails inside the block.

• **Let rule:**
  The let rule assigns a value to a variable and executes an ASM rule. The signed value is deleted after the rule has been completed.

• **Skip rule:**
  Skip rule represent a no operation instruction.

• **Update rule:**
  Update rule let the user change the value of ASM functions or variables

• **Conditional rule:**
  Conditional rule defines a binary branch in the flow of execution. The concept of conditional rules is similar to the conditional instructions (if-then-else) of OO programming languages.

### 2.4 Models and Transformations in Viatra2

Though Viatra2 stores models and transformations in a graph based style, and has a graphical modeling interface for the design of models and meta-models, it also has the opportunity to parse the models and transformations from textual files. Here is a brief introduction to these languages.

#### 2.4.1 Visual Textual Modeling Language

In Viatra2, the textual meta-modeling language is called Viatra Textual Meta-modeling Language (VTML).

The basic elements of the language are the element declarations which are Prolog-like facts. A VPM entity can be declared in the form `<type>(<name>)`, where `type` is the type of the given entity and `name` is the name of the new entity. Type declarations are mandatory, because all entities must have a type. If an entity has no definite type, it is instantiated from the root VPM entity. As entities may contain other model elements, the containment is done similarly to the C language, where the program blocks are marked by braces `{}`. Contained elements are represented in a block surrounded by braces after the container entity.

A VPM relation/function can be defined similarly, but the source and target model elements must also be defined. The syntax of relation definition is the following:

`<type>(<name>,<source>,<target>)`

A relation is always contained by its source entity. The containment hierarchy defines namespaces in the model space. This enables the definition of the fully qualified name
(FQN) of model elements. The FQN is equal to the list of containers of a given model element from the model space root to the element. For example, the FQN in Fig. 2.5 is RelDB.Column, while the FQN of the relation clmpky is RelDB.Column.clmpky. The local name of a model element must be unique in its container, which also ensures the uniqueness of FQNs. Special relationships can be represented by the keywords supertypeOf, and subtypeOf for generalization; and typeOf, and instanceOf for instantiation. The syntax is the following: \(<\text{relationship}>(\ <\text{supplier}\>, \ <\text{client}\>).\) For example, typeOf(RelDB.Column, C1) defines that the entity C1 is an instance of the meta-model element RelDB.Column. This way, a model element may have multiple types to support multidomain modeling.

**Example 5** The VTML description of the RelDB meta-model is presented in Fig. 2.5.

```java
entity(RelDB) //holder entity
{
    entity(Schema); //entities of the meta-model
    entity(Table);
    entity(Column);
    entity(FKey);
    entity(PKey);

    relation(schtbl,Schema,Table);
    relation(tblpky,Table,PKey);
    relation(tblfky,Table,FKey);
    relation(tblclm,Table,Column);

    relation(clmpky,Column,PKey);
    relation(pkyclm,PKey,Column);
    relation(clmfky,Column,FKey);
    relation(fkyclm,FKey,Column);
}
```

Figure 2.5: VTML representation of RelDB

- **RelDB** entity is the holder element of all other VPM entities.
- **VPM functions/relations** are created as described above. For example the relation tblpky (table to primary key) leads from the Table element, to the PKey element.

### 2.4.2 Viatra Textual Command Language

Transformation descriptions in Viatra2 consist of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph patterns (GP) define constraints and conditions on models, graph transformation (GT) rules support the definition of elementary model manipulations, while abstract state machine (ASM) rules can be used for the description of control structures.\[^1\]

\[^1\]The VPML description of the whole combined meta-model of ASM and GT rules can be found in Appendix C.6
The language that is created to implement all these concepts is the Viatra Textual Command Language (VTCL).

In VTCL, transformations are represented as **ASM machines**, which may consist of **GT patterns**, **GT rules**, **ASM rules** and **functions**.

**Graph patterns.** Graph patterns are the atomic units of model transformations. They represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. A model (i.e., a part of the model space) can satisfy a graph pattern, if the pattern can be matched to a subgraph of the model.

- Patterns are defined using the `pattern` keyword. Patterns may have parameters that are listed after the pattern name. The pattern body contains VPM model elements (i.e, a relationship definitions), which are identical to the VTML language constructs.

- The keyword `neg` marks a subpattern that is embedded into another pattern and it represents a negative condition for the original pattern. If this negative condition can be satisfied, the embedding pattern matching will fail. A unique feature of the VTCL pattern language is that, negative conditions can be embedded into each other in an arbitrary depth (e.g., negations of negations), thus the expressiveness of such patterns converges to first order logic [25].

- There can also be check conditions denoted by the `check` keyword that are Boolean formulae which must be satisfied in order to make the pattern true. These check conditions are only a practical extension to the theoretical foundations of GT patterns.

**Example 6** In the following example, this simple pattern of Fig. 2.6 can be fulfilled by `Column` instances that do not have parent `Table`, and the `Column` (as an entity of the model) has no name.

- The lhs pattern can be satisfied, if the incoming parameter (C) is a Column, and it has no name.

- The negative pattern in the example can be satisfied, if there is a table `Tbl` for the column in the parameter C that is the parent of C.

- In this example, we check whether the name of the class is empty. The pattern can be matched to columns with non-empty names only.

When a predefined graph pattern is called (using the `find` keyword), this means that a substitution for the free (unbound) parameters of the specified graph pattern has to be found that satisfies the pattern. If there are bound variables passed as parameters, they are treated as additional constraints, and they remain bound throughout the pattern matching process. By default, the free variables will be substituted by existential quantification,
/* C is a class without parents and with non-empty name */

pattern wrongColumn(C) = //C is a Column without Table and with non-empty name
{
    RelDB.Column(C);
}

neg pattern negCondition(C) = //negative pattern
{
    RelDB.Column(C);
    RelDB.Table(Tbl);
    RelDB.Table.tblclm(P,C,Tbl); //P is the variable holding tblclm
}

check (name(C)!="") //checking non-empty name

Figure 2.6: A sample graph pattern

which means that only one (non-deterministically selected) matching will be generated. If a variable is universally quantified by an external forall construct (see Sec. 2.4.2), the pattern matching will be executed in parallel by searching for all possible values of the given variable.

Graph transformation rules. While graph pattern defines logic conditions (formulas) on models, the manipulation of models is defined by graph transformation rules, which heavily rely on graph patterns which define the application criteria of transformation steps.

Graph transformation rules are defined by the gtrule keyword, and they are allowed to have directed (in/out/inout) parameters. The GT rule contains a simple pattern (called precondition), that jointly defines the left hand side (LHS) of the graph transformation rule, and negative pattern, and the action part contains the model manipulating rules. The action part is only invoked when the conditional pattern matching is succeeded. In the action part not only model manipulating rules (del, new, setValue and rename) but any other kind of ASM rules can be placed, even GT rule invocations, so GT rules can call each other recursively.

Example 7 The VTCL description of the classR GT rule of Fig. 2.3(c) is as follows:

- The condition pattern is defined as described in Fig. 2.6
- The lhs pattern can be satisfied, if the parameter (C) is a Class, the parameter (P) is a Package, the Schema (S) is connected to the Package with a relation pcksch, and Package (P) has a relation pckcls to Class (C).
- The negative pattern in the example can be satisfied, if parameter (C) is a Class, and (C) has a relation r1 to a Table (T).
- In the action part of the GT rule, new model elements are created with the new keyword. To create a new entity two parameters are passed, the type of the entity (e.g
CHAPTER 2. MODELS AND TRANSFORMATIONS

\[ \text{gtrule classR}(\text{in } P, \text{ in } C) = \{ \text{//Input parameters} \]

\[ \text{precondition pattern condition}(P,C) = \]
\[ \{ \text{UML.Class}(C); \]
\[ \text{UML.Package}(P); \]
\[ \text{RelDB.Schema}(S); \]
\[ \text{UML.Package.pckcls}(E01,P,C); \]
\[ \text{UML.Package.pcksch}(R1,P,S); \]
\[ \}
\]

\[ \text{neg pattern p}(C) = \text{//negative pattern} \]
\[ \{ \text{UML.Class}(C); \]
\[ \text{RelDB.Table}(T); \]
\[ \text{UML.Class.clstbl}(R2,C,T); \}
\]

\[ \text{action} \]
\[ \{ \text{new(RelDB.Column}(Tid)); \}
\[ \text{//entity creation} \]
\[ \text{new(} \text{RelDB.PKey}(Tpk)); \]
\[ \text{new(} \text{RelDB.Table}(T)); \]
\[ \text{//relations and functions} \]
\[ \text{new(} \text{RelDB.Table.tblpky}(E02b,T,Tpk)); \]
\[ \text{new(} \text{RelDB.Column.clmpky}(Ufa,Tid,Tpk)); \]
\[ \text{new(} \text{RelDB.PKey.pkyclm}(Ufb,Tpk,Tid)); \]
\[ \text{new(} \text{RelDB.Table.tblclm}(Cfa,T,Tid)); \]
\[ \text{new(} \text{UML.Class.clstbl}(R2a,C,T)); \]
\[ \text{new(} \text{RelDB.Table.tblcls}(R2b,T,C)); \]
\[ \text{new(} \text{RelDB.Schema.schtbl}(R1b,S,T)); \}
\]

Figure 2.7: the VTCL description of classR GT rule

\text{RelDB.Pkey} \text{ and a variable (e.g., } Tpk \text{) which will hold the newly created element. To create a new function/relation four parameters are passed, the type of the relation/-function (e.g } \text{RelDB.Table.clmpky}) \text{, the target (e.g., } Tid \text{) , the source (e.g., } Tpk \text{), and the variable (e.g., } Ufa \text{) which will hold this newly created relation/function.}

Control Structure. To control the execution order and mode of graph transformation the VTCL includes some concepts that support the definition of complex control flow. As one of the main goals of the development of VTCL was to create a precise formal language, a set of Abstract State Machine (ASM) language constructs was included that has formal semantics and correspond to the constructs in conventional programming languages.

Now, I briefly present how ASM rules and control structures can be defined in VTCL.

- **Random rule** = random \{ \{asmrule call\} \}
- **Parallel rule** = parallel \{ \{asmrule call\} \}
- **Sequential rule** = seq \{ \{asmrule call\} \}
- **Forall rule** = forall variable [ (in | below) name] [variable [ (in | below) name]] (with gt pattern call do asmrule call | do gtrule call)
The first form means that the rule is executed for any (combination of) variable substitutions that satisfy the formula. The second one means that the graph transformation rule is executed for all possible matchings of its pattern.

The scope of each variable can be narrowed to a specific container, or to a specific part of the model space. With the use of the in clause users can specify a container where the values of the variables can be taken from. Similarly, the below clause means that the values of the variables must be taken from the model tree below the given container.

- **Choose rule** = choose variable [(in|below) name] {variable [(in|below) name]} with gt pattern call do asmrule call

For a combination of non-deterministically selected values in the variables for which the pattern matching is successful, the **asmrule call** is executed. The scope constraint is the same as in the **forall rule**.

- **Iterate rule** = iterate (asmrule call)

- **Let rule** = let variable = (name | value) in asmrule call

- **Skip rule** = skip

- **Update rule** = update (function name ( variable location ) = asm value | variable = asm value)

- **Call rule** = call rule name ( term list ) ;

- **Conditional rule** =
  rule 1 = asmrule call
  rule 2 = asmrule call
  if rule = if ( formula ) rule 1 [else rule 2].

**Formulas** are Boolean expressions constructed from terms (i.e., constants, variables, or element references) using relational operators (==,<,>,!=,<=,>=), or from other formulas using logic operators (&&,||,−>). Graph pattern calls are a special type of formulas.

- **ASM function** = asmfunction function name { ( location ) = value ; }.
  Where location is the identifier of the value. A value can be a constant, a variable or an ASM reference.

There are two rules that are not part of the ASM but important, for the VTCL language: **log rule** and **print rule**. There is not to much to say about the **print rule**, string type data (constant, model element name and value) can be written to the standard output. String type data can be concatenated with the + function. The **log rule** works the same as the **print rule**, but it writes to a log file of the framework, and a **log level** also has to be passed as an extra parameter. The syntax of these rules are:
• **Print rule** = print ( value )

• **Log rule** = log ( log level , value )
  log level = fatal | error | warning | info | debug

These basic instructions, combined with graph patterns and graph transformation rules, form an expressive, easy-to-use, yet mathematically precise language where the semantics of graph transformation rules are also given as ASM programs (see [30] for more details).

**Example 8** The following example demonstrates the main control structures.

pattern isColumn(C) = {
  //simple pattern that recognizes columns
  RelDB.Column(C);
}

rule makeTable(in P, in C) =
  iterate seq {
    print("The transformation begins..."); //Print out some text
    //Call a GT rule
    apply classR(P,C);
    call printFormatted(123); //Call other rule
    //Iterate through all wrong columns
    forall Cl with find(wrongColumn(Cl)) do seq {
      print("Found wrong column: "+name(Cl));
    }
    log(info,"transformation done"); //Write to log
  }

rule printFormatted(in C) = {
  print("Value is : "+C); //Print out the value
}

Figure 2.8: ASM example

1. The rule has 2 incoming parameters, named C and P.

2. The transformation **makeTable** first iterates over a sequence of steps. When the application of any rules within the sequence fails, the execution of the **iterate** construct terminates.

3. The first step in the sequence of the transformation is to write the text "The transformation begins..." to the standard output.

4. Then **apply** calls the GT rule **classR** with two incoming parameters C and P. The acceptance criteria for the incoming parameters are prescribed by the condition part of the GT rule (e.g., condition pattern). If the pattern matching is successful, then the action part of the GT rule is executed. If there are no matches for the pattern, then the iterate sequence terminates, and the rule returns.
5. Then `printFormatted` rule is called with an integer parameter 123, and it writes the text "Value is : 123" to the standard output.

6. The `forall` iterates over all the elements one by one and tries to satisfy the GT pattern `wrongColumn`. If an element satisfies the pattern then its name is written to the standard output.

7. And the final step writes "transformation done" to the log together with the priority of `info`. 
Chapter 3

Generation of Java specific transformations

**Overall conceptual constraints.** As discussed earlier, Viatra2 uses a simple, generic representation for meta-models and models. While this formalism is suitable for the simultaneous, design-time representation of meta-models and models taken from multiple domains, it is not optimal from a performance perspective for the representation of a single (domain-specific) meta-model and its instance models.

Although Viatra2 supports the definition of multiple meta-levels, we restrict the PIT generation to two neighboring meta-levels (e.g. the model-level and the meta-level). This is suitable for most practical MDSD transformations such as PIM-to-PSM mappings, model analysis or code generation. We assume that the meta-models are fixed, and model manipulations are only carried out on model level.

This enables the generation of a static class structure in the target implementation platform (i.e., classes, methods, and attributes) from the meta-level elements. The model-level elements will be instances (objects) of these classes. This type of approach allows the easier integration of the generated class structure to existing programs.

3.1 Mapping VPM models to Java specific models

As described earlier, the basic VPM constructs of the meta-models have to be transformed to constructs of the target platform.

- VPM entities are mapped to Java classes. For every entity an *interface* is also generated, so the inheritance between entities are maintained by these *interfaces*. This kind of approach is a must, because multiple inheritance is allowed in VPM. In case of multi-level inheritance all the interfaces of the above elements are implemented by the class.

- A VPM function is mapped to a single scalar Java attribute of type `FunctionHolder` (see Appendix [B.2]) in the Java class that corresponds to the source of the function. The `FunctionHolder` has a name and a value attribute. The *name* is a String type
and holds the name of the function in the instance model, while the value holds the target element.

- A VPM relation is mapped to a HashMap attribute in the Java which class corresponds to the source of the relation. The key in the HashMap is the name of the relation in the instance model, and the value will be the target element.

Example 9 An extract of a generated Java class (for the complete program see Appendix A.1) from the VPM entity Assoc:

```java
public class Assoc implements ClassInterface, EntityInterface, NameInterface, AssocInterface {
    // Interfaces from the inheritance and from entity
    public String Value=null; // Basic entity attributes
    public String Name=null;
    public static Vector Instances=new Vector();
    public Vector Below=null; // for capturing all Assoc type element
    public Vector Sources=null;

    // Constructors
    public Assoc(){...}
    public Assoc(String n, String v){...}

    // Entity functions
    public String getName(){return Name;}
    public void rename(String s){Name = s;}
    public String getValue(){ return Value;}
    public void setValue(String s){ Value = s;}
    public static putInstance(Object value){...}
    public static Collection getInstances(){return Instances;}
    public void addBelowElement(Object o){...}
    public void deleteBelowElement(Object o){Below.remove(o);}
    public Collection getIn(){return Below;}// for the containment hierarchy
    public Collection getBelow(){...} // and for the GT pattern below, and in
    public void deleteSource(String field,String name, Object o){...}
    public void delete(){...}

    // FUNCTIONS
    private FunctionHolder AssPck=new FunctionHolder();
    public Object getAssPck() { return AssPck.get(s);}
    public void renameAssPck(String old , String s){...}
    public void setAssPck(String name,PackageInterface a){...}
    public void deleteAssPck(String a){...}
    public void deleteAssPckall(){...}

    // RELATIONS
    private HashMap AssAse=new HashMap(); // The AssAse relation of the Assoc entity
    public HashMap getAssAse(){return AssAse;}
    public void setAssAse(String name,AssocEndInterface a){...}
    public void deleteAssAse(String a){...}
    public void renameAssAse(String oldName,String newName){...}
    public void deleteAssAseall(){...}
```


CHAPTER 3. GENERATION OF JAVA SPECIFIC TRANSFORMATIONS

// SuperTypes relations & functions

//FUNCTIONS OF ClassInterface
private FunctionHolder ClsPck=new FunctionHolder();
public Object getClsPck() { return ClsPck.get(s); }
public void renameClsPck(String old, String s) {...}
public void setClsPck(String name, Packagenterface a) {...}
public void deleteClsPck(String a) {...}
public void deleteClsPckall() {...}

• Every class implements at least three interfaces: NameInterface, EntityInterface (see Appendix B.1), and his own interface. NameInterface defines the Name attribute managing functions and the delete functions, EntityInterface defines the functions for managing the Value, Sources and Below attributes. Every further interface – except his own – is an interface of an inherited model element.

