The Viatra-I Model Transformation Framework Users' Guide

Contents

1 Introduction

1.1 The VIATRA Model Transformation Framework

1.1.1 Mission statement

The mission of the VIATRA-I (VIsual Automated model TRAnsformations) framework is to provide a general-purpose support for the entire life-cycle of engineering model transformations including the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains.

1.1.2 Target application domains

VIATRA-I primarily aims at designing *model analysis transformations* to support the precise model-based systems development with the help of invisible formal methods. Invisible formal methods are hidden by automated model transformations projecting system models into various mathematical domains (and, preferably, vice versa).

More traditional model-to-model transformation within the context of the Model Driven Architecture (MDA) are also targetted by VIATRA-I , such as mapping platform independent models (PIM) to platform specific models (PSM) or shortly, *PIM-to-PSM mappings*, or *deployment transformations* from a PSM to the designated target middleware or execution environment.

VIATRA-I supports the model driven design of *code generators* as special model-to-code transformations. Unlike in many off-the-shelf modeling tools, VIATRA-I allows the designers to customize existing code generators within the same transformation framework.

VIATRA-I also serves as a *tool integration platform*. A unified view is provided for models of different front-end and back-end tools at various levels of abstraction by extensible model importers and exporters. This unified model space view allows to integrate tools by means of model transformations.

Existing application domains of VIATRA-I include dependable embedded systems, robust e-business applications and business workflows, middleware, and service-oriented architectures.

1.1.3 The approach

Models and modeling languages and transformations are all stored uniformly in the so-called *model space*, which provides a very flexible and general way for capturing languages and models on different meta-levels and from various domains, tools or technological spaces following the Visual Precise Metamodeling (VPM) approach [4]. Models taken from different tools are integrated into the model space by various importers and exporters.

Since precise model-based systems development is the primary application area of VIATRA-I , it necessitates that (i) the model transformations are specified in a mathematically precise way, and (ii) these transformations are automated so that the target mathematical models can be derived fully automatically. For this purpose, VIATRA-I have chosen to integrate two popular, intuitive, yet mathematically precise rule-based specification formalisms, namely, *graph transformation (GT)* [3] and *abstract state machines (ASM)* [1] to manipulate graph based models.

The basic concept in defining model transformations within VIATRA-I is the (graph) pattern. A pattern is a collection of model elements arranged into a certain structure fulfilling additional constraints (as defined by attribute conditions or other patterns). Patterns can be matched on certain model instances, and upon successful pattern matching, elementary model manipulation is specified by graph transformation rules. Like the Object Constraint Language (OCL) [2], graph transformation rules describe pre- and postconditions to the transformations, but graph transformation rules are always executable, which is a main advantage to OCL. Graph transformation rules are assembled into complex model transformations by abstract state machine rules, which provide a set of common control structures with precise semantics frequently used in imperative or functional languages.

1.2 The Current Release

The current release of VIATRA-I is called Release 3.0 (shortly R3.0).

You can directly contact the developer team at
`viatra [AT] optxware.hu`

1.2.1 Product features

Release R3.0 supports the following main features:

- **Modeling and transformation languages:**

  - an easy-to-understand textual language (called VTML) for capturing models and modeling languages (supported by a parser)
  - an intuitive textual language (called VTCL) for specifying model transformations defined by ASM and GT rules

- **Core engines**
  - a library for supporting the VPM metamodeling core
  - a powerful graph pattern matching engine
  - an interpreter for executable ASM and GT specifications
  - extensible model importer and exporter framework
  - parsers for the VTML and VTCL languages, and a UML2 importer

- **Graphical User Interface in Eclipse**
  - a framework view for managing multiple model spaces at a time
  - a tree viewer and editor for a model space
  - a textual editor for the VTML modeling language
  - a textual editor for the VTCL transformation language
  - a code output view as a back-end for model-to-code transformations

- **Demonstrators/Examples**
  - an initial set of examples for expressing models and transformations in the family domain.
  - a case study on transforming UML class diagrams into relational database tables

The current **Users Guide** is organized as follows

  - a brief overview on the Graphical User Interface (Chapter 2) including how to carry out initial steps;
  - Chapter 3 discusses how to write import modules to VIATRA-I ;
  - Chapter 4 describes how native functions can be implemented to extend VIATRA-I functionality by problem-specific Java code;
  - Chapter 5 specifies how to use the string manipulation library in VIATRA-I transformations;

– Chapter 6 defines how code formatters can be used to split textual output of transformations into multiple files and folders;

– Chapter 7 provides an introductory tutorial on core VIATRA-I concepts by demonstrative examples;

– Chapter 8 is a description of an object-relational case study.

1.2.2 The Development Team

The core development team of the current release of the VIATRA-I framework consists of

- Project Leaders: Dániel Varró, András Balogh
- Principal Advisor: András Pataricza
- Chief Technological Architect: István Ráth
- Transformation engine (Interpreter): Gergely Varró, Ákos Horváth, Péter Pásztor
- Graphical User Interface (editors): István Ráth, Dávid Vágó.
- Parser (and Various Importers): András Balogh
- Textual editors: Gergely Nyilas, István Ráth
- Specification, Documentation: Dániel Varró, András Balogh, Balázs Polgár

The team is also very much grateful to those people experimenting and testing the VIATRA-I framework by developing various model transformations: László Gönczy, Imre Kocsis, Máté Kovács, János Ávéd, Attila Németh, Daniel Schmidt (Kaiserslautern, Germany), Andreas Schröder (München, Germany), Dániel Tóth.

Furthermore, the following people read initial versions of the current document: Imre Kocsis, Ágnes Németh, Balázs Polgár. Their work is also highly acknowledged.

2 Graphical User Interface of VIATRA

2.1 Initial Steps with Using VIATRA

We now overview the first steps of using the VIATRA-I framework including instructions on how to install of the framework, how to parse models and transformations, and how to execute transformations.

2.1.1 Installation

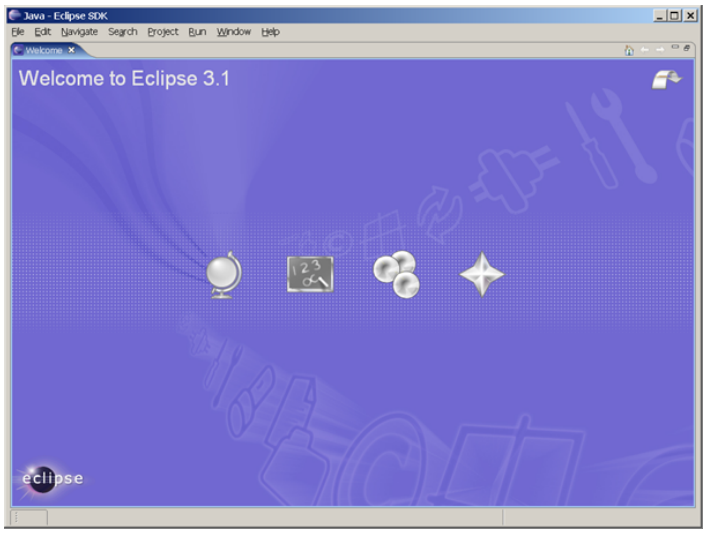
The VIATRA-I framework can be installed as an ordinary Eclipse plugin, thus we assume that Eclipse is already installed. The current version of VIATRA-I was tested with Eclipse 3.1.x and 3.2.x, but we had no problems so far with installing it under Eclipse 3.3.

Note that VIATRA-I is licensed under Eclipse Public Licence (EPL).

REMARKS

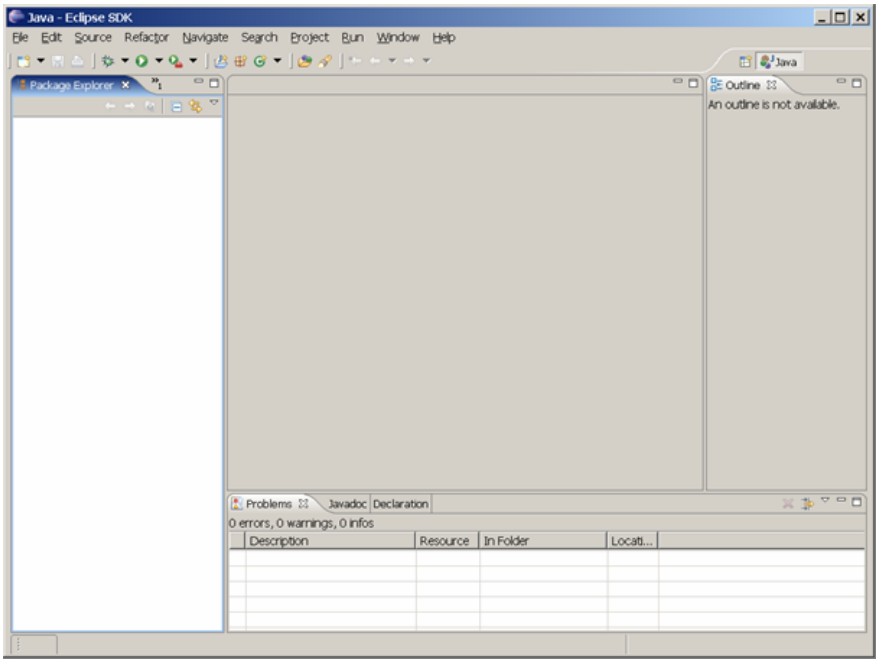
A prerequisite of the VIATRA plug-in is the existence of the EMF and GEF plug-ins which you can download and install from the Eclipse site: http://www.eclipse.org/emf and www.eclipse.org/gef! You need the versions that are compatible with your Eclipse SDK version!

Step 1: Start Eclipse and see the Welcome Screen.



Step 2: Check the Eclipse version under Help → About Eclipse SDK. (The Eclipse version should be 3.1.1 or above)

Step 3: Close the Welcome Screen by pressing the X on the Welcome tab.



Step 4: Select Help → Software Updates → Find and Install

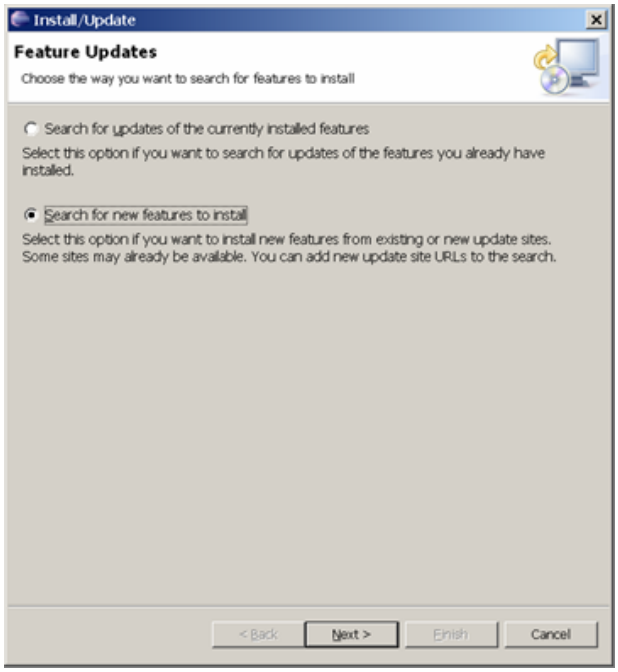
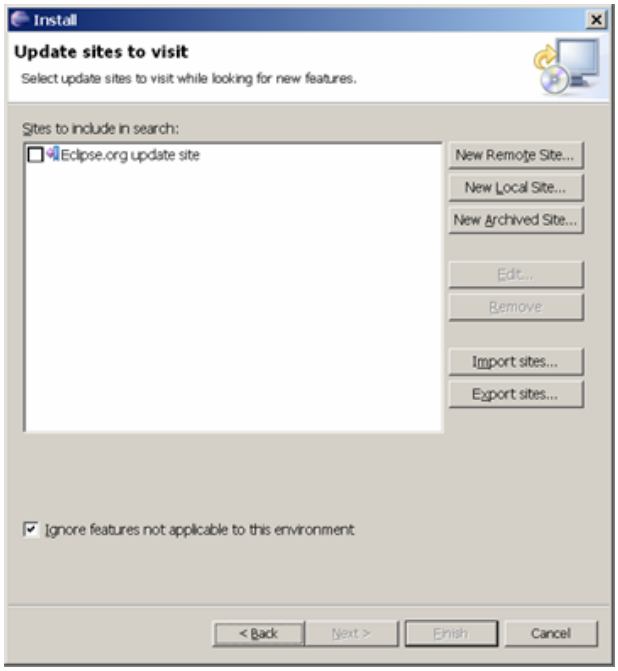