• For each relation and function five managing methods are defined:
  – The getXYZ returns the target element and the Collection of target elements in case of VPM functions and VPM relations, respectively
  – With the setXYZ method the input parameter can be added. In case of a VPM function it replaces the old one, while in case of a VPM relation it is added to the HashMap.
  – The deleteXYZ function deletes the VPM function/relation.
  – RenameXYZ can be used to replace the name (in PSM) of the target element.
  – deleteXYZall can be used to delete all the elements of the XYZ relations/function. This method is only invoked in the delete method of the current entity.

• Name and Value attributes of VPM entities are stored in String attributes having the same name.

• The static Instances attribute stores the instances of the corresponding Java class. The getInstances function returns a Collection with the instances of the class, while putInstance can put a new element into the Collection.

• In order to maintain containment relations between entities, each object has a special attribute called Below (type of java.lang.Vector). This attribute holds the elements of the next level of the containment tree. Four methods are connected to this attribute.
  1. The getIn method returns the Below Collection (only the next below level elements of the containment tree),
  2. The getBelow returns a Collection with all the elements of the below containment tree (recursively calling getBelow, on the elements in Below).
  3. The addBelowElement methods adds the input parameter to the Below vector.
4. The `deleteBelowElement` method removes the input parameter from the `Below` vector.

- The `Sources` attribute stores all the relations/functions whose target element is the current entity. The functions/relations are stored in a container class called `SourceHolder` (see Appendix B.3), this container stores the type, the name, and the source of the function/relation. Two methods are connected to this attribute.
  1. The `deleteSource` method deletes the stored value from the `Sources` vector, if each input parameter matches an element of the vector.
  2. The `addSource` method adds a new element to the `Sources` vector.

Every time a relation or function is manipulated, the appropriate method or methods are invoked to maintain the consistency of the `Sources` vectors in the model space.

The `delete` method can erase the current entity from the model space. It erases all the below elements in the containment hierarchy.

When relations/functions are manipulated, then they are held by a special container class called `ConHolder` (see Appendix B.2). A function/relation is added into a `ConHolder` object when it is passed as an output parameter from a GT pattern or GT rule. The class implements the `NameInterface`.

### 3.2 Mapping PIT to PST

This section describes the generation of Java specific transformers from the PIT description using the meta-model specific model manipulation API derived in Sec. 3.1. As described in Sec. 2.4.2, Viatra2 uses a two level approach for designing transformations, for basic model manipulating GT rules are used, and to form complex transformations ASM gives the control structure, where GT rules are treated as simple ASM rules. The first part (in Sec. 3.2.1 and 3.2.4) of the section introduces the mapping of the ASM constructs, while the second part (in Sec. 3.2.6) focuses on the mapping of GT rules.

ASM programs (in VTCL) have a close correspondence with object-oriented technologies, that makes the generation easier, but there are some critical parts that are basically different from OO technologies like handling variables. The other problematic part in the mapping to Java, is the integration of pattern matchings with the control sequences (forall, choose, GT rule call, etc.).

#### 3.2.1 ASM machine and rule definitions

One transformation is represented by one ASM machine. A machine is transformed to two Java classes. One class (Rule Class, `RC`) holds the representation of the ASM rules, functions, and the action part of the GT rules, while the other (Pattern Class, `PC`) holds the representation of all the GT patterns used in the machine.
• **Machine definition:** The RC is a container of the derived ASM and GT rules. Three global variables are defined, that can be accessed from any method. Random-Generator and randomValue are auxiliary variables for the random rule. The log is an instance of the Log4j [2] class. This object is responsible for logging.

```java
public class Graph2Java {

    Random randomGenerator= new Random(System.currentTimeMillis());
    int randomValue;
    log = new SimpleLogger();
    //methods and variables defined
}
```

The PC is discussed later in Sec. 3.3.

• **Rule definition:** Each rule is mapped to a Java method. The name of the method is equal to the name of the rule. Input parameters of a rule are typically mapped to input parameters of a method, and the output parameter of a rule is mapped to the return value. This solution is unfeasible, since there can be several output parameters. Moreover, an inout parameter can be considered as both input and output. Therefore, a Java HashMap is used for passing rule parameters in each directions. There are four other special HashMaps in each method. Inp and Out attributes are used, when an other method is invoked, the input parameters of the invoked method are passed with Inp, and the output parameters are captured by the Out. The Variable HashMap holds the actual variables inside the method, and the Scope Stack is used to maintain the scope of the actual variables.

```java
public HashMap ruleName(HashMap Input)
{
    HashMap OutPut = new HashMap(); //holder of the output variables
    //all variables are stored in this HashMap
    HashMap Variables = new HashMap(Input);
    //other rule invocation input parameters stored in this
    HashMap Inp = new HashMap();
    HashMap Out = new HashMap();
    //used to maintain the scope of the variables
    Stack Scope= new Stack();
    ...
    OutPut.put("P",Variables.get("P"));
    return OutPut; // returns the output parameters can be empty HashMap
}
```

### 3.2.2 Variables and ASM functions

Variables in ASM rules are mapped to Java variables, but in VTCL there is no need for implicit variable declaration, as a consequence, this simple mapping between variables cannot be used. To overcome this problem, a Variable HashMap is used to handle the variables:

- To insert a value into an already existing variable or into a new one simply the put(name, value) method is used,
• to get a value of a variable the get(name) is used, where if a name cannot be found in the Variables HashMap then a null value is returned.

In order to handle the scope of the variables correctly we maintain a Scope Stack. Before every block rule (forall, choose and let rule) a shallow copy (only references are copied) of the Variables is pushed to the stack, and after the block rule the old Variables HashMap is popped from the stack. Because of the copy, the variables inside the block can be manipulated, and has no effect on the outside variable.

ASM functions are the global variables of the VTCL. An ASM function is derived to a HashMap, and it is contained by the RC. The key of the ASM function is mapped to the key of the HashMap.

```java
public class Graph2Java {
    //classes is an ASM function
    HashMap classes = new HashMap();
    public HashMap rule1(HashMap Input) {
        HashMap Variables = new HashMap(Input);
        Stack Scope = new Stack();
        asmfunction_name.put("Pck", Variables.get("Package"));
        Scope.push(new HashMap(Variables));
        {
            //block rule begins
            //local scope only
            Variables.put("new_Package", classes.get("Pck"));
            ...
        }
        //block rule ends
        Variables = (HashMap) Scope.pop();
        //a null will be the value of "nPack"
        Variables.put("nPack", Variables.get("new_package"));
    }
    //other methods
}
```

3.2.3 ASM rules

These basic rules and control structures have been mapped to native Java to exactly follow the execution of the parallel ASM program in the VIATRA2 framework.

• Skip rule: This simple rule is mapped to an empty statement.

• The Update rule can be called on variables and ASM functions. Both of them are represented in PST by a HashMap, so the update rule is mapped to a function call put(key, value).

```java
Variable.put("Var2", classes.get("Var1"));
classes.put("Var1", "newValue");
```
• **Call rule:** Each user defined ASM rule is mapped to a Java method, so the call rule, which invokes an other user defined ASM rule, is mapped to the corresponding method invocation in Java. As described earlier input and output parameters are handled with `Inp` and `Out` HashMaps.

```java
Inp = new HashMap();  // fitting the input parameters
Inp.put("C","123");  // to the Inp HashMap
Out = printFormatted(Inp); // the output parameters are captured with the
// Out HashMap
```

Each output variable is copied to the `Variables HashMap` one by one.

• **Random rule:** A huge switch simulates the random block, and the `randomGenerator` returns the number of the rule which will be called. The `randomValue` is a random number, and it can take a number from zero to `number`, where `number` is one smaller than the number of rules.

```java
randomValue = randomGenerator.nextInt(number);
switch(randomValue){
    case 0 : //ASM rule
        break;
    case 1 : //ASM rule
        break;
    // other cases of the switch
    default : throw new FatalError("Wrong random number");
}
```

• **Parallel rule:** A random order of the rules is generated from the parallel block of the VTCL to imitate the parallel behavior, but these random rules are executed sequentially.

• **Sequential rule:** Because Java has a sequential execution order, the rules in the `seq` block are transformed in the same order as in the Viatra2 framework.

```java
print("Classname:"+name(P));  //rule1
call printFormatted("Second"); //rule2
```

• **Iterate rule:** The as long as possible rule is implemented by an infinite loop. The only way to exit the loop, is to throw a `RuleFailedException`.

```java
//iterate catches
try{
    while(true)
    {
        // this rule can end the iterate block, if the pattern
        // matching is failed.
        Inp.put("C", Variables.get("C"));
        Out = wrongColumn(Inp);
    }
    catch(RuleFailedException e){}
```
• Let rule: New variables can be declared with the let rule. The scope of the newly declared variables are handled by the Scope stack.

```java
Scope.push(Variables.get("Var1")); \saving the outside Variables
Variables.put("Var1",Out.get("newVar1"));
...
//rules inside let block
...
Variables.put("Var1",Scope.pop()); \end of let block
```

• Print rule: It writes to the standard output.

```java
System.out.print("Transformation␣begins...");
System.out.print("Err"+((NameInterface)Variables.get("Cl")).getName());
```

• Log rule: As described earlier in Sec. 3.1 the log rule is derived to the Apache logger for Java. For each log level a different method is used, for example for information level the info method is called.

```java
//the different log levels
log.info("Starting...");
log.fatal("Fatal");
log.error("Nullpointer_exception");
log.warning("Variable_empty");
log.debug("value: ");
```

• Conditional rule: The rule is mapped to the same if construct as in the VTCL.

- The relational operator "==" is mapped to the equals() method in case of string variables.
- The binary implication operation "A → B", is mapped to the "!A or B" Java expression.
- If there is Graph pattern call in the condition part of the rule, then a special method (which returns only a boolean) is generated for the pattern.

3.2.4 Control structures

Rules that are connected to GT patterns, are generated in a special way, because not only the Java source code is generated, but a special graph (context graph, CG) is also built up in the model space for each pattern or pattern call appearing in a choose or a forall context. These graphs hold all the information about the context of the GT pattern call, which will be discussed in Sec. 3.3.6.

• Forall rule: As described in Sec. 2.4.2 there are mainly two types of forall rules in VTCL. Each is mapped to Java in a slightly different way.

1. If the forall used with a GT pattern, then the generated code structure is as follows:
CHAPTER 3. GENERATION OF JAVA SPECIFIC TRANSFORMATIONS

```java
Iterator iter1 = Patterns.wrongColumn1().iterator();
Scope.push(new HashMap(Variables));
while(iter1.hasNext())
  {Variables = (HashMap)Scope.peek();
   Variables.putAll((HashMap)iter1.next());
   //Other rules
  }
Variables = (HashMap) Scope.pop();
```

First, the pattern method being generated for this `forall` rule is called (e.g. `WrongColumn1`), and returns a Collection which contains HashMaps. Each HashMap holds a matching graph segment. The `while` goes through every matching graph segment and executes the rules defined inside the `forall` rule. In each iteration, the variables are reset to the beginning state, with the `peek()` method.

As mentioned in Sec. 3.1, the `forall` is a scoped rule, so the `Variables` are saved before the rule begins, and popped after the rule is finished `Scope stack`.

2. When a `GT` rule is used in a `forall` context, the HasMaps representing the matching graph segments, are reused as input parameters for the method representing the `GT` rule. In every other aspect the two code segments are the same.

```java
//precondition pattern classR GT rule
iter2 = Patterns.lhs_forall1(Variables.get("P"), Variables.get("C"));
Scope.push(new HashMap(Variables));
while(iter2.hasNext())
  {Variables = (HashMap)Scope.peek();
   classR_action((HashMap)iter2.next()); //the call of the GT pattern
  }
Variables = (HashMap) Scope.pop();
```

- **Choose rule:**

  Choose rules can only be used with `GT patterns`. For each rule in the ASM program a unique pattern method is generated. These methods return only a `HashMap` containing only one non-deterministically selected graph segment, that completes the matching. If there is no matching graph segment then a `RuleFailedException` is thrown.

```java
//precondition pattern classR GT rule
HashMap hmap3 = Patterns.wrongColumn2(); // the pattern call
Scope.push(new HashMap(Variables));
Variables.putAll(hmap3); //out parameters are initialized

//rules of the choose
```

```java
Variables = (HashMap) Scope.pop(); //end of the choose rule
```
CHAPTER 3. GENERATION OF JAVA SPECIFIC TRANSFORMATIONS

- **Apply rule:**
  
The apply rule is very similar to the Choose rule as it is used to call a GT rule for only one non-deterministically selected graph segment. If there are no matching graph segments, then a RuleFailedException is thrown. The only difference is that the output parameters of the GT rule’s pattern method generated for the GT rule, are the input parameters of the action method derived from the GT rule.

\[ \text{Out} = \text{classR.action(Patterns.lhs(Variables.get("P"),Variables.get("C")))}; \]

The lhs pattern is called ed with two input parameters C and P, and the output of the lhs method is the input of the classR action method.

3.2.5 Model manipulation rules

- **Element creation rules**

There are two types of model element creation rules. One is for creating a new VPM entity, and the other is to create a VPM relation/function. Because the representation of the VPM entities are Java classes, a simple class instantiation is used, where a new entity is created. In case of function/relation creation the setXYZ method of the source class is called, where XYZ is the name of the function/relation.

```java
//a new Column class is created and The variable Tid holds
//a reference to this class
Variables.put("Tid",new Column());
// new Pkey in variable Tpk
Variables.put("Tpk",new PKey());
//The element in the Tpk variable is put
//in the containment hierarchy tree under the element
//held by the Tid variable.
((EntityInterface)Variables.get("Tid")).addBelowElement(Variables.get("Tpk"));
...
// Clmpky relation is created from the Tid to the Tpk element.
// The name of this function is randomly generated,
// because the user doesn’t gave one
((ColumnInterface)Variables.get("Tid")).setClmpky(RandomName.uniqueName(), Variables.get("Tpk"));
//if the connection type model element is later used, it has to be save
//for further manipulation into a holder class
Variables.put("Ufa",new ConHolder("Clmpky", getKeyHashMap((ColumnInterface)Variables.get("Tid").getClmpky(), Variables.get("Tpk")),Variables.get("Tid")));
```

If a VPM entity (e.g., Tpk) is put into a given container in the containment hierarchy, the addBelowElement method is called on the container (e.g., Tid) class, with the input parameter of the VPM entity.

If a VPM function/relation type model element is manipulated later in the program, then it is passed to the container class ConHolder.
• Element deletion rules
To delete a model element, the delete method is called on the variable, which holds the model element. The delete method is in the NameInterface, so all the VPM entity classes and the ConHolder class implement it.

```java
((NameInterface)Variables.get("Cl")).delete();
```

• Auxiliary rules.
The methods implementing the functionality of these rules are located in the Java class of each VPM entity. These rules manage the name attribute of the model elements and the value of the VPM entities.

– SetValue / getValue
Methods setValue (getValue) sets (retrieves) the value of an entity.

```java
String s = ((EntityInterface)Variables.get("Cl")).getValue()
... ((EntityInterface)Variables.get("Cl")).setValue("new value");
```

– getName All classes of the model-space implement the NameInterface, simply to get the name of the element invoke the getName method.

– rename To rename a model element the method rename is called. This method is part of the NameInterface.

3.2.6 GT rule definition
As described in Sec. 2.4.2, the GT rule has two parts: a precondition and an action. Each part is mapped separately to a Java method. The action part is mapped like a standard user defined ASM rule, while the precondition part is mapped to as many Java representation of pattern matching method as many different form (e.g., with forall rule, with apply rule) of the GT rule were called in the ASM program. The generation of the pattern matching methods are discussed later in Sec. 3.3.6.

Here is an example, which shows the generated method for the action part of the GT rule classR:

```java
public HashMap classR_action(HashMap Input) throws RuleFailedException {
    HashMap OutPut = new HashMap();
    HashMap Variables = new HashMap(Input);
    HashMap Inp = new HashMap();
    HashMap Out = new HashMap();
    Stack Scope= new Stack();

    //model manipulation rules
    Variables.put("Tid", new Column());
    Variables.put("Tpk", new PKey());
    Variables.put("T", new Table());
    ((TableInterface)Variables.get("T")).setTblpky(RandomName.uniqueName(),
```
Variables.get("Tpk");
((ColumnInterface)Variables.get("Tid")).setClmpky(RandomName.uniqueName(), Variables.get("Tpk");
((PKeyInterface)Variables.get("Tpk")).setPkyclm(RandomName.uniqueName(), Variables.get("Tid");
((TableInterface)Variables.get("T")).setTblclm(RandomName.uniqueName(), Variables.get("Tid");
((ClassInterface)Variables.get("C")).setClstbl(RandomName.uniqueName(), Variables.get("T");
((TableInterface)Variables.get("T")).setTblcls(RandomName.uniqueName(), Variables.get("C");
((PackageInterface)Variables.get("P")).setPcksch(RandomName.uniqueName(), Variables.get("S");
((SchemaInterface)Variables.get("S")).setSchtbl(RandomName.uniqueName(), Variables.get("T");

return OutPut;
}

The `uniqueName` method generates a random name for the entity, if the user does not give one explicitly.

### 3.3 Mapping graph pattern matching techniques to Java

#### 3.3.1 Search plan generation

The most critical step for the performance of graph transformation is the graph pattern matching phase. In the field of graph theory, this problem is also known as the subgraph isomorphism problem, and it is known to be NP complete [19].

The pattern matching is determined by only the precondition of a graph transformation rule. For this purpose, the generation of search plans is a frequently used and efficient strategy. Informally, a search plan defines the order of traversal (a search sequence) for the nodes of the instance model to check whether the pattern can be matched.

The search space traversed according to a specific search plan is represented as a search space tree [35] (SST) which contains all the decisions that can be made at a certain point during pattern matching. The root node of a SST represents a partial matching as provided by fixing the input parameter nodes of rules. Each path of a SST starting from the root node extends this partial matching by the matching of a fresh (unmatched) node in the pattern.

The current section presents a search plan generation technique in the following way.

1. First, the concept of search graphs and weighted search graphs and the building process of search graphs in Viatra2 is introduced in Sec. 3.3.2.

2. The concept of search plans is defined together with a cost function that helps estimating the performance of search plans and formulating when a search plan is optimal in Sec. 3.3.3.
3. An overview of the currently available and widely used search plan evaluating algorithms is given in Sec. 3.3.4.

4. The currently implemented low cost search plan seeking algorithms are introduced in Sec. 3.3.5.

5. Finally, the mapping of a pattern to Java is presented in Sec. 3.3.6.

### 3.3.2 Search graphs

In the first phase of the search plan generation process, a search graph is created for each pattern.

A *weighted search graph* is a directed graph with numeric weights on its edges. (Weights are denoted numeric labels on edges in figures.) The weighted search graph has the following structure. Each node of the pattern is mapped to a node in the search graph. We also add a *starting node* to the graph.

1. Directed edges connect the starting node to every other search graph nodes. If the target graph node has already been bound, then the weight of the corresponding search graph edge will be 0. If the target node is unbound, then the edge weight will be 20. When the latter type of edge is selected in the search graph for a certain search plan, then the graph pattern matching engine executing this search plan needs to iterate over all objects in the model of the corresponding type.

2. Each VPM function and relation of the pattern is mapped to an edge of the search graph that connect the corresponding end nodes. Weights of search graph edges originating from a VPM function and a VPM relation are 1 and 10, respectively. An edge can be selected by the pattern matching engine only when the source pattern node is already matched. In this case, the selection of such a search graph edge means a navigation along the corresponding pattern edge towards the unmatched (target) pattern node.

Search graphs for negative application conditions can be handled similarly. In this case, all the matched nodes (i.e., the ones that are shared with LHS graphs) have to be considered as starting nodes. Negative application conditions are typically checked after a complete matching has been found for the LHS.

**Example 10** The precondition pattern of the graph transformation rule `classR` of Example 3 is shown in Fig. 3.1(b). The weighted search graph that corresponds to the LHS of this precondition pattern is depicted in Fig. 3.1(a).

**Building up the search graph.** In our approach for each GT pattern and in case of `forall` and `choose` GT pattern call a search graph is built up, and a pattern matching Java method is generated.

The context graph (CG) (for meta-model see C.4) stores the information about the context (input parameters, output parameters, type of caller (GT rule, choose,
CHAPTER 3. GENERATION OF JAVA SPECIFIC TRANSFORMATIONS

(a) Search graph for classR
(b) Precondition of classR

Figure 3.1: A sample graph transformation rule and its corresponding search graph

forall, if and a reference to the pattern) of the GT pattern/pattern call. The main
core of the graph is the patternMatch, all other nodes are connected to this one. The type
entity stores the type of the pattern caller, while all the other information is stored in the
bound entities. These graphs are the input parameters of the search graph construction
algorithm.

The constraints below and in are added into the search graph in the following way. If
a node \( n \) has one of these constraints, then a new node \( m \) (which represent the container
element in the constraint) is added to the search graph in a special way, because this node
\( m \) is not connected to the center node, only one edge \( e \) connects it to the search graph.
This source of edge \( e \) is the node \( n \). The weight of the edge \( e \) is 15 and 5, in case of
below and in constraints, respectively. These nodes are treated in the whole search plan
generation as ordinary nodes, except for the code generation (see in Sec. 3.3.6).

The algorithms are running on this kind representation of the search graph (for the
meta-model of the search graph see Appendix. C.5).

3.3.3 Search trees and plans

At this point, a weighted search graph is available. In this section, first, we introduce the
concept of search trees and search plans based on weighted search graphs. Then a cost
function is defined for search plans to predict its performance.

A search tree is a spanning tree of the weighted search graph. As the starting node
has no incoming edges, all other nodes should be reachable on a directed path from the
starting node.

A search plan is one possible traversal of a search tree. A traversal defines a sequence
in which edges are traversed. The position of a given edge in this sequence is marked by
increasing integer numbers written on the thick edges in Fig. 3.2. A sample search plan
(with its corresponding search tree) is shown in Fig. 3.2

The cost of a search plan (denoted by \( w(P) \)) is defined as
\[
    w(P) = \sum_{i=1}^{n-1} \prod_{j=1}^{i} w_j,
\]

where \( w_j \) is the weight of the edge labeled with integer \( j \), and \( n \) is the number of
edges in the search plan.
where $w_j$ is the weight of the $j^{th}$ edge according to the order defined by the search plan and $n$ is the number of nodes of the search plan. Based on our special edge weight definitions, minimum cost search plans try (i) to select bound variables, (ii) to traverse VPM functions, (iii) in constraint edges, (iv) VPM relations, (v) below constraint edges, and (vi) to iterate over all VPM entities of a given type, in this priority order. By using this cost function on the search plan of Fig. 3.2 on, we get cost value 35.2.

3.3.4 Overview of the search plan evaluating algorithms

As mentioned in Sec. 3.3.1 graph pattern matching is NP complete. As a consequence in practice, finding a match for a given rule is exponential in the size of the LHS
\footnote{the size of the graph is given by the sum of its nodes and edges}. Of course, an algorithm with exponential time complexity is unacceptable for practical implementations, but fortunately it is possible to considerably reduce the average-case complexity with heuristics.

Currently there are two main approaches for graph pattern matching:

1. Algorithms based on constraint satisfaction \footnote{Constraint satisfaction problem (CSP), used mainly in artificial intelligence, is a very promising approach in the field of pattern matching, capable of handling exponential complexity problems with backtracking algorithms. The definition of the Constraint Satisfaction Problem is the following:}

   1. Algorithms based on constraint satisfaction \footnote{Constraint satisfaction problem (CSP), used mainly in artificial intelligence, is a very promising approach in the field of pattern matching, capable of handling exponential complexity problems with backtracking algorithms. The definition of the Constraint Satisfaction Problem is the following:}

   \begin{itemize}
   \item a finite set of variables $X = \{x_1, \ldots, x_n\}$,
   \end{itemize}

\begin{definition}
A CSP consists of
\end{definition}

\begin{itemize}
\item a finite set of variables $X = \{x_1, \ldots, x_n\}$,
\end{itemize}
• a finite and discrete domain $D_i$ of possible value for every variable $x_i \in X$, and

• a finite set constraints on the variables of $X$.

A constraint comprises the values, a variable is allowed to take with respect to other variables. A constraint is satisfied if the instantiation of the variables is taken from the Domain allowed by the constraint. A CSP is satisfied if all constraints are satisfied.

The analogy to the constraint satisfaction problem is quite oblivious: In graph pattern matching we are also looking for a mapping between two sets, namely between the a set of variables and a set of values, where some restrictions apply which is called constraints. The CSP equivalent of a GT pattern can be described as follows:

• The nodes of the pattern graph is mapped to a variable of the CSP.

• The edges of the graph are treated as constraints on the nodes.

• The domains of the variables are the nodes of the instance graph.

• Instead of unary type constraints on the node elements, the not proper values are deleted from the Domains. This way it is avoided to check the unary type constraints over and over again dynamically.

This way the graph pattern matching can be described as a CSP.

The most widely used approach to solve a constraint satisfaction problem is backtracking. Backtracking essentially performs a depth-first search on the search space of potential solutions. The good thing about backtracking is that whenever a constraint check fails for a variable, it eliminates an exponential sub-problem [17]. This leads to the first fail principle, that variables with the highest probability of instantiation failure should always come first.

While research activity is lively in the area, results are manifold already. Hence, representing and solving a given graph pattern matching problem as a CSP has currently no widely useable, efficient solution (a detailed comparison of different algorithms is discussed in [34]). This is the main reason why I did not based my transformator on CSP, but in the future as an alternative pattern matching solution it could become a part of the generated plugin.

Local searches. There are already a huge number of algorithms based on local searches and most of the graph rewriting systems are using this kind of approach at their pattern matching phase.

Local search based algorithms can be categorized according to the graphs on which are they working on:

1. Algorithms running on the weighted search graph, searching for a low cost search plan. Using only the structure and the weights of the pattern graph for the optimization. The majority of algorithms belongs to this group.
2. Algorithms using an operation graph \[37\] for the optimization. These operation graphs are built from the pattern graph and contain nodes representing operations such as GetTarget, TestEdge, etc. These operations are to check the constraints given by the pattern. As the operation graph is built up it contains all possible execution of the pattern matching. The objective is to generate a possible low cost search plan of the operations, where the weight of the graph are based on the resource necessity of the operations. For example a TestEdge operation using 3 times more resources during pattern matching then a GetTarget, then it can be better to use GetTarget often but not always, as sometimes a TestEdge can replace more then 3 GetTarget operations.

According to the comparison discussed in \[34\], I chose a simple weight search graph based algorithm proposed in \[35\] as a basis ground for a model-sensitive adaptive algorithm.

### 3.3.5 Implemented algorithms for finding a low cost search plans

Two traditional greedy algorithms are used to solve the problems of finding (i) a low cost search tree for a given weighted search graph and (ii) a low cost search plan for a given search tree. Traditional algorithms use a different cost function (i.e., the sum of weights) for determining the cost of a spanning tree, which means that their solutions are not necessarily optimal in this case.

**Finding a minimum search tree.** For finding a minimum search tree in a weighted search graph, the Chu-Liu / Edmonds algorithm \[8, 10\] is used. This algorithm searches for a spanning tree in a directed graph that has the smallest cost according to a cost function defined as the sum of weights. This algorithm can be outlined in Alg. \[\] .

**Algorithm 1** Given a weighted search graph with a starting node.

**Step 1:** Discard the edges entering the starting node.

**Step 2:** For each other node, select the incoming edge with the smallest weight. Let the selected \(n - 1\) edges be the set \(S\).

**Step 3:** If there are no cycles formed by the edges of \(S\), then the selected edges constitute a minimum spanning tree of the graph and the algorithm terminates. Otherwise the algorithm continues.

**Step 4:** For each cycle formed, contract the nodes in the cycle into a pseudo-node \(k\), and modify the weight of each edge entering node \(j\) in the cycle from some node \(i\) outside the cycle according to the following equation.

\[
c(i, k) = c(i, j) - (c(x(j), j) - \min_l\{c(x(l), l)\})
\]

where \(c(x(j), j)\) is the weight of the edge in the cycle which enters \(j\).
Step 5: For each pseudo-node, select the entering edge, which has the smallest modified weight. Replace the edge, which enters the same real node in $S$ by the new selected edge.

Step 6: Go to step 3 with the contracted graph.

Finding low cost search plan  In case of finding a low cost search plan in a given search tree, a simple greedy algorithm is used, which is introduced in Alg. 2. It is important to mention that the value of the edges are the values before the Chu-Liu / Edmonds algorithm was used.

Algorithm 2 Given a search tree with a starting node.

Step 1: Set the counter to 1 and let $P$ be the set consisting of the starting node.

Step 2: Select all the edges that goes out from the stating node, with 0 weight, and add the target nodes of these edges to $PS$. (The input nodes are traversed first.)

Step 3: Select the smallest tree edge $e$ that goes out from $P$.

Step 4: Set the label of $e$ to the value of the counter.

Step 5: Increment the counter by 1 and add the target node of $e$ to $P$.

Step 6: If the search tree still has a node that is not in $S$, then go back to Step 3.

3.3.6 Mapping the search plan to Java

Now, we have a low cost search plan. This section shows the mapping of a graph pattern to Java, using the search plan, including the constraints from the caller of the GT pattern.

Types of pattern calls. There are four types of methods generated on the basis of the GT pattern call. These methods only differ in the input and output parameters.

1. The output parameter of a forall type method is a Collection. For each complete matching, a HashMap (result) is added to the Collection (CollResult), containing the output parameters of the pattern. At the end the CollResult is returned.

```java
public static Collection wrongColumn1() throws RuleFailedException {
    HashMap result = new HashMap();
    boolean success = false;
    Vector CollResult = new Vector();
    // matching the nodes
    // pattern matching successful
    CollResult.add(new HashMap(result));

    return CollResult;
}
```
2. The output parameter of a **choose type** method is a *HashMap*. The output parameters are collected in the *result HashMap*, if the matching is successful, then the *result HashMap* is returned. In all other cases a **RuleFailedException** is thrown.

```java
RuleFailedException {
    HashMap result = new HashMap();
    boolean success = false;
    //pattern matching
    ...
    //if the matching is successful
    success=true;
    if(success)
        return result;
    else
        throw new RuleFailedException();
}
```

3. A **GT rule type** is similar to the **choose type** or to the **forall type** if it is called by an **apply rule** or by a **forall rule**, respectively. In case of GT rule type, all involved elements in the graph pattern a returned.

4. The output parameter of an **if type** method is a boolean, because it is used in a conditional rule. All negative patterns are mapped to **if type** Java pattern matching method. If the matching is successful a **true** is returned, in other case a **false**.

```java
public static boolean negCondition(Object CFIX)throws RuleFailedException {
    boolean success = false;
    ...
    //if the pattern match is successful
    success = true;
    return success;
    ...
    return success;
}
```

**The mapping of the traversal** The pattern nodes and edges are traversed one by one according to the search plan.

- When an edge representing a *VPM function* is traversed, a single object is navigated, which is retrieved directly from the **FunctionHolder**.

```java
try{
    SchemaInterface S = (SchemaInterface) P.getPcksch();
    ...
} catch (ClassCastException e) {} 
```

- In case of *VPM relation* an iterator is generated from the *HashMap* representing the *relation* to investigate all possible continuations.
CHAPTER 3. GENERATION OF JAVA SPECIFIC TRANSFORMATIONS

Iterator iter_C = P.getPckcls().values().iterator();
while(iter_C.hasNext()){
  try{
    ClassInterface C = (ClassInterface) iter_C.next();
    ...
  } catch (ClassCastException e) {} }

• If a node is traversed by an edge from the starting node, then this node can be a bounded one (input parameter of the GT pattern), or a node without an incoming edge that connects it to the already traversed part of the pattern, so all instances of the VPM entity has to be investigated.

  //CFIX is an input parameter of the Java method
  try{
    ClassInterface Class1 = (ClassInterface) CFIX;
    ...
  // all instances of the Column entity has to be investigated
    Iterator iter_C = Column.getInstances().iterator();
    while(iter_C.hasNext()){
      try{
        ...
      } catch (ClassCastException e) {} }
  } catch (ClassCastException e) {}

These mapping rules generate the main structure of the pattern matching method, but the additional constraints are mapped in the following way:

• The other edges of the search graph which are not part of the search plan, are checked as soon as their target and source nodes, become bounded (except below and in edges). A simple containment or equality check is generated in case of a VPM relation or a VPM function, respectively.

  //Investigate that there is a relation Pckcls from P to Class1
  if(!P.getPckcls().contains(Class1))
    throw new ClassCastException();
    ...
  //Investigate that there is a function Pcksch from P to Schema1
  if(!P.getPcksch().get().equals(Schema1))
    throw new ClassCastException();

• The below and in edges are treated differently from the other edges. As soon as the below/in edge is traversed, a containment check is generated with the proper getBelow or getIn method.

  //Schema1 is in the container of P
  if(((EntityInterface)P.getIn().contains(Schema1)))
    throw new ClassCastException();
    ...
  //Schema1 is below of the P in the containment tree.
  if(((EntityInterface)P.getBelow().contains(Schema1)))
    throw new ClassCastException();
• **Check** conditions are mapped to if-then-else constructs, and used when a complete matching is found.

```java
// every node is bounded
if(((C).getName()).equals("")) // name of the entity C is not empty
    throw new ClassCastException();
```

• Negative pattern conditions **NAC** are checked after a complete matching has been found for the LHS. This is mapped to an if-then-else construct with a pattern method call of the negative pattern.

```java
// negCondition is a derived negative pattern, which returns a boolean
if(negCondition(C));
    throw new ClassCastException();
```

An example search plan in Fig. 3.2 of the pattern of GT rule **classR**, with no input parameters, **choose type** pattern call and P and C as output parameters is shown in Appendix A.3.
Chapter 4

Automated code generation by meta and general transformations

At this point, the mapping of PIM and PIT to Java was introduced in Chapter 3. This chapter briefly introduces the transformations used for the Java source code generation.

**Overall concepts**  
There are 2 different code generating transformations implemented in the Viatra2 framework.

1. **PSM code generator:**
   
   This transformation generates the Java classes and interfaces from VPM models as described in Sec. 3.1.