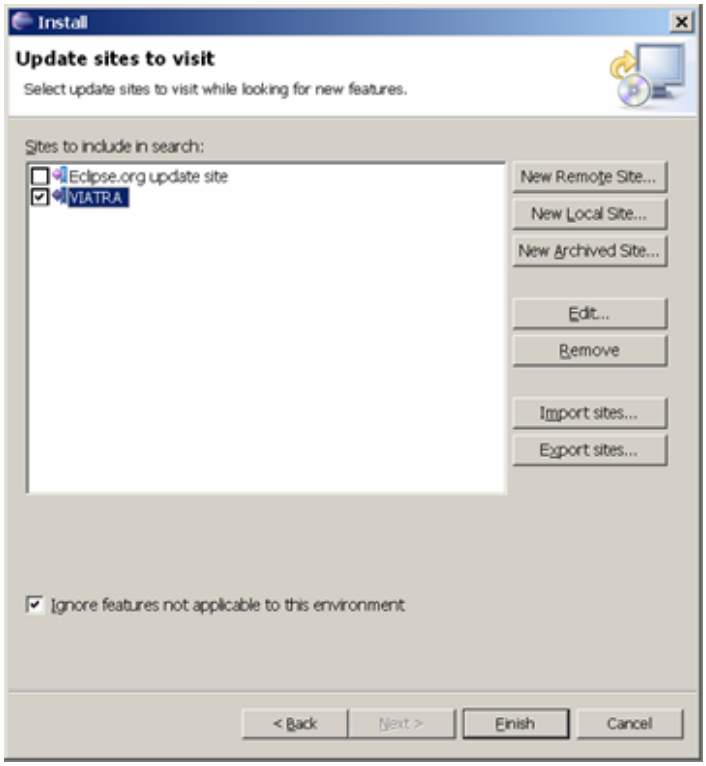
Figure 2.1: Find and Install

Step 5: Check "Search for new features to install" and press Next

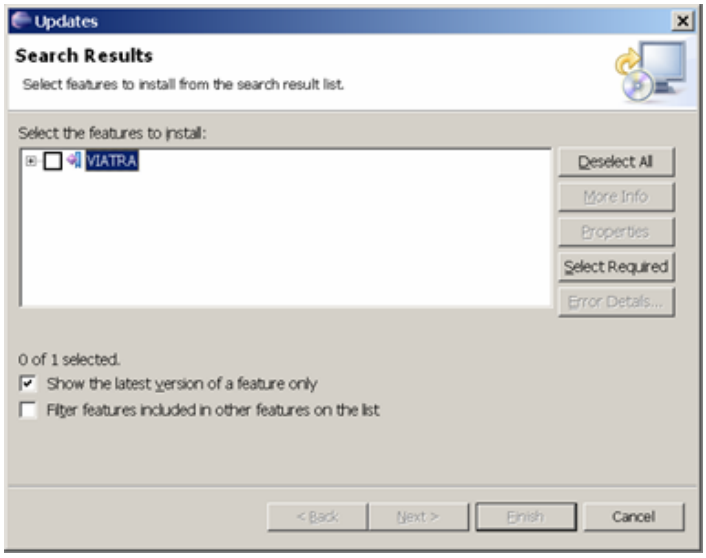


Step 6: Press New Remote Site

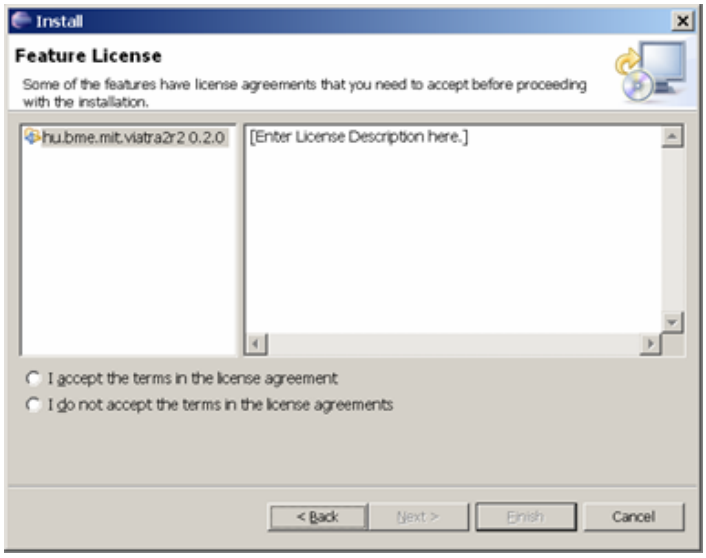
Step 7: Add the Name: "VIATRA", add the URL: http://www.optxware.hu/update and press OK



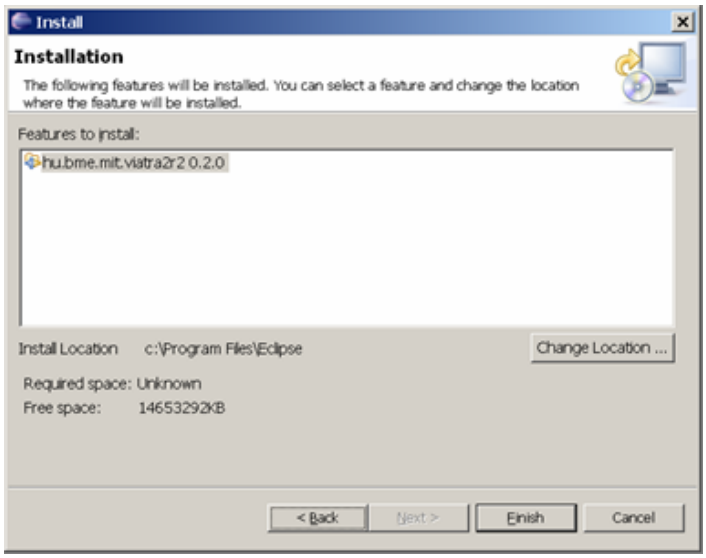
Step 8: Check the box beside VIATRA and press Finish



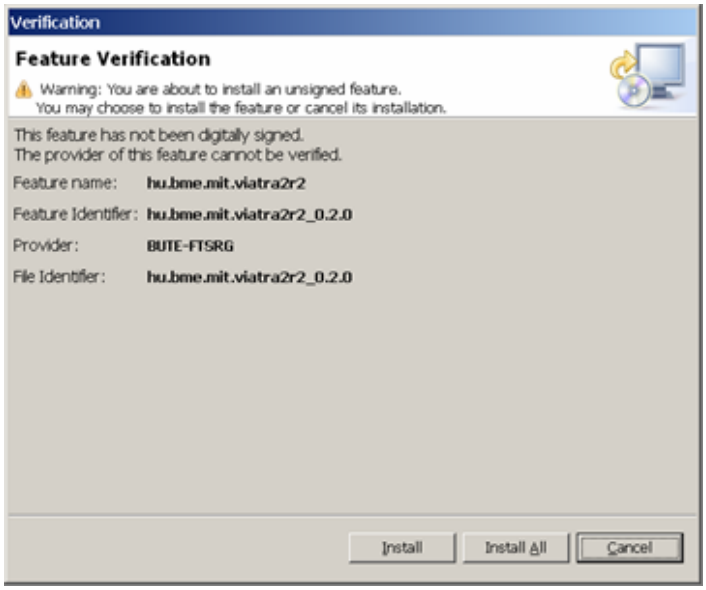
Step 9: Check the box beside VIATRA and press Next



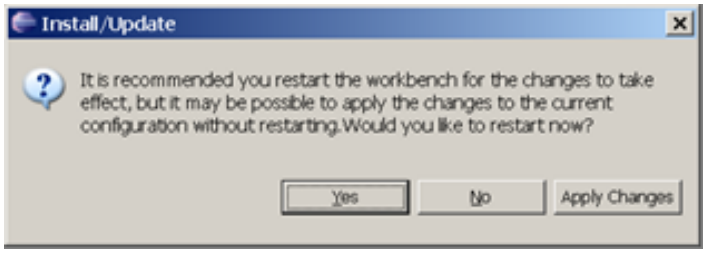
Step 10: Check the "I accept the terms and license agreement" and press Next



Step 11: Press Finish



Step 12: Press Install All



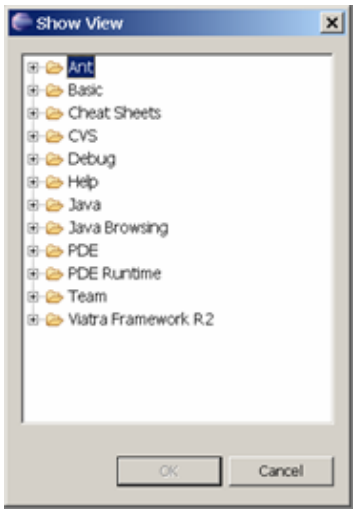
Step 13: Press Yes

After Eclipse has been restarted, you have successfully installed VIATRA-I .

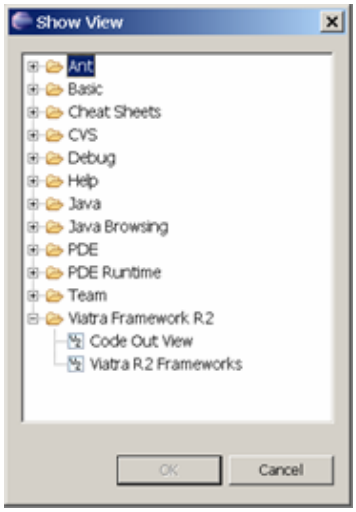
2.1.2 Setting up the VIATRA environment in Eclipse

VIATRA uses some additional views, which should be opened first

Step 1: Select Window→Show View → Other



Step 2: Expand Viatra Framework R2



Step 3: Select Code Out View and VIATRA R2 Frameworks (placed inside Viatra Framework R2 folder) and press OK.

Step 4: If not yet opened, also open the Properties View and the Error Log View (placed under the Basic folder) as described above.

Step 5: The rest of the important views (Outline, Navigator, Problems) should be open by default. If not, open it as before.

After that you can save the perspective under a new name (Save Perspective as) so that the environment could be set up easily next time.

A fully customized VIATRA-I environment is depicted in Fig. 2.2.

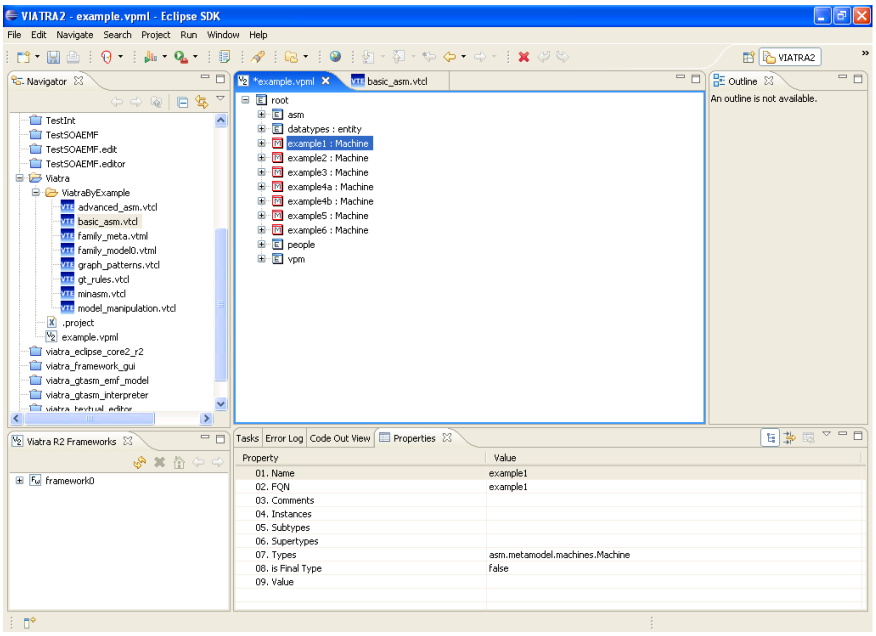


Figure 2.2: The VIATRA-I environment

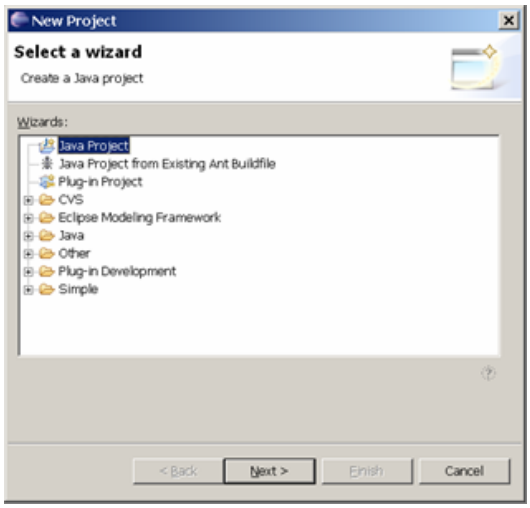
Now you are ready to use VIATRA-I !

2.1.3 Creating a new project

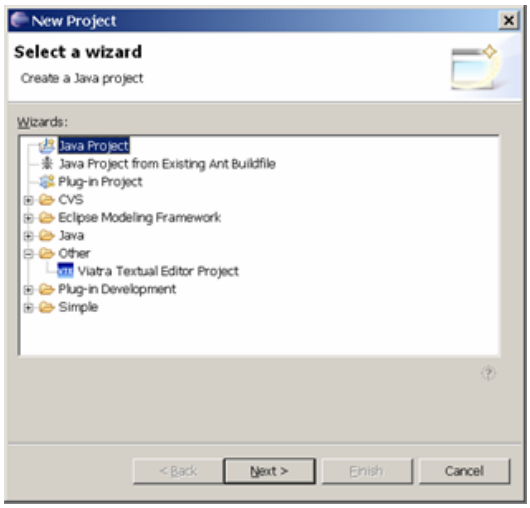
Before creating models and transformations in VIATRA-I , you need to create a new project under Eclipse.