2. **PST code generator:**
   
   This is a meta-transformation, generating Java classes RC and PC from ASM and GT based transformation descriptions.

### 4.1 PSM code generator

The same code generator transformation is used for both the source and target meta-models. A brief overview of the transformation is provided in Fig. 4.1.

The transformation gets a (source or target) meta-model as its input. It iterates over the entities of the meta-model and invokes the class and interface generation rules on each entity. The corresponding VTCL code is as follows:

```vtcl
rule psmCodeGen(in metaModel) =  
forall Cl below metaModel with find(findVPMEntity(C1)) do seq {  
   // Java class generation  
   call classGen(Cl);  
   // Java interface generation  
   call interfaceGen(Cl);  
}
```

`findVPMEntity` is a GT pattern that matches a single VPM entity.
CHAPTER 4. AUTOMATED CODE GENERATION

Figure 4.1: Overview of meta-model Java generator

**Class generation rule.** This rule generates a Java class for each VPM entity. Its VTCL code is as follows:

```vtcl
rule classGen(in Cl) = seq {
  // Commands for printing the Java code for the consistent part of the class
  call printClassHeader(CL);
  call makeFunction(CL);
  call makeRelation(CL);
  // to implement the super types methods
  forall A with find(supertypes(A,CL)) do seq
    { call makeFunction(A);
      call makeRelation(A);
    }
}
```

ASM rule `printClassHeader` generates those parts of the code (e.g., class header, variables, etc.) that are common for all classes. Then it invokes two rules (`makeRelation`, `makeFunction`) to generate the Java equivalent of the VPM relations and functions, respectively. ASM rule `makeRelation` (`makeFunction`) iterates over VPM relations (VPM functions) and it generates the corresponding Java source code. If the VPM entity has supertypes then the same set of rules are invoked on each supertype VPM entities one-by-one to generate the Java equivalent of inherited functions/relations.

**Interface generation rule.** The interface generation rule is similar to the above-mentioned class generation rule, but now `makeRelationInt` and `makeFunctionInt` rules are invoked instead of corresponding `makeRelation` and `makeFunction` rules, respectively. Furthermore, no code is created for supertype VPM entities in this case.

```vtcl
rule interfaceGen(in Cl) = seq {
  // Commands for printing the Java code for the consistent part of the interface
```
This example shows the ASM rule `makeRelation` to demonstrate how attributes and methods representing VPM relations are generated for Java classes.

The input parameter `CL` is a VPM entity of the meta-model. The `classrelation` GT pattern matches a VPM relation (N) whose source is `CL` and target is `RA`.

The whole VTCL description (including `print` commands) of `makeRelation` ASM rule can be found in Appendix C.1. The presentation of the ASM rules (`makeFunction`, `makeRelationInt`, `makeFunctionInt`) is omitted as they have a similar structure.

### 4.2 PST code generator

The overall process of meta-transformation generating Java code from the platform independent transformation is presented in Fig. 4.2.

The **ASM rule generator** produces a rule class in the Java code and it builds the context graph of GT pattern calls in the ViATRA2 model space. The **GT pattern generator** produces the pattern class based on the preconditions of graph transformation rules and the context graphs.

#### 4.2.1 ASM rule generator

This transformation is mainly an ASM model parser. It starts parsing the ASM machine (input parameter), and it step-by-step evaluates rules and definitions on the underlying
ASM rules \texttt{asmFunctiondef} and \texttt{definitionRule} generate the corresponding Java code segment for ASM function and ASM rule definitions, respectively. Finally, the action part of GT rules are transformed into Java methods according to the mapping introduced in Sec. 3.2.6. The VTCL code for mapping ASM constructs is as follows:

```vtcl
rule parseASMmachine(in BASE) {
    // Commands for printing the Java code for the Rule class header
    // like package name, imports, fixed code segments, etc.

    //the definition of the ASM functions are iterated by the forall
    // and generated by the \texttt{asmFunctiondef} ASM rule
    forall Func below BASE with find(asmfuncdef(Func)) do call asmFunctiondef(Func);

    // This forall iterates over the action part of the GT rules
    // and generate the corresponding Java method
    forall Gt1 below BASE with find(gtRuleAction(Gt1)) do call parseGtRule(Gt1);

    // all the defined ASM rules are iterated through with this forall and parsed
    // by the \texttt{definitionRule}.
    forall Rule below BASE with find(rD(Rule)) do call definitionRule(Rule);
}
```

The \texttt{asmFunctiondef} ASM rule is very simple and contains only a couple of \texttt{print} rules. The \texttt{parseGtRule} and \texttt{asmFunctiondef} ASM rules are very similar as they both (i) generate the header of the equivalent Java method, (ii) handle the output parameters, and (iii) invoke the \texttt{parseRule} on the nested ASM rule.

**ASM rule definition processing.** ASM rule \texttt{definitionRule} processes ASM rule definitions of the PIT. An extract from the VTCL code is the following:

```vtcl
rule definitionRule(in Def) = seq {
    // Commands for generating the Java code of the method header

    // the pattern \texttt{definitionRuleSearch} matches on the ASM rule call (T) of
    // the rule \texttt{definitionDef}
    choose T with find(definitionRuleSearch(Def,T)) do seq
        call parseRule(T);
    // handles to generate the Java code that adds
    // the adequate output parameters (Q) to the output Hashmap in the Java method
    if(find(definitionRuleParameter(Def,Q)))
        choose Q with find(definitionRuleParameter(Def,Q)) do seq
            { call variables(Q); }
}
```

The ASM rule \texttt{variables} adds a Java code segment that inserts the Java equivalent of parameter \texttt{Q} and its value into the \texttt{HashMap} that has been created for storing output parameters of the corresponding method.
ASM construct processing. The ASM rule `parseRule` processes the ASM constructs that are nested into another ASM rule (Rule) originating from the input PIT.

The ASM rule `parseRule` processes the ASM constructs nested into another ASM rule (Rule) originating from the input PIT. As ASM rules can be nested in arbitrary depth, their processing might require recursive invocation.

Two examples are given to illustrate how the Java equivalent of ASM rules are generated. I selected rules `nestedR` and `updRule` for demonstration purposes as they are typical representatives of recursive and non-recursive meta-transformation ASM rules, respectively.

- The `updRule` generates the Java code segment of the ASM update rule. The update rule can be used on a variable or on an ASM function. The GT pattern `updateRule(updateRuleFunc)` matches the input update rule which is called on a variable (ASM function). The `printTerm` rule evaluates the new value of the variable (ASM function), and generates the corresponding Java code segment.

```java
rule parseRule(in Rule) = seq {
  //print rule
  if(find(pR(Rule))) call printRule(Rule);
  //GT rule invocation (apply)
  if(find(gTRI(Rule))) call gtRuleInv(Rule);
  //ASM rule call
  if(find(cR(Rule))) call asmRule(Rule);
  //log rule
  if(find(lR(Rule))) call logRule(Rule);
  // model manipulation rule (new, delete, rename, setValue)
  if(find(mMR(Rule))) call modelmRule(Rule)
  // nested rule (sequential, parallel, random)
  if(find(nR(Rule))) call nestedR(Rule);
  //iterate rule
  if(find(iR(Rule))) call iterateR(Rule);
  //skip rule
  if(find(sR(Rule))) print(";\n");
  //update rule
  if(find(uR(Rule))) call updRule(Rule);
  //block rule (let, choose, forall)
  if(find(bR(Rule))) call blockRule(Rule);
  //conditional rule
  if(find(coR(Rule))) call conditionRule(Rule);
}
```

The ASM rule `parseRule` figures out which kind of ASM construct is nested into the input parameter `Rule` and it invokes the corresponding ASM rule for further processing. As ASM rules can be nested in arbitrary depth, their processing might require recursive invocation.

Two examples are given to illustrate how the Java equivalent of ASM rules are generated. I selected rules `nestedR` and `updRule` for demonstration purposes as they are typical representatives of recursive and non-recursive meta-transformation ASM rules, respectively.

```java
rule updRule(in Rule) = seq {
  if(find(updateRule(Rule,Ter,Var)))
    choose Ter,Var with find(updateRule(Rule,Ter,Var)) do seq {
      print("Variables.put("+value(Var)+"",");
      call printTerm(Ter);
      print(";");
    }
```
• The ASM rule `nestedR` is responsible for the processing of ASM rules `seq`, `random` and `parallel`. (The whole code can be found in Appendix C.2.) As a common feature, these rules must contain nested ASM rules, so the `parseRule` is recursively invoked in order to generate the Java code for the nested ASM rules.

```java
rule nestedR(in R) = seq
{
  if(find(sequentialR(R,Fir)))
    choose Fir with find(sequentialR(R,Fir)) do seq
    { update In = Fir;
      call parseRule(In);
      iterate choose Next with find(ruleNext(In,Next)) do seq
        { call parseRule(Next);
        update In = Next;
        }
    }
  if(find(parallelR(R)))
    forall Next with find(nestedRule(R,Next)) do call parseRule(Next);
  if(find(randomR(R))) seq {
    // Commands to print the switch structure for the random rule
  }
}
```

As other ASM rules use similar techniques, their detailed presentation is omitted from the thesis.

### 4.2.2 GT pattern generator

This transformation generates the platform specific (Java) implementation of graph pattern matching as already described in Sec. 3.3.

The transformation has three phases. (i) In the first phase, a search graph is generated from the input GT pattern also taking into account all constraints on VPM entities of the pattern. (ii) By using Alg. 1 and 2, a low cost search plan is calculated. (iii) Java code is generated based on the search plan. The corresponding VTCL code is as follows:

```java
rule gtPatternGenerator(in CG, in Pattern) = seq {
  searchGraphGen(CG, Pattern, SG);
  searchPlanCalc(SG);
  searchPlan2Code(SG);
}
```
Search graph generation. The first task is to generate the search graph (SG) from the input GT pattern and the constraints on the VPM entities. A \texttt{forall} rule iterates over all the VPM entities of the pattern and generates the corresponding nodes of the search graph. Then the edges of the search graph are generated by invoking GT rules in \texttt{forall} mode on all VPM functions and relations of the pattern. Finally, the constraints on the VPM entities are added to the search graph according to the mapping discussed in Sec. 3.3.2. The corresponding VTCL code is as follows:

\begin{verbatim}
rule searchGraphGen(in CG,in Patt, out SG) = seq {
    //To build the search graph nodes
    forall Ent below Patt do apply entity2SGNode(Ent,No,SG);
    forall Ent below Patt, Node with find(findEntity(Ent,Node)) do seq {
        //to build the function corresponding search graph edges
        forall T below Patt, N ,No do apply function2SGEdge(No,Ent,T,N,No);
        //to build the relation corresponding search graph edges
        forall T below Patt, N ,No do apply relation2SGEdge(No,Ent,T,N,No);
    }
    //to give the in and below constraints
    forall Bel with find(findBelow(CG,Bel)) do
        forall No below SG with find(findNode(No)) do
            if(name(Bel)==name(No))
                call makeNode(No,Bel,"Below");
    forall In with find(findIn(CG,In)) do
        forall No below SG with find(findNode(No)) do
            if(name(In)==name(No))
                call makeNode(No,In,"In");
}
\end{verbatim}

The input parameters of the rule are the context graph and the GT pattern, while the output parameter is the search graph. The first \texttt{forall} iterates over VPM entities of the pattern and generates the nodes of the search graph by invoking the \texttt{classnode} GT rule. The next \texttt{forall} combined with two inner \texttt{forall}s using the \texttt{classfunction} and \texttt{classrelation} GT rules generates the edges of the search graph. In the last part of the rule, the outer \texttt{forall}s iterate over the \texttt{in} and \texttt{below} constraint descriptions of the context graph, while the inner \texttt{forall}s iterate over the nodes of the search graph. In case of name correspondence, a new node is added to the search graph by the \texttt{makeNode} rule.

Example 11 This example shows the GT rule \texttt{relation2SGEdge} (presented in Fig. 4.3) in a graphical notation, which generates the edge of the search graph for a VPM relation. Parameters \texttt{X} and \texttt{T} are VPM entities and they correspond to search graph nodes \texttt{Node} and \texttt{No}, respectively. \texttt{N} is the relation between the two VPM entities in the GT pattern. The VTCL code describing this sample GT rule can be found in Appendix C.3. The other two GT rules are similar to this one.

Search plan calculation. For finding a low cost search plan the algorithms presented in Sec. 3.3.5 are used. These algorithms can be implemented in an obvious way by GT
Figure 4.3: The relation2SGEdge GT rule

and ASM rules by directly following the procedural pseudo-code that can be found in [20]. As a consequence, only the skeleton of the implementation is discussed here.

```
rule searchPlanCalc(in SG) = seq {
    choose BeginNode with find(selectBeginNode(BeginNode,SG)) do
        // Executes Chu-Liu / Edmonds algorithm on SG with
        // BeginNode as a center node
        call chualg(BeginNode,SG);

    // Runs the simple greedy algorithm
    sPlan(SG);
}
```

GT pattern `selectBeginNode` selects the center (begin) node of the search graph.

- **Chu-Liu and Edmonds algorithm.** The main concept of the implementation is that if a depth search algorithm starting from the center node of the search graph, that traverses only on the edges selected from $S$, does not visit all the nodes of the search graph then unvisited nodes contains a cycle, and the pseudo node can be easily selected from the unvisited nodes.

The extract of the commented VTCL code that realises the Chu-Liu and Edmonds algorithm is as follows:

```
rule chualg(in BeginNode,in SG) = seq {
    //STEP 1&2 Select begin node and for all other nodes select the incoming node
    // with the smallest weight
    forall No below SG with find(findNode(No)) do seq
        { //building up the beginning S list
            update values("Min") = "infinite";
            update nodes("MinEdge") = No;
```
if(value(No)=="fix")
    choose Bsz1 below SG, Msz1 below SG, En below SG with
    find(nextnode(BeginNode,No,Bsz1,Msz1,En))
    do update nodes("MinEdge")= En;
else
    forall En below SG,We below SG with find(findIncEdgeWeight(No,En,We))
    do seq
        {
            if(smaller(value(We),add(values("Min"),"0"))=="true") seq
                {
                    update nodes("MinEdge") = En;
                    update values("Min") = value(We);
                };
        
    choose Sln below SG, Tln below SG with
    find(getLabels(nodes("MinEdge"),Sln,Tln)) do
        setValue(Sln,"S");
    
    //Step 3 looking for circles
    call step3(BeginNode, SG);
    //have the minimum weight spanning tree

ASM rule **step3** is the most important part of the algorithm, when cycles are being searched in a recursive procedure. As the complicated details of the concrete cycle searching would highly decrease the perspicuity of this section, these details are not presented here.

**rule step3**(in Center,in SG) = seq {
    call countnodes(Center, SG, values("sizeofT"));
    if(values("sizeofT") != values("nodeMaxNumber")) seq
        {//there are circles in the "tree" search graph
            choose Bnode below SG with find(findBeginNode(Bnode)) do
                call findCircle(Center,Bnode, SG);
                call step3(Center, SG); //to check if there is another
        
    

- **Simple greedy algorithm.** The algorithm simply selects the smallest edge that goes out from the search graph nodes that are already in \( P \), and adds the target of the selected edge to \( P \). In the beginning, the center node and the bounded nodes are already in \( P \). The first **forall** selects the smallest edge that goes out from the \( P \), by using the ASM function **nodes** and **values** to store the actual smallest edge. Then the **choose** selects the target of the edge and adds it to \( P \) by setting the value of the node to \( P \). The recursion terminates when the counter of nodes in \( P \) reaches the number of the nodes in the search graph (stored in the ASM function **values**).

**rule sPlan**(in SG) = seq
CHAPTER 4. AUTOMATED CODE GENERATION

{ update values("Min") = "infinite";
  update nodes("MinEdge") = "1";
  forall No below SG, NextNode below SG, Edge below SG, Owe below SG
  with find(searchPlan(No,NextNode,Edge,Owe)) do
    if((value(Owe) < values("Min") && value(NextNode)!="P") seq
      update values("Min") = value(Owe);
      update nodes("MinEdge") = Edge;
    );
  choose No below SG, NextNode below SG, Owe below SG
  with find(searchPlan(No,NextNode,nodes("MinEdge"))) do seq
    {update Bracket="false";
      setValue(NextNode, "P");
      update values("searchPlan") = values("searchPlan")+1;
    }
  if(values("searchPlan") != values("nodeMaxNumber"))
    call sPlan(SG);
}

Code generation. Finally, the search graph is traversed by the newly constructed search plan, and the output Java code is generated according to the mappings discussed in Sec. 3.3.6 An iterate rule combined with a choose rule iterates over the edges and nodes of the search plan, and calls the generateMatch rule, which mainly consists of print rules in order to generate the corresponding code segments. The corresponding VTCL code is as follows:

rule searchPlan2Code(in SG) = seq {
  update number("sPlan")=1 //ASM function
  iterate
    choose Targ below SG, Sour below SG, Con below SG
    with find(searchPlanN(number("sPlan"),Targ,Sour,Con)) do seq {
      generateMatch(Targ,Sour,Con);
      update number("sPlan") = number("sPlan")+1;
    }
}

The ASM function number is used to store the actual position in the traversal order. The GT pattern searchPlanN is used to get the actual edge of the traversal order according to the position number. Con is the actual edge in the traversal order, Sour and Targ are the source and target of the edge, respectively. As soon as the value of sPlan exceeds the number of nodes in the search graph, the choose rule fails finding a matching pattern, and the iterate rule terminates.
Chapter 5

Performance assessment

This section focuses on the runtime performance assessment of the generated platform specific (i.e., Java) model transformer.

Based on my initial experiments with the interpreted PIT of the Viatra2 framework, the pattern matching phase has the most significant effect on the performance. As similar experiments can be expected for the generated platform specific transformer, I performed measurements on the graph transformation engine being generated as a part of the PST.

By using the terminology and the running example of [34], I selected the object-relational mapping as a benchmark example also for the current measurements, which can be considered as an incarnation of a typical model transformation scenario. In order to fix a test set, which is a complete, deterministic, but parametric specification, the structure of the initial model and the transformation sequence have been fixed up to numerical parameters. In our case, the number of Classes in the initial instance model (denoted by \( N \)) is selected as the single numerical parameter.

![Initial model of the test case for the N=3 case](image)

Figure 5.1: Initial model of the test case for the N=3 case

The structure of the initial model is presented in Fig. 5.1 for the \( N = 3 \) case. The model has a single Package that contains \( N \) classes. An Association and two AssociationEnds are added to the model for each pair of Classes, thus initially, we have \( N(N-1)/2 \) Associations and \( N(N-1) \) AssociationEnds. Associations are also contained by the single Package as expressed by the corresponding links of type EO. Each AssociationEnd is connected to a corresponding Association and Class by a CF and SFT link, respectively.

The transformation sequence consists of four macro steps that are executed in this spe-
specific order. The first macro step is a single application of the SchemaR (Fig. 2.3(a)). This is followed by a macro step that consists of \(N\) applications of rule ClassR (Fig. 2.3(c)). Then Classes are transformed by the execution of rule AssociationR (Fig. 2.3(d)) for \(N(N - 1)/2\) times. Finally, a macro step of length \(N(N - 1)\) follows, which prescribes the application of rule AssocEndR (Fig. 2.3(e)).

This test set can be characterized by large patterns and a large number of possible matchings for a rule. The maximum degree of nodes (fan-out) and the length of the transformation sequence depend on parameter \(N\).

According to the earlier analysis reported in [34], the most significant speed-up could be observed when parallel rule execution is used as an optimization strategy. In case of parallel rule execution, all matchings of a rule are calculated in the pattern matching phase, and then updates are performed as a transaction block on the collected matchings without re-evaluating valid matchings during the transaction. In our case, parallel execution of GT rules is expressed by forall ASM constructs, thus, the VTCL code of parallel and sequential rule execution is presented in Fig. 5.2.

```vtcl
//sequential execution
apply schemaR(Pack);
itrate choose Class with find(classR.lhsR(Pack,Class))
do apply ClassR(Pack,Class);
itrate choose Assoc with find(classR.lhsA(Pack,Assoc))
do apply AssocR(Pack,Assoc);
itrate choose AssocEnd with find(classR.lhsAE(Pack,AssocEnd))
do apply AssocEndR(Pack,AssocEnd);

//Parallel execution mode
apply schemaR(Pack);
forall Class with do apply ClassR(Pack,Class)
forall Assoc with do apply AssocR(Pack,Assoc)
forall AssocEnd with do apply AssocEndR(Pack,AssocEnd)
```

Figure 5.2: Sequential and Parallel execution

Another characteristic tool feature of the generated platform specific model transformer is the multiplicity based optimization, when the transformer employs a more sophisticated strategy in order to find matching model elements for an edge with bounded multiplicity. In our case, unoptimized version handles all relationships in preconditions of GT rules as VPM relations when weights of the search graph are calculated, while the optimized version differentiates between VPM functions and relations by preferring VPM functions over VPM relations during the search space traversal (SST) as presented in Sec. 4.2.2.

As two orthogonal features have been identified, I performed my measurements on all the four possible combinations of these features, which means that four test cases have been analyzed. The parameter \(N\) was fixed to 10, 50, 100 and 250 in all test cases.

Our measurements were performed on a 2200 MHz AMD machine with 1024 MB RAM. A Windows XP build SP2 served as an underlying operating system. The execution time results are shown in Table 5.1.
Table 5.1: Experimental results

<table>
<thead>
<tr>
<th>Class</th>
<th>Model size</th>
<th>TS length</th>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>match</td>
<td>update</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>#</td>
<td>m sec</td>
<td>m sec</td>
</tr>
<tr>
<td>AssocEndRule</td>
<td>10</td>
<td>1342</td>
<td>148</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>34702</td>
<td>3726</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>139402</td>
<td>14951</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>872192</td>
<td>90626</td>
<td>1.03</td>
</tr>
<tr>
<td>AssocEndRule</td>
<td>10</td>
<td>1342</td>
<td>148</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>34702</td>
<td>3726</td>
<td>2.29</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>139402</td>
<td>14951</td>
<td>8.57</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>872192</td>
<td>90626</td>
<td>21.71</td>
</tr>
<tr>
<td>AssocEndRule</td>
<td>10</td>
<td>1342</td>
<td>148</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>34702</td>
<td>3726</td>
<td>4.88</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>139402</td>
<td>14951</td>
<td>26.10</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>872192</td>
<td>90626</td>
<td>217.64</td>
</tr>
</tbody>
</table>

The head of a row (i.e., the first column) shows the name of the rule. The second column (Class) depicts the number of classes in the run, which is, in turn, the runtime parameter $N$ for the test case. The third and fourth columns show the concrete values for the model size and the transformation sequence length, respectively. Heads of the remaining columns unambiguously identify the optimization strategy settings (i.e., status of parallel rule execution and multiplicity based optimization) on which the average time of execution is calculated. (Note that a rule is executed several times in a run.) Values in match and update columns depict the average times needed for a single execution of a rule in the pattern matching and updating phase, respectively. Execution times were measured on a microsecond scale, but a millisecond scale is used in Table 5.1 for presentation purposes.

Our initial experiments can be summarized as follows.

- In accordance with our assumptions, parallel rule execution has a significant effect on pattern matching, especially in larger model space.

- The update phase is slightly different in the sequential execution, but in parallel the execution time is dramatically increasing proportional to the size of the model. This increase is caused by the Collection used in the forall construct to store all the successful matching of a GT pattern. In future work this kind of mapping of the forall ASM construct has to be changed in order to optimize the update section execution.

- The advantage of the multiplicity based optimization can be observed when processing complex GT patterns, like AssocEndRule.
Chapter 6

Case Study: Integrating graph transformation into EMF

This chapter demonstrates how to integrate the developed PST generator plugin into Eclipse Modeling Framework (EMF [1]), giving the ability to use graph transformations directly on EMF models.

6.1 EMF Model Integration

6.1.1 Eclipse Modeling Framework

Eclipse Modeling Framework is a Java framework and code generation facility for building tools and other applications based on a structured model. EMF provides a meta-model (Ecore) for describing structured models. Using these structured models EMF provides tools and runtime support to produce a set of Java classes representing the model in Java Virtual Machine (JVM), a set of adapter classes that enable viewing and a basic editor. As EMF already supports a large set of modeling constructs my discussion mainly focuses on the Ecore meta- and instance models and follows [15].

Ecore, which is essentially the class diagram subset of UML, is based on the Object Management Group’s (OMG) Meta Object Facility (MOF) specification. A simplified Ecore model is depicted in Fig. 6.1 in order to demonstrate the most important elements of the meta-model.

- **EClass** models classes themselves. Classes are identified by their name and can contain a number of attributes and references. To support inheritance, a class can refer to a number of other classes as its supertypes.

- **EAttribute** models attributes, the components of an object’s data. They are identified by their name, and they have a type.

- **EDataType** models the types of attributes, representing object data types that are defined in Java, but not in EMF. Data types are also identified by their name.
**ERefERENCE** is used in modeling associations between classes; it models one end of such an association. Like attributes, references are identified by their name and have a type.

![Ecore kernel](image)

**Figure 6.1: The Ecore kernel**

**Representing an Ecore model.** The basic form of an EMF model is an XML Metadata Interchange (XMI) serialization of an Ecore model. However, one of the main strengths of EMF is its flexibility with respect to the means by which the Ecore model can be defined:

- an (Ecore) **XMI document** can be created directly, using an XML or text editor, or using EMF’s simple tree-based sample Ecore editor,
- an Ecore model can be created using a commercial **UML modeling tool** such as Rational Rose,
- basic **Java interfaces with annotations** (@generated) can be used to describe an Ecore model,
- an **XML Schema** defining the data structures for the model, can be converted to an Ecore model,
- **EMF is a highly extensible and customizable framework/toolkit. Other forms** of model definition are also supported.

The first approach is the most direct, but generally appeals only to users expert in XML. The second choice is the most desirable if the user is already familiar with UML and Model Driven Software Development, while the Java approach provides the benefits of modeling in a pure Java development environment like Java Development Tool (JDT) [28]. The XML Schema approach is most desirable when the application is intended to manipulate XML data that is already defined using an XML Schema (as with web services). Regardless of which input form is used to define an EMF model, the benefits are the same, and almost all type can be generated directly from the Ecore model.
Extension of the Ecore. From an Ecore model, the generator of EMF can create a corresponding set of Java implementation classes. Every generated EMF class extends from the framework base class, EObject, which enables the objects to be integrated and appear in the EMF runtime environment. EObject provides an efficient reflective API for accessing the properties of the object generically. In addition, change notification is an essential property of every EObject and an adapter framework can be used to support extension to the objects. As one of its main advantage EMF provides support for dynamic models; that is, Ecore models that are created (in memory), and then instantiated without generating code. All modeled objects implement EObject in order to provide these features:

- The reflective API in EMF enables to manipulate all attributes and references attached to the EObject by using the eSet and eGet functions. This is conceptually equivalent to java.lang.reflect.Method.invoke() Java method, though it is much more efficient in the aspect of performance.

- Notification observers (or listeners) in EMF are called adapters because in addition to their observer status, they are often used to extend the behavior (that is, support additional interfaces without subclassing) of the object they are attached to. An Adapter, as a simple observer, can be attached to any EObject by simply adding the adapter to the eAdapters list of the EObject. This Adapter implements a function called notifyChanged, which is called any time when the EObject which contains the Adapter is manipulated. All information about the manipulation is held by a notification object which is the input parameter of the notifyChanged function. A sample adapter EMFInstanceManager, can be found in Appendix B.3. This adapter is responsible for sending the notifications to the EMFInstanceAdapter manager class.

- EMF provides support for creating both meta- and instance level dynamic models. Combining the dynamic abilities of the framework and the reflective API gives an opportunity to extend the meta-model in runtime.

6.1.2 Listeners in Viatra2

Viatra2 also includes a notification framework similar to the EMF concept mentioned in Sec. 6.1.1. The main differences are as follows:

- Adapters in Viatra2 are called Listeners and they cannot extend the behavior of the model elements,

- notifications are sent by the model space and the listeners are connected to the notification manager of the model space instead of each model element as in EMF, as shown in Fig. 6.2.

- the Notification object holds all information about the notified event. In EMF for all types of events the same notification object is used, while in Viatra2 for each manipulation type a different objects are sent, which have a common super type.
CHAPTER 6. CASE STUDY

A sample listener `SetNameListener` is presented in Appendix B.3. It is responsible for handling name manipulations. The listener checks the notification and if it is of type (ACTION_SET_NAME) then it sends the information towards the `ViatraIntegrationManager`.

6.2 Connecting Viatra2 to EMF

The integration of graph transformations to EMF can be divided into two main parts. First, the EMF models have to be imported and synchronized with the Viatra2 framework. This synchronization is handled by the notification mechanisms of the two frameworks discussed in Sec. 6.2.1. As for the second part the PIT to PST transformation has to be modified in order to handle the EMF based PSM, discussed in Sec. 6.3.

6.2.1 Overview of the Integration

As both modeling methodologies are similar, a dynamic synchronization is possible between the two models. To avoid the upcoming problems from the difference currently only a subset of the Ecore models can be used in the generated graph transformations. The restrictions and assumptions are the following:

- Only instance level EMF models are allowed, as notifications for Ecore meta-models in EMF version 2.1.2 are not completely implemented.

- It is assumed that the meta-model of the EMF model is already available in the Viatra2 model space. Model importation\(^1\) can be made from the Ecore xmi file mentioned in Sec. 6.1.1.

- Only in the Sec. 6.1.1 mentioned Ecore elements: EClass, EAttribute, EReference and EDataType are allowed.

**Connecting EMF models to Viatra2.** An overview of the integration is provided in Fig. 6.2

The upper and lower parts represent constructs and activities of the Viatra2 and of the EMF framework, respectively.

- The `ViatraIntegrationManager` is responsible for the synchronization of the Viatra2 model to the EMF model, while the `EMFInstanceManager` handles the synchronization the other direction.

- Each Viatra2 IElement holds a reference to its corresponding EMF EObject. These references are omitted to keep the Fig. 6.2.

- For each EMF EObject a corresponding InstanceAdapter is attached. Each adapter holds the id of the corresponding Viatra2 IElement, by which the `EMFInstanceManager` class can manipulate the element.

\(^1\)The importer plugin for Ecore is currently under development by another student.
Only a single `InstanceListener` is connected to the VIATRA2 framework. This listener determines the type of the notification and calls the corresponding method of the VIATTEGRATINManager class, which synchronize the EMF model using the reference of the source `IElement`.

6.2.2 Notification handling

As events have finer granularity in VIATRA2 than in EMF, events sent by VIATRA2 do not have a direct equivalent in EMF. For example in VIATRA2 a relation is instantiated in two steps: first the relation is created between the source and target element, and then a type is assigned to the relation, but in EMF this can be done in only one step. To overcome this problem a simple one step delay is used in case of the `InstanceListener`. This means that each notification for a certain `IElement` is delayed and processed only after a second one is sent by the model space. With this kind of one step look up, all notified events can be mapped to EMF model manipulation commands. Notifications are stored in the VIATRALINTEGRATIONManager by a HashMap, with the id of the `IElement` as the key.

The VIATRALINTEGRATIONManager and the EMFInstanceManager have a similar functionality as they both synchronize by parsing the notifications given by the listener and the adapters, respectively. The only difference is that in case of the VIATRALINTEGRATIONManager the type of the notification is determined by the `InstanceListener` and the listener calls the corresponding method for each type, while in the EMFInstanceManager the `InstanceAdapters` sends only the notifications and the type is determined by the manager.

In case of the EMFInstanceManager a single switch-case construct is used, as the type of the notification is returned by the `getEventType` method as an integer. The `Notification` class holds the static value for each type. The following example represents a case branch, which is responsible for the notification type ADD. The
Feature attribute in the if construct determines whether the notification is sent by an EAttribute or an EReference and the addInstanceModelEReference or the addInstanceModelEAttribute method is invoked, respectively.

switch (notification.getEventType()) {
  ...
  case Notification.ADD: //EMF ADD notification object
    if (not.getFeature() instanceof EAttribute)
      addInstanceModelEReference((EObject)notification.getNotifier(),
        notification);
    else
      addInstanceModelEAttribute((EObject)notification.getNotifier(),
        notification);
    break;
  // ... other case branches
}

For example, if the addInstanceModelEReference method is called, then the target EObject is returned by the getNewValue function and the createoraddReference method is called with three parameters: the source and target of the EReference, and the EReference, which is returned by the getFeature method. The createoraddReference returns the target EObject with a newly connected Instance-Adapter and creates the corresponding VPM elements, if the target element did not have an InstanceAdapter, which means that the element has not been instantiated in the Viatra2 framework yet. In all other cases it returns a null. If a non null value is returned, the parseEClass method continues the synchronization with the contained elements of the viatrareferenced EObject.

protected void addInstanceModelEReference(EObject source, Notification not) {
  //gets the target of the reference
  EObject target = (EObject)not.getNewValue();
  //check if it is a new element or an already synchronized
  //returns the target element with the Adapter connected or null
  EObject viatrareferenced = createoraddReference(source, target,
    (ERefERENCE)not.getFeature());
  // processes the EObject further
  // if it is a new element in the modelspace
  if (viatrareferenced != null)
    mp.parseEClass(viatrareferenced);
}

• In the InstanceListener the type of notifications are determined by if constructs and the instanceof operator. The following example represents the corresponding code segment responsible for the relation creation notification. It calls the synchronizeRelation method of the ViatraIntegrationManager with a Notification casted to ICoreNotificationObjectCreateRelation.

if (Notification instanceof ICoreNotificationObjectCreateRelation) {
  vManager.synchronizeRelation(}
The `synchronizeRelation` method simply saves (delays) `ICoreNotificationObjectCreateRelation` with the id of the IRelation as a key in the `delay` HashMap. After the save each time an `ICoreNotificationObjectCreateInstanceOf` is sent the `synchronizeCreateInstanceOf` method is called and it checks that the new instance can be found in the `delay` HashMap. A `VPMCoreException` is raised if it is not found. In the other case the type of the notification is selected by the `instanceof` operator, and the corresponding EMF elements are generated. In this example the parameters of the IRelation are retrieved from the notifications and the `createEMFEReference` method is called.

```java
protected synchronizeRelation(
    ICoreNotificationObjectCreateRelation not) { // it only saves the notification
    // and the creation is made when the type is determined
    delay.put((not.getNewRelation().getID(), not)); }

... 

protected synchronizeCreateInstanceOf(ICoreNotificationObjectCreateInstanceOf not) throws VPMCoreException { // check if it has no type currently
    throws error if it is not in the list
    if(!delay.containsKey(not.getInstance().getID())) throw new VPMCoreException(VPMCoreException.NoInstance);
    Notification notification = delay.get(not.getInstance().getID()); ...
    if(notification instanceof ICoreNotificationObjectCreateRelation)
    {
        ICoreNotificationObjectCreateRelation oldnot = (ICoreNotificationObjectCreateRelation) notification
        IElement source = oldnot.getFrom();
        IElement target = oldnot.getTo(); // get the values
        IRelation rel = not.getInstance(); // of the relation
        IElement type = not.getType();
        // synchronize with the EMF model space
        createEMFEReference(source, target, rel, type);
        // removes from the list
        delay.remove(not.getInstance().getID());
    }
}
```

## 6.3 Graph Transformation in EMF

At this point, the models in the two frameworks are synchronized. Now we present how the `EObject` based PST can be generated. Note that, the PSM is now based on the generated EMF Java classes opposed to the VPM based code generation discussed in Sec. 3.1. As EMF is has a different philosophy as VPM, a few features are infeasible:
• The value attribute, which is part of the VPM entity.