In order to create a new project in the second way, proceed as follows:

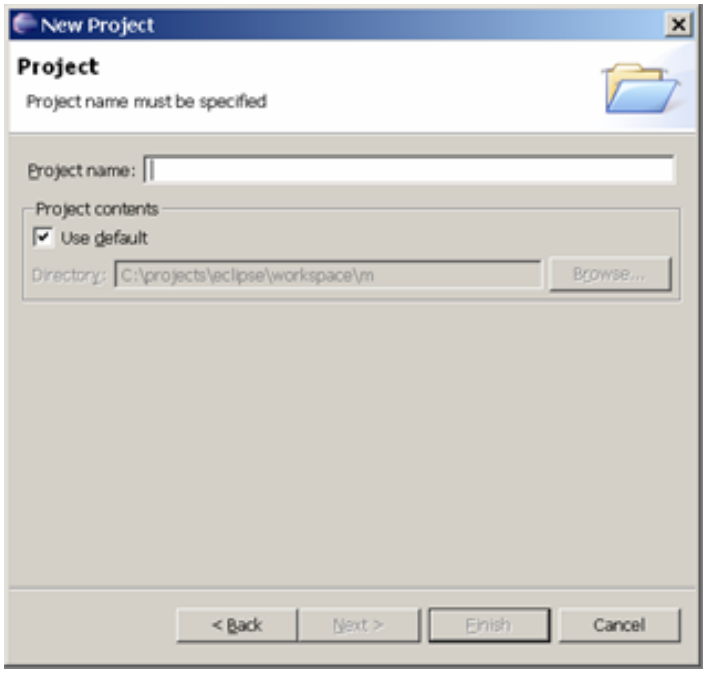
Step 1: Select File → New → Project



Step 2: Expand Other



Step 3: Select Viatra Textual Editor Project and Press Next



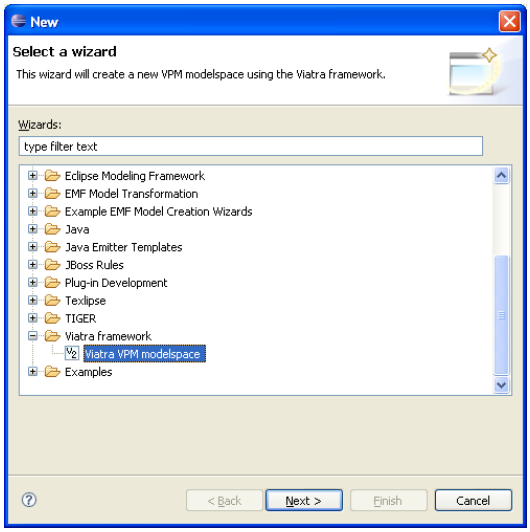
Step 4: Type a project name and press Finish

Alternatively, you can create a General Eclipse project in the usual Eclipse way (by selecting Project under General).

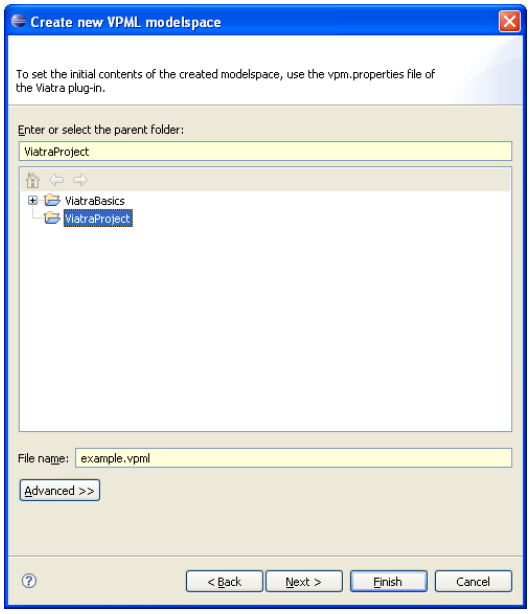
2.1.4 Creating a model space

Models, metamodels and transformations are stored in a VIATRA-I *model space*, which is serialized as a VPML file). You can create a new model space as follows.

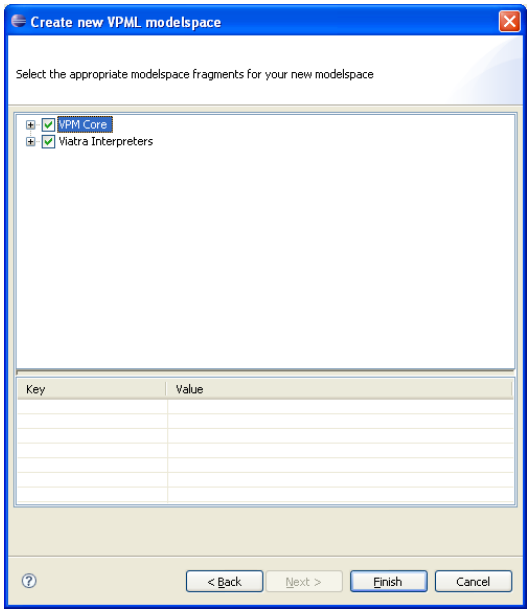
Step 1: Select File → New → Other



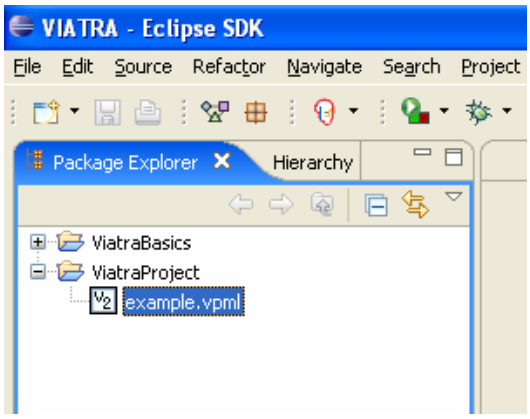
Step 2: Select Viatra VPM modelspace (under Viatra framework folder), and click Next



Step 3: Choose an appropriate Folder and Name for the model space, and click Next



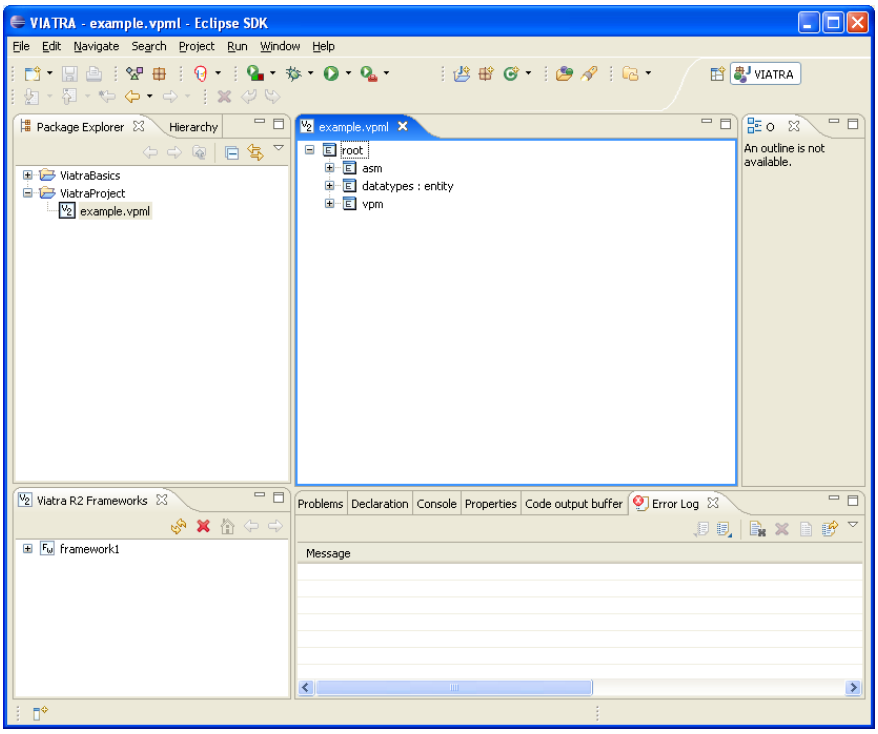
Step 4: Check both checkboxes (the first is needed for the metamodeling core, the other is for the transformation metamodel), and click Finish.



Now you have successfully created a VPML file as the model space representation in the designated folder.

2.1.5 Opening and saving a model space

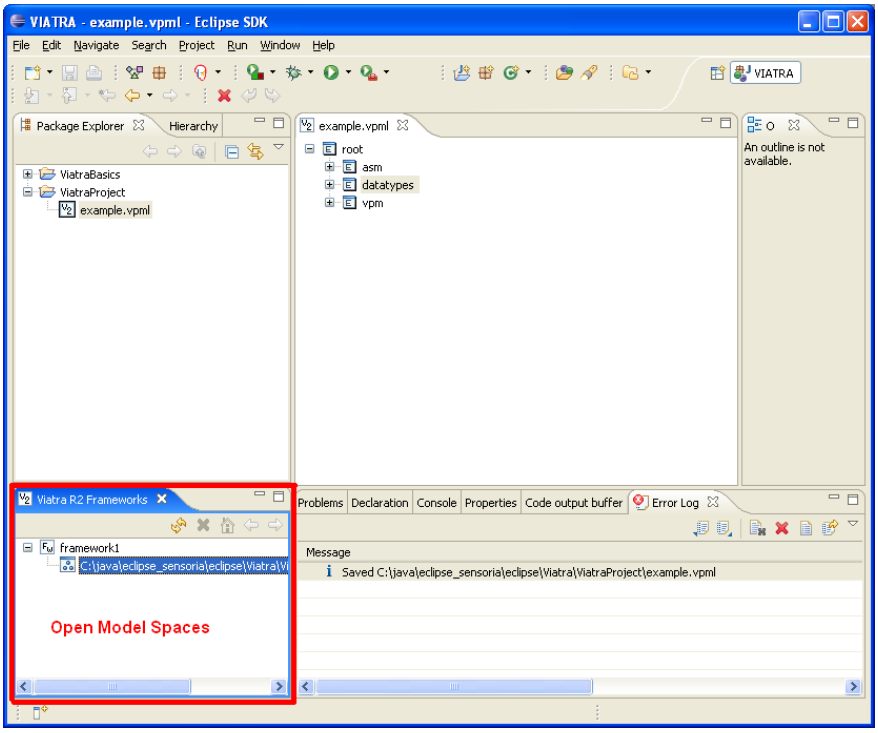
Step 1: In order to open a model space, double-click on the VPML file in the Package Explorer tab. After that, a Tree View of the model space will be opened with default entities under the root element: asm, datatypes, vpm.



When using VIATRA-I , you will make modifications to the model space by parsing models, metamodels and transformations, and of course, by executing transformations.

A model space can be saved, and thus it can be reloaded later. Note that multiple model spaces can be open at a time in parallel. All open model spaces are listed in the VIATRA-I Frameworks view.

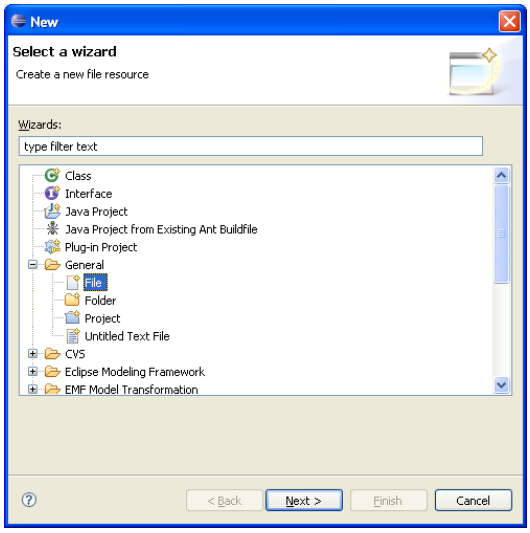
Step 2: In order to save a model space, press Ctrl-S (select or File → Save).



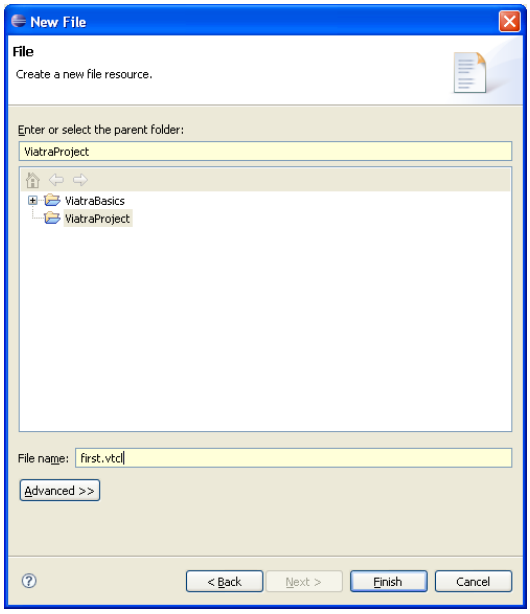
2.1.6 Creating a metamodel or transformation

Metamodels (and also models) can be described using VTML (Viatra Textual Modeling Language) files. Transformations are defined in a regular file with VTCL extension (Viatra Textual Command Language). Both VTML and VTCL files can be *created* as follows:

Step 1: Select File → New → Other

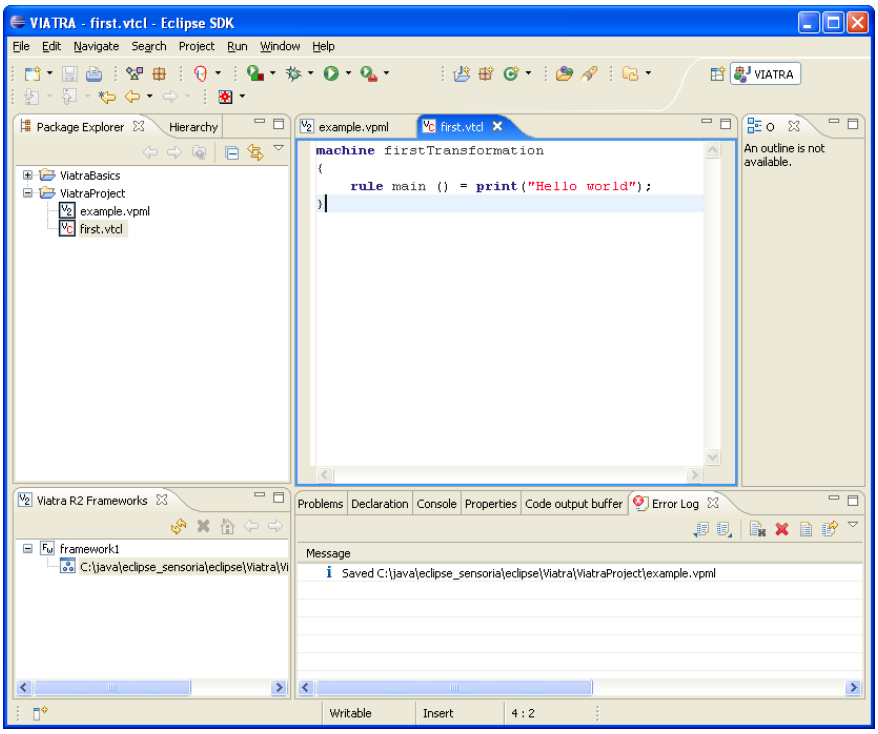


Step 2: Select File (under General folder), and click Next



Step 3: Choose an appropriate Folder and Name (with VTCL or VTML extension) for the model space, and click Finish.

Step 4: Open the new file by double-clicking on them in the Package Explorer. This opens a corresponding textual editor with syntax highlighting.

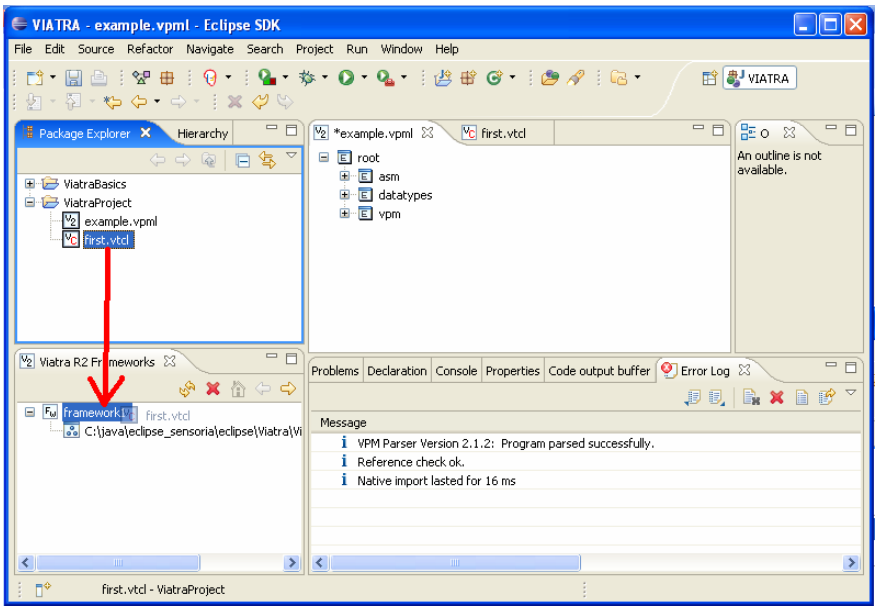


2.1.7 Parsing a metamodel or transformation

Existing VTML and VTCL files can be parsed in a similar way.

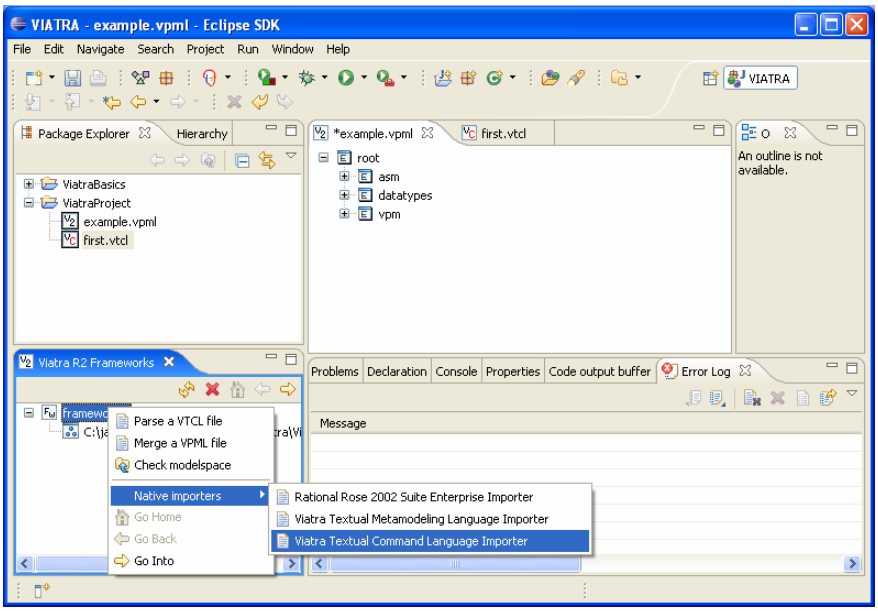
Step 1-2a: Drag-and-drop the VTML/VTCL file in the Package Explorer to the designated framework in the Viatra R2 Frameworks View.

(Note that you should drag and drop the file to the framework node itself, and not to the VPML document placed inside).

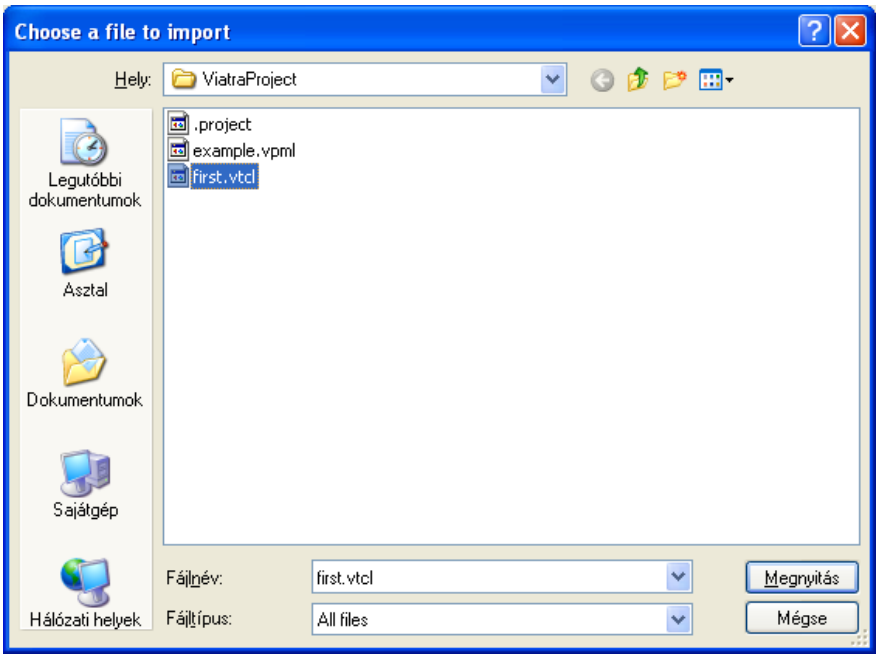


Alternatively, parsing can be done as follows:

Step 1b: Right-click on the designated framework in Viatra R2 Frameworks View, and select the appropriate importer under Native Importers



Step 2b: Select the VTCL file to parse and click Open.

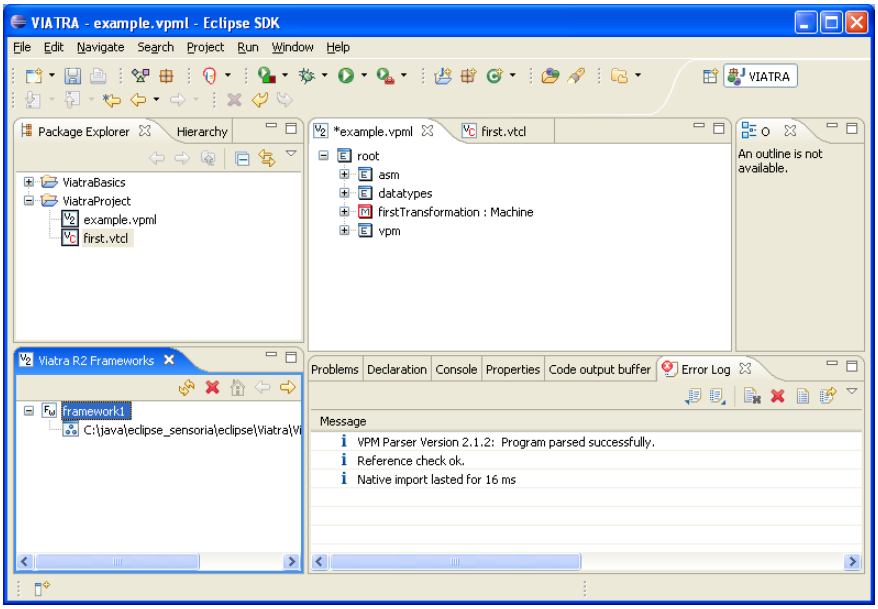


Note that a successful parsing is reported by TWO information messages:

1. VPM Parser Version 3.0.1: Program parsed successfully (which reports that syntactic parsing succeeded)
2. Reference check ok (which is for additional well-formedness checks)

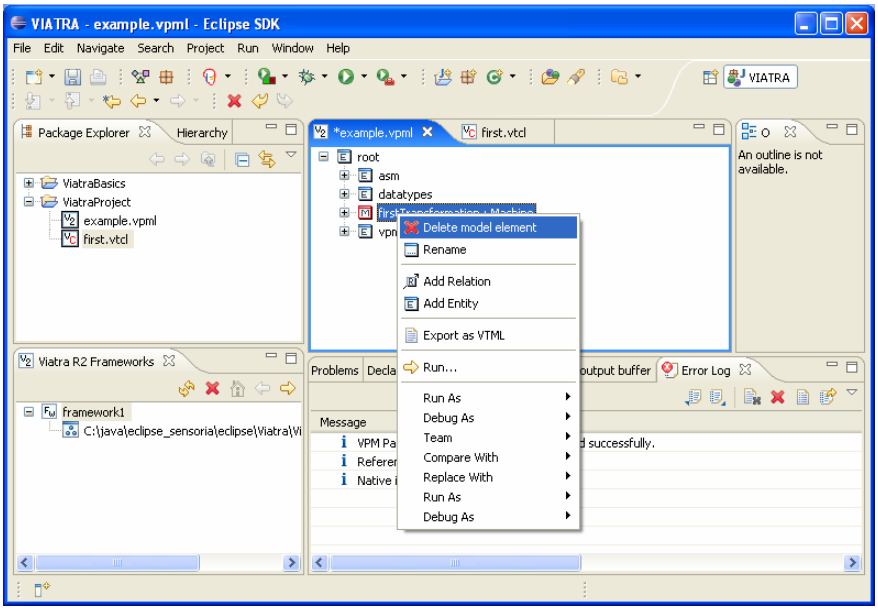
Error messages can be observed in the Error Log View.

Step 3: Note that only upon successful parsing of a VTML/VTCL should the model space reflect the changes as seen below!



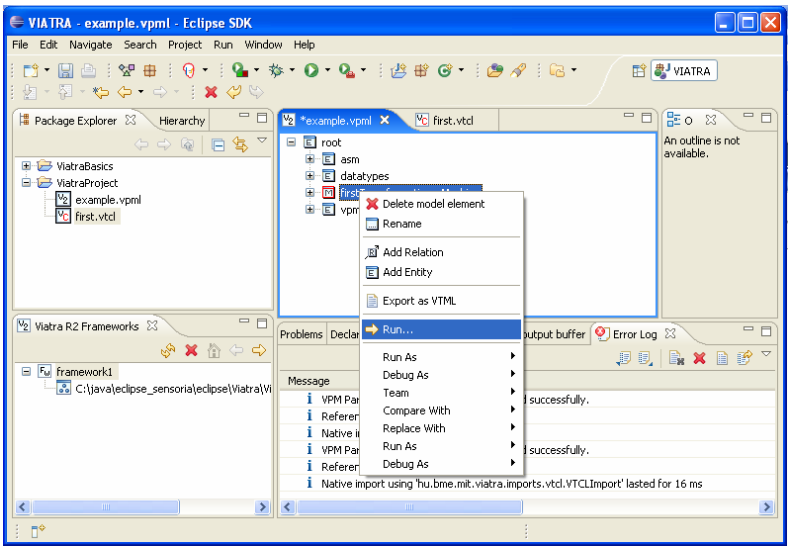
Step 4: If a model / transformation was parsed successfully, it should be deleted explicitly from the model space before reparsing by

- clicking on the entity / machine symbol in the Tree View, and pressing Delete, or
- right clicking on the machine symbol and pressing Delete model element.