• The in, below type pattern matching constraints.

6.3.1 EMF based PSM

Each object in the EMF extends the EObject class, which gives a huge advantage as it supports management and reflective methods as discussed in Sec. 6.1.1. A detailed discussion about the EMF code generation can be found in [15]. For this reason this section focuses on the differences from the mapping already discussed in Sec 3.1.

Generated EMF code.

• The name attribute is handled outside the EMF domain, as it is not obligatory in EMF, but obligatory in Viatra2.

• An EMF object has always a primary supertype (in our example ClassImpl), from which the class extends, while the other super types are generalized by interfaces.

• The AssocImpl class does not implement the EntityInterface, as it is responsible only for managing Value, Below attributes.

• Different relation managing functions are generated by EMF for relations with 0..1 multiplicity (AssPck) and arbitrary multiplicity (AssAs).

• The instances of the class are managed similar with the Instances vector.

• Each class holds a reference domain to its containing modeling domain.

An extract of an EMF based Java class (for the complete program see Appendix A.2) being generated for the Assoc class (See in Example 9):

```java
public class AssocImpl extends ClassImpl implements AssocInt{
public static ResourceSet domain;
public String Name=null;
public static Vector Instances=new Vector();
public Assoc(){Instances.add(this);}
//NameInterface functions
public String getName(){return Name;}
public void reName(String s){Name = s;}
public Collection getInstances(){return Instances;}
public void delete(){...}
/**
 * The cached value of the '{@link #getAssAse()
 * <em>Ass Ase</em>}' containment reference list.
 * @generated
 */
//GENERATED by EMF
/**
 * The cached value of the '{@link #getAssAs()
 * <em>Ass As</em>}' containment reference list.
 * @generated
 */
```
protected EList assAse = null;
protected AssocImpl() {...}
protected EClass eStaticClass() {...}
public PackageImpl getAssPck() {...}
public void setAssPck(PackageImpl newAssPck) {...}
public EList getAssAse() {...}
public NotificationChain eInverseAdd(InternalEObject otherEnd,
int featureID, Class baseClass, NotificationChain msgs) {...}
public NotificationChain eInverseRemove(InternalEObject otherEnd,
int featureID, Class baseClass, NotificationChain msgs) {...}
public NotificationChain eBasicRemoveFromContainer(NotificationChain msgs) {...}
public Object eGet(EStructuralFeature eFeature, boolean resolve) {...}
public void eSet(EStructuralFeature eFeature, Object newValue) {...}
public void eUnset(EStructuralFeature eFeature) {...}
public boolean eIsSet(EStructuralFeature eFeature) {...}
}

//AssocImpl

**EMF representation of Auxiliary classes** As the manipulation methods for the relations and attributes are changed, the auxiliary classes have to be changed too. The SourceHolder is no longer needed, because the EObject can handle cross references. The FuncHolder is only a slightly modified, the type of the Target attribute is changed to EObject, while the ConHolder class is modified in order to use the reflective API of the EObject. The modified ConHolder class is as follows:

```java
public class ConHolder implements NameInterface {
    String field;
    EObject Source, Target;
    public ConHolder(String Fi, EObject T, EObject S) {
        field = Fi; //Field of the function/relation(Java)
        Source = S; Target = T;
    }

    public void delete(){
        try {
            //get the List of the relation and removes the element
            ((List)company.eGet((company.eClass()).getEStructuralFeature(field))).
                remove(department);
        } catch (Exception e) {}    
    }
}
```

**6.3.2 Modifications in the generated PST**

As an effect of the modifications and restrictions, the generated PST must be modified only in the EObject specific parts: (i) model manipulation rules, (ii) pattern matching.
**Element creation rules.** There are three types of model element creation rules. One is for creating a new entity, and the other two is to create EReferences and EAttributes. The difference is only in the EReference and EAttribute creation, as the methods have different input parameters. For the EAttributes and 0..1 multiplicity EReferences a `setXYZ` method is called with the source as its input parameter. In case of arbitrary multiplicity EReference, the container of the EReference is returned by the `getXYZ` method, and the new source is directly added to the container by the `add` method. $XYZ$ is the name of the EReference/EAttribute.

```java
// a new AssocImpl class is created and the variable Ass holds
// a reference to this class
Variables.put("Ass", new AssocImpl());
// new AssAse EReference Assoc and AssociationEnd variables
((AssocInt)Variables.get("Tid")).getAssAse()
  .add(Variables.get("AssociationEnd"));
// in case of 0..1 multiplicity EReference and EAttribute
((AssocInt)Variables.get("Tid"))
  .setAssPck((PackageImpl)Variables.get("Package"));
```

The same code segments are generated for the delete, reName and setName rules.

**Pattern matching.** The only modification made in the pattern matching code segments is that in case of arbitrary multiplicity EReferences the `getXYZ` returns a list instead of a HashMap, so the iterator can be achieved a differently.

The pattern matching code segment of the search plan of Fig 3.2 with no input parameters and choose type pattern call having $P$ and $C$ as output parameters are the following:

```java
public static HashMap lhs() throws RuleFailedException {
    HashMap result = new HashMap();
    boolean success = false;
    try{
        Iterator iter_P = Package.getInstances().iterator();
        while(iter_P.hasNext()){
            try{
                // Instances
                PackageInt P = (PackageInt) iter_P.next();
                result.put("P", P);
                try{
                    // 0..1 Multiplicity EReference
                    SchemaInt S = (SchemaInt) P.getPckSch();
                    // arbitrary multiplicity EReference
                    Iterator iter_C = P.getPckCls().iterator();
                    while(iter_C.hasNext()){
                        try{
                            ClassInt C = (ClassInt) iter_C.next();
                            result.put("C", C);
                            // NAC
                            if(p(C))
                                throw new ClassCastException();
                        }
                    }
                }
            }
        }
    }
    return result;
}
```

The same code segments are generated for the delete, reName and setName rules.
//check
success = true;
} catch (ClassCastException e) {} } } } catch (ClassCastException e) {} } } catch (ClassCastException e) {} } catch (ClassCastException e) {} } //
if(success)
    return result;
else
    throw new RuleFailedException();
}
Chapter 7
Conclusions

7.1 Related Work

While there is already a large set of model transformation tools available in the literature using graph rewriting, below I focus on providing a brief comparison with the four most popular and advanced compiled approaches that show conceptual similarities with my approach.

Fujaba [21] compiles visual specifications of transformations [14] – defined by the combination of graph transformation and UML activity graph – into executable Java code based on an optimization technique using search graphs with a breadth-first traversal strategy. Fujaba performs local search [16] starting from the node selected by the system designer and extending the matching step-by-step by neighbouring nodes and edges. Fujaba fixes a single, breadth-first traversal strategy at compile-time (i.e., when the pattern matching code is generated) for each rule. Fujaba uses simple rules of thumb for generating search plans. A simple rule is that navigation along an edge with an at most one multiplicity constraint precedes navigations along edges with arbitrary multiplicity. My solution was, influenced by Fujaba; however, provides a better solution for complex control structures using the derived Java code from the Viatra2 ASM instead of the activity graphs in the Fujaba.

PROGRES [27] supports both interpreted and compiled execution (generating C code) of programmed graph transformation systems. It uses a very sophisticated cost model for defining costs of basic operations (like enumeration of nodes of a type and navigation along edges). These costs are not domain-specific in the sense that they are based on assumptions about a typical problem domain on which the tool is intended to be used. Operation graphs of PROGRES, which are similar to search graphs in the current thesis, additionally support the handling of path expressions and attribute conditions. The compiled version of PROGRES generates search plan at compile-time by a greedy algorithm, which is based on the a priori costs of basic operations. Some of the solutions in the graph pattern matching used in PROGRES are planned to be used in the future.

The pattern matching engine of compiled GReAT [36] (generating C++ code) uses a breadth-first traversal strategy starting from a set of nodes that are initially matched.
This initial binding is referred to as pivoted pattern matching in GReAT terms. The GReAT engine provides efficient and optimized compiled code for pattern matching, but does not assist easy maintenance and reusability.

AGG [13] is currently the only pattern matching engine based on CSP. It supports only interpreted execution, and cannot match in performance with the other engines, but in many other fields it is very promising as a new breed of pattern matching approaches. Some of the techniques (backtracking algorithm, dynamic domain reduction) used in AGG are planned to be integrated in the future.

7.2 Result Assessment

In this final section, a short summary, evaluation and conclusion are presented together with a brief additional overview focusing on possible directions of future work.

7.2.1 Main Conclusion

As a summary, my results are expressed in the form of main conclusion statements.

Main Conclusion 1 I overviewed the main meta-modeling and model transformation concepts and notations of the ViATRA2 framework.

Main Conclusion 2 I specified an automated mapping of VPM-based meta-models and models to Java classes and objects.

Main Conclusion 3 I defined and implemented a module that automatically generates a Java specific model transformer from the platform independent model transformation description of the ViATRA2 framework.

Main Conclusion 4 I proposed a meta-transformation for the PIT-to-PST mapping.

Main Conclusion 5 I assessed the run-time performance of the graph transformation engine that have been generated as a part of the PST.

Main Conclusion 6 I proposed a technique for the integration of graph transformation capabilities into the Eclipse Modeling Framework.

7.2.2 Future work

My future research mainly focuses on five possible directions:

- The implementation of a platform-specific transformer based on EJB 3.0. The key feature of this approach is the robustness that, the underlying EJB3 application servers can offer for managing huge models with persistent entity beans stored in database, handling transformations as transactions, increase performance with clustering and the use of EJB-QL.
• An improvement in the optimization of graph pattern matching by (i) using more sophisticated algorithms for finding low cost search plans, (ii) designing more optimized mapping for the sequential and parallel execution using model-specific and adaptive search plans, and (iii) implementing a CSP based solution to improve the capability of the adaptive search plan generating algorithm.

• The integration of this module to the Viatra2 framework to provide an alternative, underlying model transformation engine for Viatra2.

• The execution of further measurements on the graph transformation engine of the generated platform specific transformer to assess the performance of my approach on other model transformation scenarios.

• The completion of the implementation of the proposed integration with EMF by improve the level of integration by using the EMF.Edit and EMF.Util frameworks in case of the model manipulations.

7.2.3 Acknowledgement

I would like to thank Dániel Varró who undertook the supervision of my self-directed studies. I am also very much grateful to Gergely Varró for reading the earlier versions of the thesis and suggesting valuable improvements, and I also very much appreciate the valuable comments of András Balogh. I would also like to thank Dániel Tóth who helped understanding the Eclipse Modeling framework.
Bibliography


[12] Elena Litani, E. M. and D. Steinberg, Discover the eclipse modeling framework (emf) and its dynamic capabilities, \url{http://www.devx.com/Java/Article/29093/0/page/1}


Appendix A

Generated code

A.1 PSM model of Assoc entity in Java

```java
public class Assoc
    implements ClassInterface, EntityInterface, NameInterface, AssocInterface
{
    public String Value=null;
    public String Name=null;
    public static Vector Instances=new Vector();
    public Vector Below=null;
    public Vector Sources=null;
    public Assoc()
    {
        Instances.add(this);
        Below = new Vector();
        Sources= new Vector();
    }
    public Assoc(String n, String v)
    {
        Name=n;
        Value=v;
        Instances.add(this);
        Below = new Vector();
        Sources= new Vector();
    }
    //Entity functions
    public String getName()
    {
        return Name;
    }
    public void rename(String s)
    {
        Name = s;
    }
    public String getValue()
    {
        return Value;
    }
    public void setValue(String s)
    {
        Value = s;
    }
    public static void putInstance(Object value)
    {
        if(Instances==null)
        {
            Instances = new Vector();
        }
        Instances.add(value);
    }
    public static Collection getInstances()
    {
        return Instances;
    }
    public void addBelowElement(Object o)
    {
        if(Below==null)
        {
            Below = new Vector();
        }
        Below.add(o);
    }
    public void deleteBelowElement(Object o){Below.remove(o);} 
    public Collection getIn()
    {
        return Below;
    }
    public Collection getBelow()
    {
        if(Below==null) return null;
    }
}
```

73
Vector r = new Vector();
for(Iterator i = Below.iterator(); i.hasNext();)
{
    Object O = i.next();
    if(!(O instanceof String))
    {
        Collection temp = ((EntityInterface) O).getBelow();
        if(temp!=null) r.addAll(temp);
    }
    else
    {
        r.add((String)O);
    }
}
    r.addAll(Below);
return r;

public void deleteSource(String field,String name, Object o)
{
    Iterator iter = Sources.iterator();
    while(iter.hasNext())
    {
        SourceHolder h = (SourceHolder) iter.next();
        if(h.name.equals(name) && h.hold.equals(o) && h.field.equals(field))
        {
            Sources.remove(h);
        }
    }
}

public void addSource(String field,String name, Object o)
{
    Sources.add(new SourceHolder(field,name, o));
}

public void delete()
{
    Instances.remove(this);
    Iterator iter = Below.iterator();
    //all below element
    while(iter.hasNext())
    {
        ((NameInterface)iter.next()).delete();
    }
    //where entity is the target
    Iterator iterS = Sources.iterator();
    while(iter.hasNext())
    {
        SourceHolder h= (SourceHolder) iterS.next();
        try
        {
            Method m= h.hold.getClass().
            .getMethod(h.field,new Class[]{String.class});
            m.invoke(h.hold,new Object[]{});
        } catch (Exception e) {}  
    }
    //entity is the source
    deleteAssPckall();
    deleteAssAseall();
    deleteClsPckall();
}

//FUNCTIONS
private FunctionHolder AssPck=new FunctionHolder();
public Object getAssPck() { return AssPck.get();}
public void renameAssPck(String old , String s) {
    ((EntityInterface)AssPck.get())
      .deleteSource("deleteAssPck",AssPck.getName(),AssPck.get());
    AssPck.setName(s);
    ((EntityInterface)AssPck.get())
      .addSource("deleteAssPck",s,AssPck.get());
}
public void setAssPck(String name, PackageInterface a) {
    AssPck.set(name, a);
    ((EntityInterface) AssPck.get()).addSource("deleteAssPck", name, a);
}

public void deleteAssPck(String a) {
    ((EntityInterface) AssPck.get()).deleteSource("deleteAssPck", AssPck.getName(), AssPck.get());
    AssPck = new FunctionHolder();
}

public void deleteAssPckall() { deleteAssPck(""); }

//RELATIONS
private HashMap AssAse = new HashMap();

public HashMap getAssAse() {
    return AssAse;
}

public void setAssAse(String name, AssocEndInterface a) {
    if (AssAse == null) {
        AssAse = new HashMap();
    }
    AssAse.put(name, a);
    ((EntityInterface) AssPck.get()).addSource("deleteAssAse", name, a);
}

public void deleteAssAse(String a) {
    if (AssAse == null) {
        return;
    } else {
        Object Temp = AssAse.get(oldName);
        AssAse.remove(a);
        ((EntityInterface) Temp).deleteSource("deleteAssAse", a, Temp);
    }
}

public void renameAssAse(String oldName, String newName) {
    Object Temp = AssAse.get(oldName);
    ((EntityInterface) Temp).deleteSource("deleteAssAse", oldName, Temp);
    AssAse.remove(oldName);
    AssAse.put(newName, Temp);
    ((EntityInterface) Temp).addSource("deleteAssAse", newName, Temp);
    Temp = null;
}

public void deletAssAseall() {
    Iterator iter = AssAse.entrySet().iterator();
    while (iter.hasNext()) {
        Map.Entry m = (Map.Entry) iter.next();
        ((EntityInterface) m.getValue()).deleteSource((String) m.getKey(),
            "deleteAssAse", m.getValue());
    }
    AssAse.clear();
}