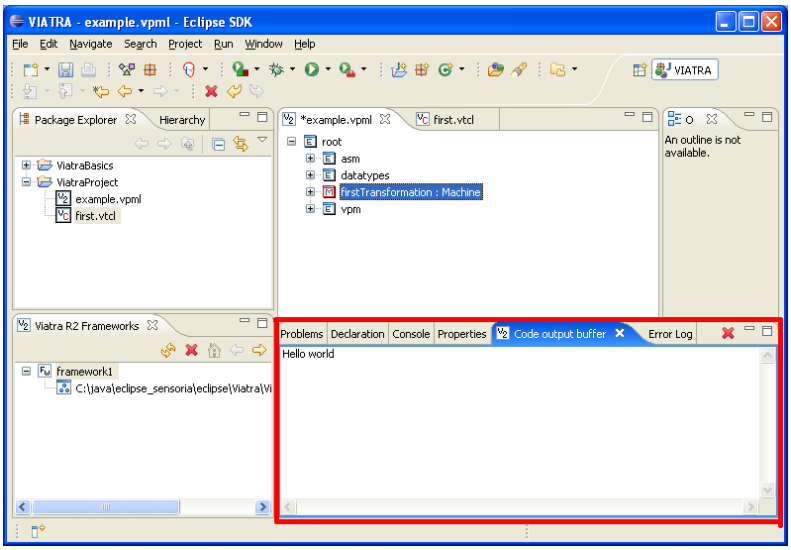
2.1.8 Executing a transformation

After successfully parsing a VTCL document, a transformation appears as an (ASM) Machine in the model space with red "M" icons. Transformations can be executed by right-clicking on the icon, and selecting "Run".



If a transformation takes an input parameter, then a dialog box appears, where parameters should be listed separated with SPACES.

Error messages are listed again in the Error Log, while the textual output of the transformation is listed in the Code Out View.



2.2 Syntax for Format String in the Tree Editor

To allow the customization of the appearance of model elements in the tree editor, we provided a framework property called "format string", which defines what is displayed for a given model element. Using the format string, you have access to various properties of the model element, and you may decide what should be shown and how that information is formatted.

The format string is a C printf() style string, where certain escape sequences refer to the various model element properties.

Each escape sequence is started with a 'width (in characters), an optional formatting environment, and the actual property name.

```
<formatString>     ::= <formatElement> <formatString> | empty string

<formatElement>    ::= <characterFmt> | <escapeSequence>
<characterFmt>     ::= any Unicode character except for '%'

<escapeSequence>   ::= '%' <escapeBody>
<escapeBody>       ::= '%' | '#' | <escapeProperty>
```

There are two trivial escape sequences, '%%' and '%#', the first shall be used when a '%' character is contained in the output string. The second ('%#') represent the empty string, it may be used, when a property value is followed by a English letter ('A'-'Z', 'a'-'z'), for example "%value%#apple" displays the value of a VPM entity and the string "apple" concatenated without any separating space characters between them.

```
<escapeProperty>   ::= [ <propertyWidth> ] [ <propertyEnv> ] <propertyName>

<propertyName>     ::= <letter> <propertyName> | <letter>
<letter>           ::= any letter of the English alphabet ('A' - 'Z' and
                         'a' - 'z' inclusive)

<propertyWidth>    ::= '[' <widthValue> ']'
<widthValue>       ::= <digit> <widthValue> | <digit>
<digit>            ::= any decimal digit ('0' - '9')
```

For every displayed property, a maximal width may be specified. If the property value is longer than this specified width, only the first (width - 3) characters are display, followed by three dots ("..."). If the width is not specified, or is greater than 99,999,999, the entire property value is displayed.

```
<propertyEnv>      ::= '{' <environmentChars> '}'

<environmentChars> ::= <environmentChar> <environmentChars> | empty string
<environmentChar>  ::= <characterEnv> | '\$' | <environmentEsc>

<characterEnv>     ::= any Unicode character except for '\%', '\}' and '\$'
```

```
<environmentEsc>   ::= '\%' '\%' | '\%' '\}' | '\%' '\$'
```

Optionally, every property reference (escape sequence) may contain an additional formatting specifier called the property environment. Such an environment is useful, when we want to use a kind of container (like "value") for a property value, but when the property is empty, we do not want to see empty pair of "-s displayed. The property environment is something like a format string within the format string, it refers to exactly one property, it just gives some extra characters, whose appearance is determined by whether the property value is empty or not.

Within this property environment, you must use the '$' character to specify where the actual property value should be placed. If you want to use '$', '%' or '}' within the environment as characters being displayed, you must escape them with a leading '%' character. Note that the escape sequences '%$' and '%}' cannot be used outside of environment definitions. If an environment contains not exactly one (non-escaped) '$' character, the behaviour is undefined (the actual implementation replaces only the last occurence of the '$' character with the property value and leaves other '$' characters unchanged).

So the processing of the property escape sequences is the following:

1. The property value is computed
2. If width was specified and the property value is longer than 'width' characters, the value is truncated to (width - 3) character and "..." is appended at the end of that truncated string.
3. If a property environment is not specified, the property value is simply displayed.
4. If a property environment was specified and the property value is not empty, the '$' token in the property environment is replaced by the (possibly truncated) property value, and that is displayed.

Finally every property has a name, which is a case-sensitive sequence of English letters. The list of supported property names, their abbreviation and their semantics are listed in the table below.

2.2.1 Supported property names

See Fig. 2.3 for supported properties in the format string.

Certain properties (value, source, target*) are only valid for either entities or relations. The value of these properties for an inappropriate model element will be an empty string.

Property name	Abbreviation	Semantics
name	*n*	relative name of the model element
NAME	*N*	fully qualified name of the model element
value	*v*	value of the entity
source	*src*	relative name of the source element
SOURCE	*SRC*	fully qualified name of the source element
target	*trg*	relative name of the target element
TARGET	*TRG*	fully qualified name of the target element
type	*t*	comma separated list of the direct types of the element
TYPE	*T*	comma separated list of fully qualified names of the direct types of the element
instance	*i*	comma separated list of the direct instances of the element
INSTANCE	*I*	comma separated list fully qualified names of the direct instances of the element
supertype	*s*	comma separated list of the direct supertypes of the element
SUPERTYPE	*S*	comma separated list fully qualified names of the direct supertypes of the element
subtype	*st*	comma separated list of the direct subtypes of the element
SUBTYPE	*ST*	comma separated list fully qualified names of the direct subtypes of the element
alltype	*at*	comma separated list of all types of the element
ALLTYPE	*AT*	comma separated list of fully qualified names of all types of the element
allinstance	*ai*	comma separated list of all instances of the element
ALLINSTANCE	*AI*	comma separated list of fully qualified names of all instances of the element
allsupertype	*as*	comma separated list of all supertypes of the element
ALLSUPERTYPE	*AS*	comma separated list of fully qualified names of all supertypes of the element
allsubtype	*ast*	comma separated list of all subtypes of the element
ALLSUBTYPE	*AST*	comma separated list of fully qualified names of all subtypes of the element
targetvalue	*tv*	the value of the target entity
targettype	*tt*	comma separated list of the direct types of the target element
TARGETSUPERTYPE	*TS*	comma separated list of fully qualified names of direct supertypes of the target element
targetalltype	*tat*	comma separated list of all types of the target model element
TARGETALLTYPE	*TAT*	comma separated list of fully qualified names of all types of the target model element
targetallsupertype	*tas*	comma separated list of all supertypes of the target model element
TARGETALLSUPERTYPE	*TAS*	comma separated list of fully qualified names of all supertypes of the target element

Figure 2.3: Viatra Editor format string property names

2.2.2 Examples

1) "%value"

Displays the value of the model element (only valid for entities)

2) "%[10]value"

Displays the value of the entity truncated to 10 characters.

3) "%[20]{"$"}value"

Displays the value of the entity (truncated to 20 character) between apostrophes, but the apostrophes are not shown, when the value is empty.

4) "%[20]{{$%}}value%#apple"

Displays the value of the entity (truncated to 20 characters) between curly braces, and the curly braces are immediately followed by the string apple. For example if the value is "cat", the displayed string will be "{cat}apple", if the value if "This sentence is much longer than 20 characters", the displayed string will be "{This sentence is ...}apple". Note that the end of the curly braces within the environment string must be escaped with the '%' character, since an unescaped '}' marks the end of the environment string.

5) "%n%[20]{ {$%}}v%{ : $}t%{ [$]}s"

Displays the name of the element, followed by the truncated value within curly braces, followed by colon and the list of direct types (the colon is displayed only if there is at least one direct type) and at the end come the list of direct supertypes within brackets (this is the default format string for entities).

3 Writing Import Modules

Import modules are one of the most important parts of the VIATRA-I Framework. They act as the entry point for all models and other source files that are needed for transformations. Importers are concrete-syntax specific, which means that even if a modeling language has a fixed metamodel, but the various modeling tools use diverse concrete syntax for model export and import, separate model importers must be implemented for each tool. For instance in the case of UML the metamodel is fixed and standardized, but the concrete syntax of XMI output of the modeling tools varies between tools and even between versions. This requires the implementation (or tailoring) of model importers for each new tool. To ease the integration of model import modules to the framework, a simple plug-in extension point is defined in the VIATRA-I core plugin that accepts new import modules. The importers can be implemented as separate Eclipse plug-ins in order to let the user control the active set of importers using the standard Eclipse configuration mechanisms.

We introduce the importer plug-in facilty and extension point of VIATRA-I in this Chapter and walk through the steps of plugin implementation using a simple running example, the implementation of a Petri Net import module. Petri Nets are used in formal verification and validation and system analysis processes. They provide a simple and powerful language to model concurrent systems.

Petri nets contain places (data stores, processing units) and transitions (action executions). Places and transitions are connected with directed arcs. Places may contain tokens (uninterpreted data units) that flow between places through transitions. A *firing* of the net is the execution of a transition. The execution takes the tokens from the places that are connected to the transition (*incoming places*) and generates tokens to the places that are reachable from the transition via a single arc. The detailed firing rules can be found in many books and articles. From our point of view, a Petri Net is a model that has to be imported to VIATRA-I .

3.1 Creating a meta model

If we want to integrate a new modeling language to VIATRA-I the first thing we have to do is to create its meta model. Meta models describe the structure of a modeling language by defining the possible elements

and their connections. Viatra uses the VPM metamodeling language for model and meta model representation, so we have to compose the meta model using VPM entities, relations, and functions. The metamodel can be created either by hand as a textual (VTML) file, or using the Visual Editor of Viatra. If the metamodel has been already prepared in the form of an UML model, it can be imported with the appropriate UML importer and transformed to VPM metamodel by a simple model transformation program.

The Petri net language has a simple structure so we can easily create the metamodel even in textual format that can be seen in the next listing.

```
entity(petrinet.metamodel)
{
        entity(pnet)
        {
                entity(place);
                entity(trans);

                relation(inArc,trans,place);
                relation(outArc,place,trans);

                relation(places,pnet,place);
                relation(transs,pnet,trans);

                function(token,place,datatypes.'Integer');
        }
}
```

The entity named `petrinet.metamodel` defines a container in the VPM model space that stores the Petri net meta model elements. The entity named `pnet` represents a Petri net instance. It contains places (`place`) and transitions (trans), and input and output arcs between them (`inArc`, `outArc`). Function `token` defines the token number for each place. Relations `places` and `transs` represent the explicit containment relation between the Petri net and its parts.

3.2 Handling concrete syntax

The metamodel represents only the abstract syntax of a modeling language, but for interfacing to modeling tools we also have to use the concrete syntax of that tool. For this reason, we have to create an import module, which consists of a parser that parses the input file (which was exported from the modeling tool), and a module that builds up the model in the VPM modelspace. As most of the modeling tools use XML-based format for model export and import, we can use the XML parsers that are included in the J2SE (Java 2 Standard Edition) contribution. We recommend the use of a DOM (Document Object Model) parser, even it is much slower than the SAX-based

(Simple API for XML Processing) ones. If a DOM parser is used, the model importer only has to traverse the DOM tree to explore the elements of the model and create the appropriate model elements in the model space.

For the Petri net example we want to export models from an XML-based format that is quite simple. This format has been created only for demonstration purposes but it can represent any simple Petri nets. The next listing is an example of the input file format.

```
<?xml version="1.0" encoding="UTF-8" ?>
<pnet name="exampleNet">
        <place id="p1" name="place1" token="2"/>
        <place id="p2" name="place2" token="0"/>
        <trans id="tr1" name="t1">
                <input idref="p1"/>
                <output idref="p2"/>
        </trans>
</pnet>
```

The `<pnet>` tag marks the whole net, and has a unique name. The `<place>` tag represents a place, with its id (for referencing between transitions and places in the xml file), its name, and its marking (number of tokens). The `<trans>` tag represents a transition between places, with its unique id and name. The `<trans>` tag may have child tags (`<input>`, and `<output>`) that represent input and output places of the transition. In the example, there is an arc between place p1 and transition tr1, and between tr1 and p2. The input plugin only has to traverse the DOM tree of the input file, and create a Petri net for each `<pnet>` element, and the appropriate child elements in it for places, transitions, and arcs.

3.3 Building up models

To build up models in the VPM modelspace, the importer has to use the `hu.bme.mit.viatra.core.IModelManager` interface that provides primitive model manipulation methods. This is the only interface that connects the Framework components to the actual modelspace. Its model management methods include queries, element creation, and element modifiifaction functions. The complete reference of all methods can be found in the VIATRA-I Framework Javadoc document. The model elements are represented through objects (`IEntity`, `IRelation`, and `IFunciton`, respectively) that contain methods only for getting information about the element (name, value, connections, types, supertypes, and so on). Manipulation of these properties is only possible through the model manager interface.

If we create an instance model we have to instantiate its elements from

the elements of the meta model of the language. This means that the meta model must be in the model space if we want to import instance models, and that the importer has to ensure the proper instantiation of the elements from the meta model. For doing this the importer has to have references to the meta elements. We recommend the use of attributes in the importer class for each meta element, and store the references to the VPM elements in these attributes. The references can be built up in the initialization phase of the import process, and can be used during the import. It is important to note that the references must be rebuilt before each file import, because the model space references can be become outdated between two imports.