//SuperTypes relations & functions

//FUNCTIONS OF ClassInterface
//The ClsPck function of the Assoc entity
private FunctionHolder ClsPck = new FunctionHolder();

public Object getClsPck() { return ClsPck.get(); }

public void renameClsPck(String old, String s) {
    ((EntityInterface) ClsPck.get())
}
APPENDIX A. GENERATED CODE

```java
public class AssocImpl extends ClassImpl implements AssocInt, EntityInterface{
    public static ResourceSet domain;
    public String Name=null;
    public static Vector Instances=new Vector();

    public Assoc(){Instances.add(this);}  
    
    //Entity functions
    public String getName(){return Name;}  
    public void reName(String s){Name = s;}

    public Collection getInstances(){return Instances;}
    //deletes the element from the model space
    //and cleans the cross references
    public void delete(){
        Instances.remove(this);
        
        Map usages = EcoreUtil.UsageCrossReferencer.findAll(this, domain.getResourceSet());
        for (Iterator i = usages.entrySet().iterator(); i.hasNext(); )
        {
            Map.Entry entry = (Map.Entry)i.next();
            EObject eObject = (EObject)entry.getKey();
            Collection settings = (Collection)entry.getValue();
            for (Iterator j = settings.iterator(); j.hasNext(); )
            {
                EStructuralFeature.Setting setting =
                (EStructuralFeature.Setting)j.next();
                EObject referencingEObject = setting.getEObject();
                EStructuralFeature eStructuralFeature =
                setting.getEStructuralFeature();
                if (eStructuralFeature.isMany())
                {
```
```java
appendAndExecute(RemoveCommand.create(domain, referencingEObject,
    eStructuralFeature, eObject));
}
else
{
    appendAndExecute(SetCommand.create(domain, referencingEObject,
        eStructuralFeature, null));
}
}

//GENERATED by EMF
/**
 * The cached value of the '{@link #getAssAse() <em>Ass Ase</em>}'
 * containment reference list.
 * @see #getAssAse()
 * @generated
 * @ordered
 */
protected EList assAse = null;
/**
 * @generated
 */
protected AssocImpl() {
    super();
}
/**
 * @generated
 */
protected EClass eStaticClass() {
    return UML_PackagePackage.eINSTANCE.getAssoc();
}
/**
 * @generated
 */
public PackageImpl getAssPck() {
    if (eContainerFeatureID != UML_PackagePackage.ASSOC__ASS_PCK) return null;
    return (PackageImpl)eContainer;
}
/**
 * @generated
 */
public void setAssPck(PackageImpl newAssPck) {
    if (newAssPck != eContainer || (eContainerFeatureID !=
        UML_PackagePackage.ASSOC__ASS_PCK
    && newAssPck != null)) {
        if (EcoreUtil.isAncestor(this, newAssPck))
            throw new IllegalArgumentException(
                "Recursive containment not allowed for "
                + toString());
    NotificationChain msgs = null;
    if (eContainer != null)
```
msgs = eBasicRemoveFromContainer(msgs);
if (newAssPck != null)
    msgs = ((InternalEObject)newAssPck).eInverseAdd(this,
            UML_PackagePackage.PACKAGE__PCK_ASS,
            PackageImpl.class, msgs);
msgs = eBasicSetContainer((InternalEObject)newAssPck,
            UML_PackagePackage.ASSOC__ASS_PCK, msgs);
if (msgs != null) msgs.dispatch();
else if (eNotificationRequired())
    eNotify(new ENotificationImpl(this, Notification.SET,
            UML_PackagePackage.ASSOC__ASS_PCK,
            newAssPck, newAssPck));
/**
 * @generated
 */
public EList getAssAse() {
    if (assAse == null)
        assAse = new EObjectContainmentWithInverseEList(AssocEndImpl.class, this,
                UML_PackagePackage.ASSOC__ASS_ASE, UML_PackagePackage.ASSOC_END__ASE_ASS);
    return assAse;
}
/**
 * @generated
 */
public NotificationChain eInverseAdd(InternalEObject otherEnd, int featureID,
        Class baseClass, NotificationChain msgs) {
    if (featureID >= 0) {
        switch (eDerivedStructuralFeatureID(featureID, baseClass)) {
        case UML_PackagePackage.ASSOC__CLS_PCK:
            if (eContainer != null)
                msgs = eBasicRemoveFromContainer(msgs);
            return eBasicSetContainer(otherEnd,
                    UML_PackagePackage.ASSOC__CLS_PCK, msgs);
        case UML_PackagePackage.ASSOC__ASS_PCK:
            if (eContainer != null)
                msgs = eBasicRemoveFromContainer(msgs);
            return eBasicSetContainer(otherEnd,
                    UML_PackagePackage.ASSOC__ASS_PCK, msgs);
        case UML_PackagePackage.ASSOC__ASS_ASE:
            return ((InternalEList)getAssAse()).basicAdd(otherEnd, msgs);
        default:
            return eDynamicInverseAdd(otherEnd, featureID, baseClass, msgs);
        }
    }
    if (eContainer != null)
        msgs = eBasicRemoveFromContainer(msgs);
    return eBasicSetContainer(otherEnd, featureID, baseClass, msgs);
}
/**
 * @generated
 */
public NotificationChain eInverseRemove(InternalEObject otherEnd, int featureID, Class baseClass, NotificationChain msgs) {
    if (featureID >= 0) {
        switch (eDerivedStructuralFeatureID(featureID, baseClass)) {
        case UML_PackagePackage.ASSOC__CLS_PCK:
            return eBasicSetContainer(null, UML_PackagePackage.ASSOC__CLS_PCK, msgs);
        case UML_PackagePackage.ASSOC__ASS_PCK:
            return eBasicSetContainer(null, UML_PackagePackage.ASSOC__ASS_PCK, msgs);
        case UML_PackagePackage.ASSOC__ASS_ASE:
            return ((InternalEList)getAssAse()).basicRemove(otherEnd, msgs);
        default:
            return eDynamicInverseRemove(otherEnd, featureID, baseClass, msgs);
        }
    }
    return eBasicSetContainer(null, featureID, msgs);
}
/**
 * @generated
 */
public NotificationChain eBasicRemoveFromContainer(NotificationChain msgs) {
    if (eContainerFeatureID >= 0) {
        switch (eContainerFeatureID) {
        case UML_PackagePackage.ASSOC__CLS_PCK:
            return eContainer.eInverseRemove(this, UML_PackagePackage.PACKAGE__PCK_CLS, PackageImpl.class, msgs);
        case UML_PackagePackage.ASSOC__ASS_PCK:
            return eContainer.eInverseRemove(this, UML_PackagePackage.PACKAGE__PCK_ASS, PackageImpl.class, msgs);
        default:
            return eDynamicBasicRemoveFromContainer(msgs);
        }
    }
    return eContainer.eInverseRemove(this, EOPPOSITE_FEATURE_BASE - eContainerFeatureID, null, msgs);
}
/**
 * @generated
 */
public Object eGet(EStructuralFeature eFeature, boolean resolve) {
    switch (eDerivedStructuralFeatureID(eFeature)) {
    case UML_PackagePackage.ASSOC__CLS_PCK:
        return getClsPck();
    case UML_PackagePackage.ASSOC__ASS_PCK:
        return getAssPck();
    case UML_PackagePackage.ASSOC__ASS_ASE:
        return getAssAse();
    }
    return eDynamicGet(eFeature, resolve);
}
```java
/**
   * @generated
   */
public void eSet(EStructuralFeature eFeature, Object newValue) {
    switch (eDerivedStructuralFeatureID(eFeature)) {
        case UML_PackagePackage.ASSOC__CLS_PCK:
            setClsPck((PackageImpl)newValue);
            return;
        case UML_PackagePackage.ASSOC__ASS_PCK:
            setAssPck((PackageImpl)newValue);
            return;
        case UML_PackagePackage.ASSOC__ASS_ASE:
            getAssAse().clear();
            getAssAse().addAll((Collection)newValue);
            return;
    }
    eDynamicSet(eFeature, newValue);
}

/**
   * @generated
   */
public void eUnset(EStructuralFeature eFeature) {
    switch (eDerivedStructuralFeatureID(eFeature)) {
        case UML_PackagePackage.ASSOC__CLS_PCK:
            setClsPck((PackageImpl)null);
            return;
        case UML_PackagePackage.ASSOC__ASS_PCK:
            setAssPck((PackageImpl)null);
            return;
        case UML_PackagePackage.ASSOC__ASS_ASE:
            getAssAse().clear();
            return;
    }
    eDynamicUnset(eFeature);
}

/**
   * @generated
   */
public boolean eIsSet(EStructuralFeature eFeature) {
    switch (eDerivedStructuralFeatureID(eFeature)) {
        case UML_PackagePackage.ASSOC__CLS_PCK:
            return getClsPck() != null;
        case UML_PackagePackage.ASSOC__ASS_PCK:
            return getAssPck() != null;
        case UML_PackagePackage.ASSOC__ASS_ASE:
            return assAse != null && !assAse.isEmpty();
    }
    return eDynamicIsSet(eFeature);
}
} //AssocImpl
```
A.3 The LHS pattern of classR, in choose mode without input parameters

```java
public static HashMap lhs() throws RuleFailedException {
    HashMap result = new HashMap(); // the output parameters are stored in the HashMap
    boolean success = false;
    try { // pattern matching begins here
        // the first edge in the traversal
        Iterator iter_P = Package.getInstances().iterator();
        while (iter_P.hasNext()) {
            try {
                // first node P
                PackageInterface P = (PackageInterface) iter_P.next();
                result.put("P", P);
                try {
                    // the second edge and node(S) in the traversal
                    SchemaInterface S = (SchemaInterface) P.getPcksch();
                    // the third edge in the traversal
                    Iterator iter_C = P.getPckcls().values().iterator();
                    while (iter_C.hasNext()) {
                        try {
                            // third node C
                            ClassInterface C = (ClassInterface) iter_C.next();
                            result.put("C", C);
                            // checking the NAC p is the name of the negative pattern
                            if (p(C))
                                throw new ClassCastException();
                            // check
                            success = true;
                        } catch (ClassCastException e) {} }
                    } catch (ClassCastException e) {} }
                } catch (ClassCastException e) {} }
            } catch (ClassCastException e) {} }
        }
        if (success)
            return result;
        else
            throw new RuleFailedException();
    }
```
Appendix B

Auxiliary classes and interfaces

B.1 Name and Entity interfaces

public interface EntityInterface {
    public String getValue();
    public void setValue(String s);
    public Collection getBelow();
    public Collection getIn();
    public void addBelowElement(Object o);
    public void deleteSource(String field, String name, Object o);
    public void addSource(String field, String name, Object o);
}

public interface NameInterface {
    public String getName();
    public void rename(String s);
    public void delete();
}

B.2 FunctionHolder, SourceHolder and ConHolder class

FunctionHolder class:

public class FunctionHolder {
    String name;
    Object Target;
    
    // for the cast Type exception
    public FunctionHolder() { name=null; Target=new Object; }
    public FunctionHolder(String N, Object T) { name=N; Target=T; }
    public Object get() { return Target; }
    public void set(String key, Object Obj) { name=key; Target = Obj; }
    public String getName() { return name; }
    public void setName(String s) { name=s; }
}
SourceHolder class:

```java
public class SourceHolder {
    String name, field; //field is the type of the connection
    Object hold; //the model element which holds the has
    //a field type connection with a name name
    SourceHolder(String f, String n, Object in)
    {
        field = f;
        name = n;
        hold = in;
    }
}
```

ConHolder class:

```java
public class ConHolder implements NameInterface {
    String name, field;
    Object Source;
    public ConHolder(String F, String N, Object S)
    {
        field = F; //Field of the function/relation(Java)
        name = N; //name of the relation in Vtcl
        Source = S;
    }
    public String getName()
    {
        return name;
    }
    public void delete()
    {
        try
        {
            Method m = Source.getClass().
                .getMethod("delete" + field,
                    new Class[] { String.class });
            m.invoke(Source, new Object[] { name });
            Source = null;
            name = null;
            //Have to be deleted from the Variables HashMap
        }
        catch (Exception e) {}
    }
    public void rename(String s)
    {
        try
        {
            Method m = Source.getClass().
                .getMethod("rename" + field,
                    new Class[] { String.class, String.class });
            m.invoke(Source, new Object[] { name, s });
            name = s;
        }
        catch (Exception e) {}
    }
}
```

B.3 EMF and Viatra2 Adapters

EMF adapter EMFInstanceAdapter class:
public class EMFInstanceAdapter extends EContentAdapter {
    // generalization
    // a reference to the manager
    protected static EMFNotificationManager NManager = null;
    public static void init(EMFNotificationManager manager){
        NManager = manager;
    }
    // initialize the manager holding reference
    public void notifyChanged(Notification notification) {
        if(!NManager.isOpen()) return; // The Manager can accept Notifications
        NManager.sendNotification(notification); // sends the Notification
        super.notifyChanged(notification); // to the manager
    }
    public static EMFInstanceAdapter
    INSTANCE = new EMFInstanceAdapter();
}

Viatra2 listener SetNameListener class:

public class SetNameListener implements
    ICoreNotificationListener {

    private ViatraIntegrationManager vManager;

    public init(ViatraIntegrationManager manager) {
        vManager = manager; // init the Manager class
    }
    public void actionPerformed(ICoreNotificationObject notification) {
        if (!vManager.getIsNotificationEnabled()) // listening for changes
            return;
        if (notification.getActionType().equals(ICoreNotificationObject.ACTION_SET_NAME)) 
            // get all involved element
            Iterator it = ((IModelElement) notification.getModel())
                .getAllInstances().iterator();
            while (it.hasNext()) {
                IModelElement instance = (IModelElement) it.next();
                try { // send the values to the manager
                    vManager.synchronizeName(instance, notification.newValue());
                } catch (VPMCoreException cex) {
                }
            }
        }
    }
}
Appendix C

Models and transformations used in the PST generation in Viatra2

C.1 The makeRelation ASM rule

```java
rule makeRelation(in CL) = seq {
    forall RA,N with find(classrelation(CL,RA,N)) do seq {
        print("private HashMap "+upperFirst(name(N))="new HashMap();");
        print("public HashMap get"+upperFirst(name(N))+"() { return "+upperFirst(name(N))+";}");
        print("public void set"+upperFirst(name(N))+"(String name,"+upperFirst(name(RA))+
            "Interface a){"+"if("+upperFirst(name(N))+"==null){"+upperFirst(name(N))+
            "= new HashMap();"+" upperFirst(name(N))+".put(name,a);"+
            " ((EntityInterface)"+upperFirst(name(N))+".get()).addSource("delete+
            "upperFirst(name(N))+"\",name,a);"+"");
        print(" public void delete"+upperFirst(name(N))+"(String a) {"+
            "if("+upperFirst(name(N))+"==null)\{return;\}+
            else(Object Temp = "+upperFirst(name(N))+"(oldName);"+
            "upperFirst(name(N))+".remove(a);"+
            "((EntityInterface)Temp).deleteSource("delete"+upperFirst(name(N))+"\",a,Temp);});
        print(" public void rename"+upperFirst(name(N))+"(String oldName, String newName)"
            "{ Object Temp = "+upperFirst(name(N))+".get(oldName);"+
            "((EntityInterface)Temp).deleteSource("delete"+upperFirst(name(N))+"\",a,Temp);"+
            "+"+upperFirst(name(N))+".remove(oldName);"+
            "+"+upperFirst(name(N))+".put(newName,Temp);"+
            "((EntityInterface)Temp).deleteSource("delete"+upperFirst(name(N))+"\",a,Temp);"+
            " Temp=null;"+
            "}");
        print(" public void delete"+upperFirst(name(N))+"all()");
        print("Iterator iter = "+upperFirst(name(N))+".entrySet().iterator();"+
            "while(iter.hasNext())"+
            "{"+
            " Map.Entry m = (Map.Entry) iter.next();"+
            "((EntityInterface)m.getValue()).deleteSource((String)m.getKey(),"delete+
            "upperFirst(name(N))+"\",m.getValue());"+
            "}"+"+
            "upperFirst(name(N))+".clear();"+";
    }
}
```
APPENDIX C. PST GENERATION IN VIATRA2

C.2 ASM rule nestedR

rule nestedR(in R) = seq
{ if(find(sequentialR(R,Fir)))
    choose Fir with find(sequentialR(R,Fir)) do seq
    { update In = Fir;
        call parseRule(In);
        iterate choose Next with find(ruleNext(In,Next)) do seq
        { call parseRule(Next);
            update In = Next;
        };
    };
    if(find(parallelR(R)))
    forall Next with find(nestedRule(R,Next)) do call parseRule(Next);
}

if(find(randomR(R))) seq
{
    update number("num")=0;
    forall Next with find(nestedRule(R,Next)) do
        update number("num")= number("num")+1;
        print("randomValue = randomGenerator.nextInt("+number("num")+");\n");
        print("switch(randomValue)\n {\n               update number("num")=0;
               // Commands to print the switch structure for the random rule
               forall Next with find(nestedRule(R,Next)) do seq
               { print("case "+number("num")+" :\n               call parseRule(Next);
               print("break;\n");
               }
            print("default : throw new FatalError("Wrong random\";\");
            };
    }

C.3 The GT rule relation2SGEdge

gtrule relation2SGEdge(in X, in Node, out T, out N, out No)) =
{
    precondition pattern lhs(X,T,N,No)= {
        'vpm'.entity'(X);
        'vpm'.entity'(T);
        'vpm'.relation'(N,X,T);
        'sGraph'.metamodel.node'(No);
        'sGraph'.metamodel.node.entity'(Rel,No,T);
    }

    action
    {
        new (sGraph.metamodel.edgenode(Enode) in Node);
        new (sGraph.metamodel.weight(Wnode) in Enode);
        new (sGraph.metamodel.weight(Ownode) in Enode);
        new (sGraph.metamodel.label(Lnode) in Enode);
        new (sGraph.metamodel.label(Lnode1) in Enode);
    }
C.4 Meta-model of the context graph

entity(sGraph.metamodel)->"
{
    entity(sGraph.metamodel.bound)->"
    entity(sGraph.metamodel.patternMatch)->"
    entity(sGraph.metamodel.type)->"
};
} function(sGraph.metamodel.bound.next,sGraph.metamodel.bound,
    sGraph.metamodel.bound);
relation(sGraph.metamodel.patternMatch.below,sGraph.metamodel.patternMatch,
    sGraph.metamodel.bound);
function(sGraph.metamodel.patternMatch.firstInc,
    sGraph.metamodel.patternMatch,
    sGraph.metamodel.patternMatch,
    sGraph.metamodel.bound);
relation(sGraph.metamodel.patternMatch.fixVariables,
    sGraph.metamodel.patternMatch,
    sGraph.metamodel.bound);
relation(sGraph.metamodel.patternMatch.in,sGraph.metamodel.patternMatch,
    sGraph.metamodel.bound);
function(sGraph.metamodel.patternMatch.negpattern,
    sGraph.metamodel.patternMatch,
    ASM.metamodel.GTPattern);
relation(sGraph.metamodel.patternMatch.nonfixVariables,
    sGraph.metamodel.patternMatch,
    sGraph.metamodel.bound);
function(sGraph.metamodel.patternMatch.pattern,
    sGraph.metamodel.patternMatch,
    ASM.metamodel.GTPattern);
function(sGraph.metamodel.patternMatch.type,sGraph.metamodel.patternMatch,
    sGraph.metamodel.type);