The following listing is a part of the Petri net importer and demonstrates the usage of the concepts discussed above.

```
  //Attribute for storing reference to the Petri net meta element
  IEntity PNET =  null;
  ...

   public void init(IModelSpace m, Logger l)
throws VPMRuntimeException {
     //Getting reference to the model manager
     IModelManager mm = m.getModelManager();

     //getting reference to Petri net meta element
     PNET = mm.getEntityByName("pnet");
     if (PNET==null)
     { //if the element is not in the model space, throw an exception
       l.error("Petri net meta element not found.");
       throw new VPMRuntimeException(
              "Error while initializing Petri net model importer");
     }
     ...
```

First, we declare an attribute (`PNET`) to store the reference to the Petri net meta element. Then, in the initialization phase, we query the model space for the meta element, and store it in the attribute. If the element is not present in the model space, the program throws an exception. This simple pattern can be applied to any meta models and importers. (the UML importers supplied with the VIATRA-I Framework also use this pattern.)

If the initialization is complete, the next step is the actual parsing of the file, and building up the instance model in the model space. The parsing is done by the XML DOM parser, and the result is a reference to the root of the DOM tree. The traversal of the tree can be done by using one of the traditional tree traversal alghorithms, but we recommend using separate methods for all tag (for node, trans, pnet tags in our example). This way it is easy to find the sources of import problems, and it is easy to extend or modify the program in case of format or meta model changes. The next example shows the

place processing part of the input module.

```
HashMap hm = new HashMap();
void processPlace(IEntity pn, Element e) throws Exception
{
   String n = e.getAttribute("name");
   String id = e.getAttribute("id");
   String t = e.getAttribute("token");
   //create a new entity in the model space
   IEntity p = mm.newEntity(n,pn);
   //set its type to pnet.place
   mm.newInstanceOf(PNPLACE,p);
   //store it in the hash map
   hm.put(id,p);
   //create the pnet.places relation
   IRelation r = mm.newRelation(pn,p,pn);
   //between the net and the place
   mm.newInstanceOf(PLACES,r);
   //create an entity for the token number
   IEntity ii = mm.newEntity(p);
   mm.setValue(ii,t);
   mm.newInstanceOf(INTEGER,ii);
   //create another relation to connect the place
   IRelation r2 = mm.newRelation(p,ii,p);
   //and the token number
   mm.newInstanceOf(TOKEN,f2);
        }
```

We create a hash map that contains references for the places in the model (to create arcs between them) based on their XML ids. The processing of a `<node>` tag starts with the query of its attributes (name, id, token number). Then, we create a new entity in the model space, and set its type to pnet.place (this means that the new entity is an instance of the pnet.place meta element). After that we create a relation between the actual Petri net and the new place that represents the containment relation between them, and finally we create an entity and a funciton to represent the number of tokens in the place. The whole model is built up using simple methods like this.

If the whole model is built up, the import process ends and the import module is deactivated. Please note, that there is no guarantee that the module instance will be reused, so it has to release any resources that have been used during the import period (files, network resources, and so on).

3.4 Structure of an import plugin

The structure of an import plugin is highly determined by the Eclipse plugin concept. Eclispe plugins consist of at least a plugin manifest (`plugin.xml`), and a plugin class. The manifest describes the properties of the plugin, including the name, vendor, version information as well as the list of extension the plugin provides and extension points

it offers to other plugins. The extension mechanism of Eclipse plugins is discussed in details in many online books so we focus on the importer-specific issues.

The VIATRA-I import modules have to extend the hu.bme.mit.viatra.-core2.modelimport extension point. The definition of the extension can be seen in the next Listing. The Listing contains the manifest of our sample Petri net import module. The extension must have an unique id, the reference to the extension point, and in this case, an `<importer>` element that contains a reference to the importer factory class.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="viatra.nat.gen.test"
   name="Test Plug-in"
   version="0.0.1"
   provider-name="BUTE"
   class="hu.bme.mit.viatra.natives.gen.test.TestPlugin">

   <runtime>
      <library name="test.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="hu.bme.mit.viatra.core2"/>
   </requires>
   <extension
         id="hu.bme.mit.import.petrinetimport"
         point="hu.bme.mit.viatra.core2.modelimport">
          <importer class=
          "hu.bme.mit.viatra.natives.gen.test.PNImporterFactory"/>
   </extension>
</plugin>
```

The importer must follow the *factory* design pattern, and the factory class must implement the hu.bme.mit.viatra.imports.NativeImporterFactory interface. This interface contains only two methods. The getImporter-Name() method must return a descriptive name for the importer, and the getImporterInstance method must return an importer instance. The factory is responsible to properly initiate the importer instance using the supplied reference to the model space. Importer instances have to implement the hu.bme.mit.viatra.imports.NativeImporter interface. This interface defines several methods, but the key method is the processFile() that is called by the framework to start the import process.

The purpose of the factory pattern is to ensure that the framework has a permanent connection to its importers (the reference to the factory classes), but the importer instances can be garabage collected right after the import is done. If the importer instances cannot be destroyed, the whole model space will reside in the memory, because of the cross references bewteen them. This leads to an extensive memory consuption or even to memory leakage. Because of this, importer implementors have to ensure that the factory will not held any references to the created importer instances, or the importer instances will clean up all references to the model space after the import is done. We recommend the usage of the first option, that way the importers are destroyed after the import process, because the framework will clear all references to them.

We can state that an importer has to have at least four components. One is the plugin manifest, the second is the default plugin class, the third is the importer factory class, and the last is the importer class itself. This simple structure ensures the easy extension of the framework and also the proper memory management.

3.5 Installing a new importer

If a new importer is implemented, it can be packaged using the Eclipse plugin development environment, and then can be easily copied to the `plugins` directory of the target Eclipse framework instance. After restarting Eclipse, the new importer will be ready to use. Please note, that the appropriate meta model has to be parsed or imported into the model space before the importer can be used.

4 Writing New Native Functions for VIATRA

Native functions can be used to implement complex computations, user interaction (through GUI), or other functionality in VIATRA-I model transformations. Native functions are implemented in Java and are parts of an Eclipse plug-in; therefore they can utilize all standard Java libraries and other Eclipse plug-ins.

4.1 Implementing a New Function

In order to implement a native function, a new Eclipse plug-in project must be created. The project must depend on the viatra_eclipse_core2_r2 plug-in. A native function is declared using the *nativefunction* extension point of the core plug-in. An extension to this point should contain at least one function element. The *function* element has three attributes: a unique id, a VTCL function *name*, and the fully qualified Java *class* name that implements the native function. This data builds up the declaration of the function.

The implementation of the function is a Java class (referred in the function configuration element). The implementation class must implement the *hu.bme.mit.viatra.natives.ASMNativeFunction* interface. This interface declares the following methods:

- `public String getId();`

  Returns the unique id of the native function.

- `public String getName();`

  Returns the VTCL name of the native function.

- `public String getDescription();`

  Returns the textual description of the native function (optional).

- ```
  public Object evaluate(IModelSpace msp,
    Object[] params) throws VPMRuntimeException
  ```

  Contains the implementation of the native function. The *msp* parameter is a reference to the actual model space. The function gets full access to the VPM model space, and can execute arbitrary model manipulation instructions on it. The *params* input parameter contains the input parameters of the function.
  ```

As the number and type of input parameters is not declared, a single native function can implement operations with different signatures. The number and type of input parameters should be validated by the function.

The return value of the method contains the return value of the native function. If runtime error occurs, the method should throw *VPMRuntimeException* that will terminate the execution of the model transformation and the message will be displayed in an error dialog to the user.

4.2 Data Type Mapping

As Java and VTCL data types are different, there is a mapping between them. The following Table summarizes the mappings:

VTCL Type	Java Type
Double	java.lang.Double
Integer	java.lang.Integer
String	java.lang.String
Boolean	java.lang.Boolean
Entity reference	hu.bme.mit.viatra.core.IEntity
Relation reference	hu.bme.mit.viatra.core.IRelation
"JavaNative Value"	Any other Java type

The basic and model element types can be used freely both in Java and VTCL and can be exchanged between native functions and GTASM code. In case of *JavaNativeValue* the value can be stored in ASM variables, and can be passed to other native functions, but no computation can be done using it on the GTASM side. This allows the interaction between subsequent native function calls (e.g., passing session variables).

4.3 Deployment

The plug-in containing native functions can be deployed using the standard Eclipse deployment process, and can be used in any VIATRA-I environment (containing all prerequisites of the plug-in).

The plug-in must be present in the environment both during transformation parse and runtime in order to be able to use native function calls to its functions.

5 String Manipulation Library

This chapter is the reference guide of the String manipulation library of VIATRA-I . The aim of the library is the implementation of the Java-like String manipulation functions in VIATRA-I model transformations. This enables the complex attribute manipulations and supports formatted code output generation.

5.1 Requirements and Installation

- The library is compiled for Java SDK 5.0. It requires VIATRA-I version (R3.0) or greater, and Eclipse 3.1 or greater.
- The library consists of a single jar (if already not present in the framework) file that has to be copied into the Eclipse plugins directory. After restarting Eclipse, the functionalities of the library will be automatically enabled.

5.2 Native functions of the library

- *str.compareTo(String,OtherString)* This function takes exactly two parameters. It compares the two parameters lexicographically. It returns 0 if the two strings are identical, a value less than zero if the first parameter is lexicographically less, and a value greather than zero if the second parameter is less than the first. If the parameter count is not 2, the function returns *undef*.
- *str.compareToIgnoreCase(String,OtherString)* This function takes exactly two parameters. It compares the two parameters lexicographically omitting the case of characters. It returns 0 if the two strings are identical, a value less than zero if the first parameter is lexicographically less, and a value greather than zero if the second parameter is less than the first. If the parameter count is not 2, the function returns *undef*.
- *str.endsWith(String,Substring)* This function takes exactly two parameters. It returns true, if the *String* ends with the given *Substring*. If the parameter count is not 2, the function returns *undef*.

- *str.equalsIgnoreCase(String,OtherString)* This function takes exactly two parameters. It compares the two parameters lexicographically (normalizing all characters to lower case) and returns true, if they are equivalent. If the parameter count is not 2, the function returns *undef*.
- *str.format(Format,Objects)* This function takes at least one parameter. Creates a formatted string based on the Format parameter (for information on the possible formattings, please see: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html-#syntax). The 2nd and following parameters are optional, and are used as parameters for the format string. If the parameter count is 0, the function returns *undef*.
- *str.indexOf(String,SubString,Offset?)* This function takes two or three parameters. It returns the index of the first occurence of *SubString* in *String*. If the Offset is specified, it starts from that point in *String*. If the parameter count is less than two or greather than three, the function returns *undef*.
- *str.lastIndexOf(String,SubString,Offset?)* This function takes two or three parameters. It returns the index of the first occurence of *SubString* in *String* starting from the end of the character string. If the Offset is specified, it starts from that point (as endpoint) in *String*. If the parameter count is less than two or greather than three, the function returns *undef*.
- *str.length(String)* This function takes exactly one parameter. It returns the length of the string. If the parameter count is not 1, the function returns *undef*.
- *str.matches(String,RegExp)* This function takes exactly two parameters. It returns true, if the RegExp regular expression matches the *String*. If the parameter count is not 2, the function returns *undef*.
- *str.regionMatches(String1,Offset1,String2,Offset2,Length,IgnoreCase?)* This function takes at least 5 or 6 parameters. It returns true, if the following sub strings (each containing *Length* characters) is equal: *String1* from position *Offset1*, and *String2* from position *Offset2*. The sixth, optional parameter specifies whether to ignore the case of characters during comparison. If the parameter count is 0, the function returns *undef*.
- *str.replaceAll(String,Pattern,Replacement)* This function takes exactly three parameters. It replaces all occurences of Pattern in *String* to the *Replacement* substring. The Pattern parameter is a regular expression (see Java API documentation). If the parameter count is not 3, the function will return *undef* value.

- *str.replaceFirst(String,Pattern,Replacement)* This function takes exactly three parameters. It replaces the first occurence of Pattern in *String* to the *Replacement* substring. The *Pattern* parameter is a regular expression (see Java API documentation). If the parameter count is not 3, the function will return *undef* value.
- *str.startsWith(String,Substring,toffset?)* This function takes two or three parameters. It returns true, if *String* starts with *Substring*. If toffset is specified, then the original string is evaulated from the given index (instead of the start). If the parameter count is not 2 or 3, the function returns false.
- *str.substring(String,start,end?)* This function takes two or three parameters. It returns a part of *String* starting from position *start* and ending at position *end-1*. If parameter end is unspecified, the function extends the string part to the end of the original *String*. If parameters 2 and 3 are not integers, the function returns *undef*.
- *str.toLowerCase(String)* This function takes exactly one parameter. It converts all the characters of *String* to lower case using the settings of the current system locale. If the parameter count is not 1, the function returns *undef*.
- *str.toUpperCase(String)* This function takes exactly one parameter. It converts all the characters of *String* to upper case using the settings of the current system locale. If the parameter count is not 1, the function returns *undef*.
- *str.trim(String)* This function takes exactly one parameter. It returns a copy of the *String* parameter with leading and trailing whitespace characters omitted. If the parameter count is not 1, the function returns *undef*.

5.3 Usage

After installation, the functions of the library are usable in GTASM transformation programs. The user can use them everywhere, where normal embedded functions and operators can be used. For instance a useage of the format function in a print rule:

```
print(str.format("apple: %d %d",1,2));
```

6 Writing Code Formatters

6.1 Introduction

The current chapter is the reference guide of the Eclipse Code Output Formatter of VIATRA-I . The aim of this component is to support the source code generation from VIATRA-I to an arbitrarty target language.

6.2 Requirements and Installation

6.2.1 Requirements

The component is compiled for Java SDK 5.0. It requires VIATRA-I version (R3.0.0) or greater, and Eclipse 3.1 or greater.

6.2.2 Installation

The component consists of a single jar file that has to be copied into the Eclipse plugins directory. After restarting Eclipse, the functionalities of the component will be automatically enabled.

6.3 Basic functionality

The component acts as a filter and router for the character stream that is produced by the GTASM print rules. The output of these rules is the input of the code formatter, and the output is a set of files in the Eclipse project structure.

The basic features of this component include

- *Multiple output file support.* The output can be redirected to files, multiple files can be generated in a single code generation run
- *Multiple output folder support.* The output files can exist in multiple folders. If a destination folder does not exist, the formatter will create it.
- *Multiple output project support.* The output of the coed generation can be written to multiple projects. If a project does not exist, the formatter will create it.

- Support for hand-coded part preservation There is a possibility to preserve hand-coded parts of the generated files between code generation runs. This enables the mixing of automatically and manually generated parts in a single file.

6.4 Settings

The basic code formatter settings can be found in the framework properties. To open the properties, first select the actual VIATRA-I framework in the VIATRA-I frameworks view. After that, the standard Eclipse Properties view will show the framework properties. The Eclipse code output formatter properties can be found on the *code-out* page. The following properties are defined for the Eclipse code output formatter:

- *efile.project:* The name of the initial output project. This will be the target of the generated files unless the code generator specifies an other runtime.
- *efile.basedir:* This will be the base folder in the output project. All target file paths will be relative to this.
- efile.autolinefeed: If this property is set to yes, the formatter will print an additional line feed character after each output string (after each print instruction). This can be used for debugging purposes.

6.5 Runtime settings

There are several special character strings that can be sent to the formatter via normal print rules, and are interpreted as commands (and are not printed to the output). The runtime setting are valid only for a single transformation execution. At the begining of a new transformation execution the framework default settings will be reset.

- `Output project specification`

  Syntax: "//!!PROJECT=< *projectname* >" *Description:* This string instructs the code output formatter to use the specified project as output. If the project does not exist in the actual workspace, it will be created. If the project exists, but it is closed, it will be opened.

  Example:

```
print("//!!PROJECT=sample_project");
//This will redirect the output into project
// sample_project.
```

- `Output base folder specification`

  Syntax: "//!!BASEDIR=< *foldername* >"

  Description: This string instructs the code output formatter to use the specified folder as base folder. If the folder does not exist in the actual project, it will be created. The name can be a path that is relative to the actual project.

  Example:

```
print("//!!BASEDIR=src/examples");
//This will set folder src/examples as base folder.
```

- `Output file specification`

  Syntax: "//!!FILE=< *filename* >"

  Description: This string instructs the code output formatter to use the specified file as output. If the file does not exist in the actual project, it will be created. The name can be a path, that will be relative to the base folder.

  Example:

```
print("//!!FILE=src/example.txt");
//This will redirect the output into file example.txt
// in the src folder.
```

- `Output file end`

  Syntax: "//!!ENDFILE

  Description: This string instructs the code output formatter to close the actual output file. After that, all output will be ignored until a new file is opened using the !!//FILE= instrcution.

  Example:

```
print("//!!ENDFILE");
//This will close the actual file.
```

6.5.1 Manual code separation related settings

These commands support the manual code separation in the output files.

- `Comment prefix setting`

  Syntax: "//!!COMMENT= < *string* >"

  Description: This string sets the code output comment prefix that is used for prefixing manual code block markers in the source code. The default setting is "//!!", as "//" is interpreted

as one-line comment by several programming language compilers (C/C++/C#/Java). In case of other languages (for instance Python) the prefic can be set to any appropriate string (in case of Python '#' is the comment prefix character).

Example:

```
print("//!!COMMENT=\#\#!!");
// This will set the comment prefix to "\#\#!!".
```

- `Manual block statement`

  Syntax: "//!!USER CODE BEGIN < *blockid* >" "//!!USER CODE END"

  Description: These strings mark the begining and end of a manual code block. The comment prefix is always the actual comment prefix (that means that the command format can change!). The block id is a unique id (in the file) that identifies the manual code block. The new and old files will be merged based on these ids. If an old block id cannot be matched, it will be deleted. This is the only command that will also be printed to the output file. The user can write any code between the two lines.

  Example:

```
print("//!!USERCODE BEGIN main,1");
print("//Place your code here");
print("//!!USER CODE END");
//This will print a manual block into the output file.
```

6.6 Usage

We illustrate the usage of the Eclipse Code Output formatter on a basic example:

```
machine hello{
rule main() = seq{
   //setting output project
   print("//!!PROJECT=my_java_project");
   //setting base folder
   print("//!!BASEDIR=src");
   //setting comment prefix
   print("//!!COMMENT=//****");

   //new file
   print("//!!FILE=hu/optxware/demo/Hello.java");

   print("package hu.optxware.demo;");
   print("public class Hello{");
   print("\tpublic static void main(String arg[]) {");
   print("\t\tSystem.out.println(\"Hello world!\");");
```

```
    //manual code block
    print("//****USER CODE BEGIN (main,1)");
    print("//Please write your custom hello message here!");
    print("//****USER CODE END");

    print("\t}");
    print("}");

    //closing file
    print("//!!ENDFILE");
} }
```

7 VIATRA Transformations by Example

In the current chapter, we give an insight into the main modeling transformation concepts of VIATRA-I by using small examples. Our intention here is to encourage VIATRA-I users to start experimenting with the system without reading the entire specification of the language. The following step-by-step introduction of will guide the reader through the most important VIATRA-I constructs.

7.1 Definition of Metamodels and Models

The role of the **VPM model space** is to provided a permanent and well-structured storage of information uniformly for models, metamodels and transformations following the Visual and Precise Metamodeling paradigm [4].

After starting VIATRA-I a default model space is initialized containing some core metamodels. This model space can be extended with additional models and metamodels by using the Viatra Textual Metamodeling Langauge (VTML).

In the current section, we only demonstrate the most crucial metamodeling concepts how to define a VPM model space using the VTML language. These issues will be discussed by using the domain of pedigrees.

7.1.1 Definition of metamodels

In a typical scenario, transformation design starts with the construction of a **metamodel**. This phase captures the main concepts of a domain (or a modeling language) and the relations between these concepts in an abstract graph representation. In the following example, we describe a metamodel of the domain of pedigrees capturing family relations of people first in a graphical MOF notation (Fig. 7.1), then step-by-step in VTML.

Entities. **Entities** define the main concepts of a modeling domain.

```
// Definition of the domain
entity(people)
{
 // Compound entity definition
 entity(metamodel)
 {
```

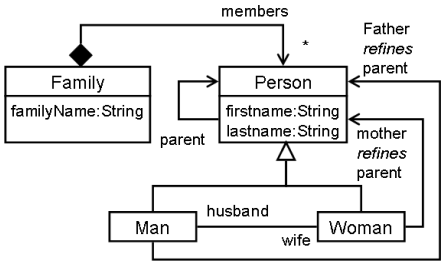


Figure 7.1: MOF metamodel of a pedigree

```
// Simple  entity definition
        entity(family);
        entity(person);
        entity(woman);
        entity(man);
 }
 entity(models);
}
```

Above we defined altogether seven entities in the pedigree domain: four entities for MOF classes (person, woman, man and family). The domain of pedigrees can be organized into a containment hierarchy by introducing additional entities (as equivalents for MOF packages). These compound entities (e.g. people, metamodel, models) may contain other entities as their content (as denoted by the pair of braces "{}").

REMARKS **Comments** in VIATRA-I follow the Java convention: one can use "//" for single-line comments, and "/* ... */" for multi-line comments.

Relations. **Relations** define binary edges between two model elements. Most typically, these model elements are entities, which impose a graph structure on VPM models. However, relations may lead between relations as well. For instance, a relation leading from a relation to a class is basically an equivalent of an association class in traditional MOF notation.

```
    relation(parent,person,person);
    relation(father,person,man);
    relation(mother,person,woman);
```

Relations are directed in the sense that they lead from a **source model element** to a **target model element**. Naturally, both source and target elements are required to be resolved to a valid element in the model space after parsing a VTML file, but the definition of an entity does not need to precede the definition of a relation which refers to it as source or target.

For instance, in case of the the husband relation above, its source model element is the woman entity and its target model element is the man entity.

```
// Default multiplicity: many_to_many
        relation(parent,person,person);

// Multiplicity: many_to_one
        relation(father,person,man);
        multiplicity(father, many_to_one);
        relation(mother,person,woman);
// Multiplicity definition
        multiplicity(mother, many_to_one);
```

Multiplicities of relations. Relations also have *multiplicities*, which impose a restrictions on the model structure. Allowed multiplicity kinds in VPM are

- **one_to_one**: for each source element at most one target element can be in relation, and for each target element at most one source element can be in relation for each instance of the relation,
- **one_to_many**: for each target element at most one source element can be in relation each instance of the relation,
- **many_to_one**: for each source element at most one target element can be in relation for each instance of the relation
- **many_to_many**: an arbitrary number of source and target elements can be in relation.

The default value of a multiplicity is many_to_many.

Aggregations in relations. Property *isAggregation* tells whether the given relation represents an aggregation in the metamodel. In case of an aggregation relation, an instance of the relation implies that the target element of the relation instance also contains the source element.

```
        relation(members, family, person);
        isAggregation(members, true);
```

Above we declare the members relation leading from entity family to entity person to be an aggregation. This means that if a concrete family (instance) is linked to a concrete person (instance) then this person should be contained by the family.

The default value of the *isAggregation* property is false.

REMARKS Note that an aggregation relation declares containment between

model elements on the *model (instance)* level, and not on the meta-level as in case of compound entities.

Inverse of relations. The *inverse* points to the inverse of the relation (if any). In a UML analogy, a relation can be considered as an association end, and thus the inverse of a relation denotes the other association end along an association.

```
entity(woman)
{
        relation(husband, woman, man);
}
entity(man)
{
        relation(wife, man, woman);
}
inverse(man.wife, woman.husband);
```

For instance, the inverse of the husband relation (leading from a woman to a man) is the wife relation (leading from a man to a woman).

Generalization / Inheritance. Generalization/Inheritance can be defined between two entities, two relations and two functions using the supertypeOf (or subtypeOf) construct. The semantics of VPM inheritance is in accordance with the traditional UML generalization mechanism, but it is also extended to relations (i.e. attributes and associations).

```
entity(people)
{
  entity(metamodel)
  {
    entity(person)
    {
        relation(parent,person,person);
        relation(father,person,man);
        relation(mother,person,woman);
// Generalization / Inheritance of relations
        supertypeOf(parent,father);
        supertypeOf(parent,mother);
    }
    entity(woman);
// Generalization / Inheritance of entities
    supertypeOf(person,woman);
    entity(man);
    supertypeOf(person,man);
  }
}
```

Fully qualified names. Entities are arranged into a strict containment hierarchy, which constitutes the VPM model space. Within

a container entity, each model element has a unique **local name**, but each model element also has a globally unique identifier which is called a **fully qualified name** (FQN). A fully qualified name is the concatenation of model element names according to the containment hierarchy using a dot (".") notation.

Since people is defined at the root of the model space in the example above, the fully qualified name of person is people.metamodel.person.

The fully qualified name of a relation is defined as the concatenation of the fully qualified name of its source model element and its local name [1]. For instance, the fully qualified name of mother is people.metamodel.person.mother.

Visibility of names. While fully qualified names uniquely identify model elements, their use can be inconvenient due to their lengthy names. In order to shorten such names, local names can also be used everywhere (i.e within the definition of any entity or relation) for the unique identification of model elements.

If a local name is used which is not found within the current entity (which is the scope / container of the definition where the name is used), the search continues in the container (parent) of the current entity as long as the local name can be resolved.

If a fully qualified name is used then it is first tried to be resolved as an **absolute fully qualified name**. If this fails, then the FQN is considered to be a **relative fully qualified name**, and a search is initiated from the current entity. If this fails, the search continues in the container of the current entity towards the root model element.

For instance, the following piece of code provides alternative definitions for relation father.

```
entity(people)
{
  entity(metamodel)
  {
    entity(person)
    {
// Absolute FQN: This definition can be used everywhere
// and it has the same effect each time
        relation(father,people.metamodel.person,
                        people.metamodel.man);
// Local: Neither man, nor person is found inside entity
// 'person', but they are found in the container entity
// of person (i.e. 'metamodel').
        relation(father1,person,man);
    }
```

[1] As a consequence of this rule, relations between relations are required to be acyclic, i.e. the transitive sources should evaluate to an entity sooner or later.

```
    entity(man);
// Local: Both man and person are found locally inside
// entity 'metamodel'.
    relation(father2,person,man);
  }
// Relative FQN: metamodel.person is not found as an
// absolute FQN, therefore it is searched as a relative
// FQN starting from the current entity people
  relation(father3,metamodel.person,metamodel.man);
// Local names cannot be used as search for local names
// proceeds outwards.
// Error:  relation(father4,person,man);
}
```

REMARKS

A VTML file may refer to model elements not defined in the same VTML file. This is due to the fact, that all element names are resolve with respect to (i) the defintions in the VTML file itself and (ii) an open VIATRA-I model space.

If you use external model elements, then it is a parsing error if this element is not present in the active model space.