C.5 Meta-model of the search graph

entity(sGraph.metamodel)->"
C.6 VTML description of the ASM and GT based model transformation engine of Viatra2

entity(ASM)->""
  entity(ASM.metamodel)->"
    entity(ASM.metamodel.ASMFunction)->""
    entity(ASM.metamodel.ASMRule)->"
      entity(ASM.metamodel.ASMRuleInvocation)->""
    entity(ASM.metamodel.ActualParameter)->""
    entity(ASM.metamodel.All)->""
    entity(ASM.metamodel.And)->""
    entity(ASM.metamodel.BinaryOperation)->""
    entity(ASM.metamodel.BlockRule)->""
    entity(ASM.metamodel.CallRule)->""
    entity(ASM.metamodel.ChooseRule)->""
    entity(ASM.metamodel.ConditionalRule)->""
    entity(ASM.metamodel.Constant)->""
    entity(ASM.metamodel.ElementCreateRule)->""
    entity(ASM.metamodel.ElementDeleteRule)->""
    entity(ASM.metamodel.ElementReference)->""
    entity(ASM.metamodel.Entity)->""
    entity(ASM.metamodel.Equals)->""
    entity(ASM.metamodel.Exists)->""
APPENDIX C. PST GENERATION IN VIATRA2

entity(ASM.metamodel.ForallRule)->"";
entity(ASM.metamodel.Formula)->"";
entity(ASM.metamodel.FunctionInvocation)->"";
entity(ASM.metamodel.GTPattern)->"";
entity(ASM.metamodel.GTPatternCall)->"";
entity(ASM.metamodel.GTRule)->"";
entity(ASM.metamodel.GTRuleInvocation)->"";
entity(ASM.metamodel.GreaterThan)->"";
entity(ASM.metamodel.Implication)->"";
entity(ASM.metamodel.IterateRule)->"";
entity(ASM.metamodel.LessThan)->"";
entity(ASM.metamodel.LetRule)->"";
entity(ASM.metamodel.Level)->"";

{ entity(ASM.metamodel.Level.debug)->"debug";
  entity(ASM.metamodel.Level.error)->"error";
  entity(ASM.metamodel.Level.fatal)->"fatal";
  entity(ASM.metamodel.Level.info)->"info";
  entity(ASM.metamodel.Level.warning)->"warning";
}
entity(ASM.metamodel.Location)->"";
entity(ASM.metamodel.LogRule)->"";
entity(ASM.metamodel.ModelElement)->"";
entity(ASM.metamodel.ModelManipulationRule)->"";
entity(ASM.metamodel.MoveRule)->"";
entity(ASM.metamodel.NativeFunction)->"";
entity(ASM.metamodel.NativeImplementation)->"";
entity(ASM.metamodel.NestedRule)->"";
entity(ASM.metamodel.Not)->"";
entity(ASM.metamodel.NotEquals)->"";
entity(ASM.metamodel.Or)->"";
entity(ASM.metamodel.ParallelRule)->"";
entity(ASM.metamodel.PrintRule)->"";
entity(ASM.metamodel.QualifiedFormula)->"";
entity(ASM.metamodel.RelationCreateRule)->"";
entity(ASM.metamodel.RelationalOperation)->"";
entity(ASM.metamodel.RelationshipCreateRule)->"";
entity(ASM.metamodel.RelationshipDeleteRule)->"";
entity(ASM.metamodel.RelationshipType)->"";

{ entity(ASM.metamodel.RelationshipType.contains)->"";
  entity(ASM.metamodel.RelationshipType.instanceOf)->"";
  entity(ASM.metamodel.RelationshipType.supertypeOf)->"";
}
entity(ASM.metamodel.RenameRule)->"";
entity(ASM.metamodel.RuleDefinition)->"";
entity(ASM.metamodel.RuleParameter)->"";
entity(ASM.metamodel.Scope)->"";
entity(ASM.metamodel.ScopedParameter)->"";
entity(ASM.metamodel.SequentialRule)->"";
entity(ASM.metamodel.SetValueRule)->"";
entity(ASM.metamodel.SkipRule)->"";
entity(ASM.metamodel.Term)->"";
APPENDIX C. PST GENERATION IN VIATRA2

entity(ASM.metamodel.UnaryOperation)->""
entity(ASM.metamodel.UpdateRule)->""
entity(ASM.metamodel.Variable)->""
}

entity(ASM.predefs)->"
{
entity(ASM.predefs.*)->""
entity(ASM.predefs.+)->""
entity(ASM.predefs.-)->""
entity(ASM.predefs./)->""
entity(ASM.predefs.fqn)->""
entity(ASM.predefs.name)->""
entity(ASM.predefs.ref)->""
entity(ASM.predefs.value)->"
}

}
APPENDIX C. PST GENERATION IN VIATRA2

ASM.metamodel.ElementReference); function(ASM.metamodel.ElementCreateRule.element,ASM.metamodel.ElementCreateRule, ASM.metamodel.ElementReference); function(ASM.metamodel.ElementCreateRule.type,ASM.metamodel.ElementCreateRule, ASM.metamodel.ElementReference); function(ASM.metamodel.ElementDeleteRule.element,ASM.metamodel.ElementDeleteRule, ASM.metamodel.ElementReference); function(ASM.metamodel.ForallRule.formula,ASM.metamodel.ForallRule, ASM.metamodel.Formula); function(ASM.metamodel.FunctionInvocation.firstParam,ASM.metamodel.FunctionInvocation, ASM.metamodel.ActualParameter); function(ASM.metamodel.FunctionInvocation.function,ASM.metamodel.FunctionInvocation, ASM.metamodel.ASMFunction); function(ASM.metamodel.GTPattern.check,ASM.metamodel.GTPattern,ASM.metamodel.Formula); function(ASM.metamodel.GTPattern.firstParameter,ASM.metamodel.GTPattern, ASM.metamodel.RuleParameter); function(ASM.metamodel.GTPatternCall.firstParameter,ASM.metamodel.GTPatternCall, ASM.metamodel.GTPatternCall, datatypes.String); function(ASM.metamodel.GTPatternCall.mode,ASM.metamodel.GTPatternCall, ASM.metamodel.GTPattern); function(ASM.metamodel.GTPatternCall.pattern,ASM.metamodel.GTPatternCall, ASM.metamodel.GTPatternCall, ASM.metamodel.Scope); function(ASM.metamodel.GTRule.action,ASM.metamodel.GTRule, ASM.metamodel.ASMRule); function(ASM.metamodel.GTRule.firstParam,ASM.metamodel.GTRule, ASM.metamodel.RuleParameter); function(ASM.metamodel.GTRule.postcondition,ASM.metamodel.GTRule, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRule.precondition,ASM.metamodel.GTRule, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.firstParam,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation, datatypes.String); function(ASM.metamodel.GTRuleInvocation.mode,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.rule,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.scope,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.firstParam,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation, datatypes.String); function(ASM.metamodel.GTRuleInvocation.mode,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.rule,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.scope,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.firstParam,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation, datatypes.String); function(ASM.metamodel.GTRuleInvocation.mode,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.rule,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.GTRuleInvocation.scope,ASM.metamodel.GTRuleInvocation, ASM.metamodel.GTRuleInvocation); function(ASM.metamodel.LetRule.value,ASM.metamodel.LetRule, ASM.metamodel.Term); function(ASM.metamodel.LetRule.variable,ASM.metamodel.LetRule, ASM.metamodel.Variable); function(ASM.metamodel.LogRule.level,ASM.metamodel.LogRule, ASM.metamodel.Level); function(ASM.metamodel.LogRule.out,ASM.metamodel.LogRule, ASM.metamodel.Term); function(ASM.metamodel.MoveRule.element,ASM.metamodel.MoveRule, ASM.metamodel.ElementReference); function(ASM.metamodel.MoveRule.from,ASM.metamodel.MoveRule, ASM.metamodel.ElementReference);
APPENDIX C. PST GENERATION IN VIATRA2

function(ASM.metamodel.MoveRule.to, ASM.metamodel.MoveRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.NativeFunction.implementation, ASM.metamodel.NativeFunction, ASM.metamodel.NativeImplementation);
function(ASM.metamodel.PrintRule.out, ASM.metamodel.PrintRule, ASM.metamodel.Term);
function(ASM.metamodel.QualifiedFormula.variable, ASM.metamodel.QualifiedFormula, ASM.metamodel.Variable);
function(ASM.metamodel.RelationCreateRule.source, ASM.metamodel.RelationCreateRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.RelationCreateRule.target, ASM.metamodel.RelationCreateRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.RelationalOperation.leftSide, ASM.metamodel.RelationalOperation, ASM.metamodel.Term);
function(ASM.metamodel.RelationalOperation.rightSide, ASM.metamodel.RelationalOperation, ASM.metamodel.Term);
function(ASM.metamodel.RelationshipCreateRule.fromRef, ASM.metamodel.RelationshipCreateRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.RelationshipCreateRule.toRef, ASM.metamodel.RelationshipCreateRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.RelationshipCreateRule.type, ASM.metamodel.RelationshipCreateRule, ASM.metamodel.RelationshipType);
function(ASM.metamodel.RelationshipDeleteRule.from, ASM.metamodel.RelationshipDeleteRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.RelationshipDeleteRule.to, ASM.metamodel.RelationshipDeleteRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.RelationshipDeleteRule.type, ASM.metamodel.RelationshipDeleteRule, ASM.metamodel.RelationshipType);
function(ASM.metamodel.RenameRule.element, ASM.metamodel.RenameRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.RenameRule.name, ASM.metamodel.RenameRule, ASM.metamodel.Term);
function(ASM.metamodel.RuleDefinition.rule, ASM.metamodel.RuleDefinition, ASM.metamodel.ASMRule);
function(ASM.metamodel.RuleParameter.direction, ASM.metamodel.RuleParameter, datatypes.String);
function(ASM.metamodel.RuleParameter.nextParam, ASM.metamodel.RuleParameter, ASM.metamodel.RuleParameter);
function(ASM.metamodel.RuleParameter.variable, ASM.metamodel.RuleParameter, ASM.metamodel.Variable);
function(ASM.metamodel.ScopedParameter.base, ASM.metamodel.ScopedParameter, ASM.metamodel.Term);
function(ASM.metamodel.ScopedParameter.mode, ASM.metamodel.ScopedParameter, datatypes.String);
function(ASM.metamodel.ScopedParameter.name, ASM.metamodel.ScopedParameter, datatypes.String);
function(ASM.metamodel.ScopedParameter.next, ASM.metamodel.ScopedParameter, ASM.metamodel.ScopedParameter);
function(ASM.metamodel.SequentialRule.firstRule, ASM.metamodel.SequentialRule, ASM.metamodel.ASMRule);
function(ASM.metamodel.SetValueRule.elementRef, ASM.metamodel.SetValueRule, ASM.metamodel.Term);
function(ASM.metamodel.SetValueRule.value, ASM.metamodel.SetValueRule, ASM.metamodel.ElementReference);
function(ASM.metamodel.Term.nextParam, ASM.metamodel.Term,
APPENDIX C. PST GENERATION IN VIATRA2

ASM.metamodel.Term);
function(ASM.metamodel.UnaryOperation.formula,ASM.metamodel.UnaryOperation,
ASM.metamodel.Formula);
function(ASM.metamodel.UpdateRule.function,ASM.metamodel.UpdateRule,
ASM.metamodel.ASMFunction);
function(ASM.metamodel.UpdateRule.location,ASM.metamodel.UpdateRule,
ASM.metamodel.ActualParameter);
function(ASM.metamodel.UpdateRule.value,ASM.metamodel.UpdateRule,
ASM.metamodel.Term);
function(ASM.metamodel.UpdateRule.variable,ASM.metamodel.UpdateRule,
ASM.metamodel.Variable);
relation(ASM.metamodel.location,ASM.metamodel.ASMFunction,
ASM.metamodel.Constant);
relation(ASM.metamodel.negative,ASM.metamodel.GTPattern,
ASM.metamodel.GTPatternCall);
relation(ASM.metamodel.nextRule,ASM.metamodel.ASMRuleInvocation,
ASM.metamodel.ASMRuleInvocation);
relation(ASM.metamodel.rule,ASM.metamodel.NestedRule,
ASM.metamodel.ASMRule);
supertypeOf(ASM.metamodel.ASMRule,ASM.metamodel.ASMRuleInvocation);
supertypeOf(ASM.metamodel.QualifiedFormula,ASM.metamodel.All);
supertypeOf(ASM.metamodel.BinaryOperation,ASM.metamodel.And);
supertypeOf(ASM.metamodel.RelationalOperation,ASM.metamodel.BinaryOperation);
supertypeOf(ASM.metamodel.ASMRule,ASM.metamodel.BlockRule);
supertypeOf(ASM.metamodel.ASMRule,ASM.metamodel.CallRule);
supertypeOf(ASM.metamodel.BlockRule,ASM.metamodel.ChooseRule);
supertypeOf(ASM.metamodel.ASMRule,ASM.metamodel.ConditionalRule);
supertypeOf(ASM.metamodel.Term,ASM.metamodel.Constant);
supertypeOf(ASM.metamodel.ModelManipulationRule,ASM.metamodel.ElementCreateRule);
supertypeOf(ASM.metamodel.ModelManipulationRule,ASM.metamodel.ElementDeleteRule);
supertypeOf(ASM.metamodel.Term,ASM.metamodel.ElementReference);
supertypeOf(ASM.metamodel.RelationalOperation,ASM.metamodel.Equals);
supertypeOf(ASM.metamodel.RelationalOperation,ASM.metamodel.Exists);
supertypeOf(ASM.metamodel.BlockRule,ASM.metamodel.ForallRule);
supertypeOf(ASM.metamodel.Term,ASM.metamodel.FunctionInvocation);
supertypeOf(ASM.metamodel.Formula,ASM.metamodel.GTPattern);
supertypeOf(ASM.metamodel.Formula,ASM.metamodel.GTPatternCall);
supertypeOf(ASM.metamodel.RelationalOperation,ASM.metamodel.GreaterThan);
supertypeOf(ASM.metamodel.BinaryOperation,ASM.metamodel.Implication);
supertypeOf(ASM.metamodel.BlockRule,ASM.metamodel.IterateRule);
supertypeOf(ASM.metamodel.RelationalOperation,ASM.metamodel.LessThan);
supertypeOf(ASM.metamodel.BlockRule,ASM.metamodel.LetRule);
supertypeOf(ASM.metamodel.ASMRule,ASM.metamodel.LogRule);
supertypeOf(ASM.metamodel.ASMRule,ASM.metamodel.MoveRule);
supertypeOf(ASM.metamodel.ModelManipulationRule,ASM.metamodel.MoveRule);
supertypeOf(ASM.metamodel.ASMFunction,ASM.metamodel.NativeFunction);
supertypeOf(ASM.metamodel.ASMRule,ASM.metamodel.NestedRule);
supertypeOf(ASM.metamodel.UnaryOperation,ASM.metamodel.Not);
supertypeOf(ASM.metamodel.RelationalOperation,ASM.metamodel.NotEquals);
supertypeOf(ASM.metamodel.BinaryOperation,ASM.metamodel.Or);
supertypeOf(ASM.metamodel.NestedRule,ASM.metamodel.ParallelRule);
supertypeOf(ASM.metamodel.ASMRule,ASM.metamodel.PrintRule);
supertypeOf(ASM.metamodel.UnaryOperation,ASM.metamodel.QualifiedFormula);
supertypeOf(ASM.metamodel.NestedRule, ASM.metamodel.RandomRule);
supertypeOf(ASM.metamodel.ElementCreateRule, ASM.metamodel.RelationCreateRule);
supertypeOf(ASM.metamodel.Formula, ASM.metamodel.RelationalOperation);
supertypeOf(ASM.metamodel.ModelManipulationRule, ASM.metamodel.RelationshipCreateRule);
supertypeOf(ASM.metamodel.ModelManipulationRule, ASM.metamodel.RelationshipDeleteRule);
supertypeOf(ASM.metamodel.ModelManipulationRule, ASM.metamodel.RenameRule);
supertypeOf(ASM.metamodel.ASMRule, ASM.metamodel.RuleDefinition);
supertypeOf(ASM.metamodel.NestedRule, ASM.metamodel.SequentialRule);
supertypeOf(ASM.metamodel.ModelManipulationRule, ASM.metamodel.SetValueRule);
supertypeOf(ASM.metamodel.ASMRule, ASM.metamodel.SkipRule);
supertypeOf(ASM.metamodel.Formula, ASM.metamodel.UnaryOperation);
supertypeOf(ASM.metamodel.ASMRule, ASM.metamodel.UpdateRule);
supertypeOf(ASM.metamodel.ElementReference, ASM.metamodel.Variable);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.*);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.+);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.-);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs./);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.fqn);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.name);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.ref);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.source);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.target);
supertypeOf(ASM.metamodel.ASMFunction, ASM.predefs.value);