It is a good modeling practice to explicitly import such elements by import declarations (see later in Sec. 7.1.2).

Built-in datatypes, Value of an Entity. VIATRA-I supports the manipulation of basic built-in datatypes, such as Strings, Booleans, Integers and Doubles. These built-in datatypes are also defined as entities within the model space inside the datatypes domain. Metamodels refer to these elements by fully qualified names.

```
        relation(familyname, family, datatypes.String);
```

For instance, the above code defines a familyname relation from entity family to the built-in string datatype (datatypes.String).

The instances of the built-in datatypes will be represented in an object-oriented way.

```
datatypes.Integer(i1) -> "2";
```

This means that a string **value** may be assigned to any entity using the → notation, and the conversion of this string value to built-in datatypes is resolved automatically according to the type information (e.g. the string value of i1 is converted into an Integer in the example.

7.1.2 Definition of (instance) models

After defining a metamodel for a certain domain, we may define (instance) models, that is type-conformant instances of the metamodel. Although in many cases, these models are created using VIATRA-I

importers, model creation can also be carried out using VTML in the same way as in case of defining metamodels. This is demonstrated in the following pedigree of Fig. 7.2.

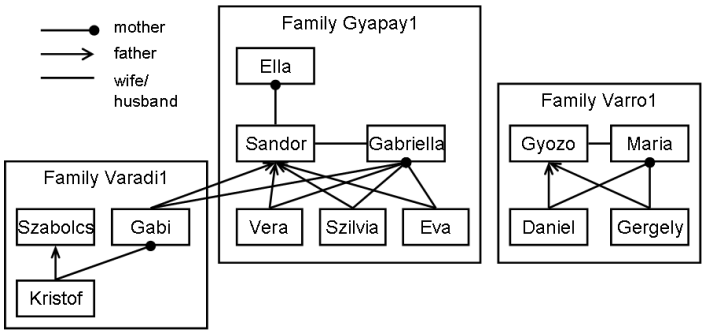


Figure 7.2: A sample pedigree model

Instance-of relationship in VIATRA models As the main conceptual difference between the UML/MOF modeling philosophy, the VPM model space does not distinguish between classes and objects: in VPM we only have a type entity and an instance entity related to each other by an **instance-of relationship**. In this sense, an entity may have a type in various domains (multiple classification), but circular typing, and similar peculiarities are disallowed, naturally.

```
entity(Daniel);
instanceOf(Daniel, people.metamodel.man);
relation(ln1, 'Daniel', str);
instanceOf(ln1, people.metamodel.person.lastname);
entity(str) -> "Varro";
instanceOf(str, datatypes.String);
```

For instance, Daniel is first defined above as a general entity, and then its type is constrained to man (from our previous domain people.metamodel). The notation for element names with and without apostrophes (') — such as Daniel, and 'Daniel' in the above code — are identical in VTML.

REMARKS Model elements with names starting with capital letters will require some precautions later on in transformation programs where they might interfere with the use of variables. A variable compulsorily start with a capital letter (Prolog-convention), thus in case of name clashes (with capital initials), names of model elements need to be surrounded with a pair of apostrophes ('), which is again a Prolog-convention.

Domain specific predicates and import declarations. For a more convenient use, type of a model element can also be specified by

using the model elements defined in the domain metamodel as fully qualified predicate names. Local names and relative FQNs can also be used as predicate names using the *import* construct.

```
import people.metamodel;

man(Daniel);
person.lastname(ln1, Daniel, str);
datatypes.String(str) -> "Varro";
```

For example, man(Daniel) is a valid definition for entity Daniel as there is an entity called man in the imported people.metamodel domain, which implies that unary predicate man can be used when defining instance models for that domain.

Furthermore, person.lastname is again a valid type reference in the definition person.lastname(ln1, Daniel, str), since (i) person is available as a local name after importing people.metamodel, and lastname is a relation in the metamodel with person as source entity.

Finally, datatypes.String(str) → "Varro" is a valid instance of the built-in datatype datatypes.String, which is accessed by its fully qualified name.

REMARKS We recommend to use the domain-specific notation wherever possible, which is much more legible. In fact, the generic notation with explicit *instance-of* relationship is mainly used in generic and meta-transformations.

Namespace declarations. In the previous example, entity Daniel is defined right below the root element of the model space, which is not a good practice as the model space will become rather chaotic after a while. Namespace declartions enable to override the default root element of the model space, and define a new scope (root) for VTML declarations.

```
namespace people.models;
import people.metamodel;

man(Daniel);
```

For instance, in the modified example above, entity Daniel is now placed inside the container entity of people.models.

7.1.3 Views of the model space

The VIATRA-I model space can be observed from two different viewpoints[2], namely, the **containment view** and the **graph view**, both

[2] Currently, only the tree view of the containment hierarchy is supported by the tool

depicted in a small sample model in Fig. 7.3.

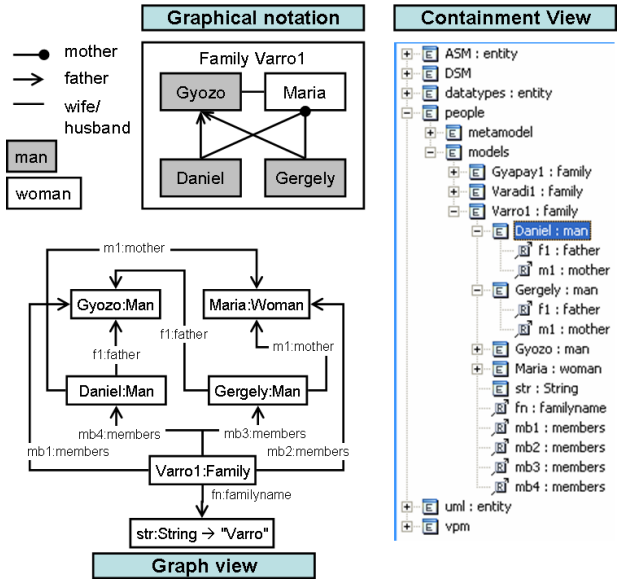


Figure 7.3: Conceptual views of the VIATRA-I model space

- The *containment view* is a tree view visualizing the containment hierarchy of entities in the way as defined by fully qualified names. Thus, relations are placed inside their source entities.
- The *graph view* presents the model space as a graph where entities are nodes and relations are edges[3].

7.2 Basic ASM Constructs

7.2.1 ASM Machines (Hello World)

Our first example is the traditional "Hello World" example, which prints "Hello Viatra World!" on the *Code Output View* console.

```
machine example1
{
  rule main() = print("Hello Viatra World!\n");
}
```

The most elementary behavioral concept in VIATRA-I is the **machine**. Each (GTASM) machine has a unique name (also organized in a strict containment hierarchy addressed by fully qualified names). After parsing the machine this will be the name of the entity that represents the internal model representation of the machine. After successful parsing, you can find your machine by this name, and run the associated interpreter on it.

[3] Since relations may lead between relations as well, formally, this is a hypergraph.

Each machine has a single **main ASM rule** that defines the rule which is first called when the transformation specified in the machine has been started.

The **print rule** is a built-in ASM rule which evaluates the expression passed as an argument using a Java-like notation and prints it to the *Code Output View*.

REMARKS It is recommended that each VTCL file contains a single machine.

7.2.2 ASM Rules: Seq rule, Random rule, Log rule

ASM machines consist of a set of **ASM rules**, where ASM rules provide us with the most traditional control structures of imperative programming languages.

```
machine example2
{
  rule main() = seq {
    print("first line\n");
    print("second line\n");
    // Random rule random rule will execute exactly one
    // of its subrules selected non-deterministically
    random {
      print("third line\n");
      print("3. line\n");
    }
    print("last line\n");
    log(info, "Transformation terminated successfully");
  }
}
```

The **sequential (seq) rule** executes all rules listed inside in the defined order. The **random rule** executes a non-deterministically chosen rule from the rules listed inside. The **log rule** prints a message into the Eclipse Error Log, which is either an info, a warning or an error message.

REMARKS The use of semicolons (;) at the end of VIATRA statements normally follows that Java convention, that is, semicolons are superfluous after closing braces (}) of complex rules, but they are compulsory otherwise after calling elementary rules.

7.2.3 ASM Variables, Let rule, Update rule

Now we discuss the definition and use of **ASM variables** on the following ASM machine.

The **let rule** defines a variable Var and initializes it with a term, and then calls its internal rule. Variable Var is accessible inside the scope (i.e. in the body) of let.

If variable Var accessed by the **update rule** is already defined then its value is updated to Term. Otherwise, if a variable is not defined previously, a compile-time error is reported.

```
machine example3
{
 rule main()= seq {
    // Variable definition by let rule
    let X = 0 in seq {
        // Update the value of X
        update X=5+6;
        // Print the value of a variable
        print("Value of X (Lev 1):" + X);
        // Print the value of a term
        print("Value of X+58 (Lev 1):" X+58);
        // Define a variable Y with one rule scope
        let Y=4*X in print("Value of Y (Lev 2):" + Y);
        // Value of Y is undefined after the block
        // Define again variable Y with one rule scope
        let Y=4*3 in seq {
                // Update variable X
                update X=Y+10;
                print("Value of Y (Lev 2):" + Y);
                print("Value of X (Lev 2):" + X);
           // Redefinition of variable Y is NOT allowed
           // let Y = 10 in print("Value of Y (Lev 3):" + Y);
        };
        // X preserves the value assigned inside
        // the previous let
        print("Value of X (Lev 1):" + X);
        // Value of Y is undefined here
    };
    // Value of X is undefined here
 }
}
```

When running the example above, the output of the transformation should be as follows.

```
Value of X (Lev 1): 11
Value of X+58 (Lev 1): 69
Value of Y (Lev 2): 44
Value of Y (Lev 2): 12
Value of X (Lev 2): 22
Value of X (Lev 1): 22
```

- The code above first defines variable X by using the **let rule** and initilizes it to 11.
- Then the term X+58 is evaluated and printed, which results in 69 appearing in the Code Output window.
- Next, we define a variable Y locally, within the scope of a **let rule** (let Y = 4*X in seq ...). As a result, the value of variable Y

is equal to 44. But as soon as we exit the scope of this internal (Level 2) let rule, variable Y becomes undefined.

- Variable Y is defined again within the scope of a second let rule (at Level 2) to be equal to 12. Then an update rule reassigns a value to variable X as Y+10, which is 22.
- Variable Y can NOT be redefined at the next level with another let construct.
- However, the value of variable X is kept even after the let rule at Level 2 has terminated since it was updated inside, and variable X within this scope.

Definition of variables. An ASM variable needs to be defined explicitly prior to its first use using the *let rule* (or *forall rule* and *choose rule*). Alternatively, they can also be passed as input parameters of ASM rules.

REMARKS ASM variables should start with capital (upper case) initials, and should always be defined prior to first use.

Variables are untyped. ASM variables are untyped, i.e. we cannot assign types at compile-time. Thus the type of a variable is induced at run-time, and this run-time type may change during execution, i.e. the same variable may store once an integer value and then a string, but this is not a good modeling practice.

Variables can contain constants of the following types: *string*, *integer*, *double*, *boolean*, and *model elements*.

Scope of variables. The **scope of an ASM variable** can be defined according to the place where the variable was defined first, thus it can be **rule-scope** or **block-scope**. ASM rules that define variables with block-scope are the *let rule*, the *forall rule* and the *choose rule*.

Each variable is visible and accessible anywhere inside its scope including sub-blocks at arbitrary depth, but it becomes undefined outside its scope (compare the use of variables X and Y in the previous example). As a result, ASM variables that are undefined within a certain scope cause compilation errors.

REMARKS ASM variables cannot be redefined within an internal scope.

7.2.4 ASM Expressions and Functions

ASM Constants and Expressions. ASM expressions are constructed in the traditional way from constants, variables and ASM functions (see later in this section). Currently, only constants have types (string, boolean, integer, double or fully qualified name) that are evaluated at compile-time, while the type of variables and ASM functions are determined dynamically at run-time.

The following example demonstrates the use of ASM constants.

```
rule main () =  seq
{
// String
  print("String:" + "A string");
// Integer
  print("Integer:" + 12);
// Double
  print("Double:" + 12.3);
// Boolean
  print("Boolean:" +true);
  print("Boolean:" +false);
// Multiplicity kind
  print("Multiplicity:" +many_to_many);
// Model elements
  print("Model element:" + people.metamodel.man);
  print("Model element:" + ref("people.metamodel.man"));
}
```

The final ref construct retrieves a model element constant "converted" from a string.

It is worth summarizing the notation of constants of different types.

- String constants are printed between quotation marks (").
- Integer constants are numerical values.
- Double constants are numerical values with a "." (floating point) separator.
- Boolean constants are true and false.
- Multiplicity kind values are one_to_one, one_to_many, many_to_one, many_to_many
- Name constants are alphanumerical literals starting with a letter. Fully qualified names are name constants concatenated by dots (".").
- undef: the undefined value

REMARKS Note that name constants starting with an upper-case initial may

rarely clash with variables in ASM programs. In case of such a conflict, the name constants have to be surrounded with apostrophes (').

ASM terms are untyped expressions constructed from ASM constants, variables and functions using traditional operators overviewed below.

- String operators: $<, >, +, ==, !=, <=, >=$
- Integer and Double operators: $<, >, +, -, ==, !=, *, /, <=, >=$
- Boolean operators: $||, \&\&, !, ==, !=$
- Multiplicity operators: $==, !=$
- Name operators $==, !=$

REMARKS If an expression is evaluated to a type where such a comparison operation is not supported (e.g. "str1" || "str2" is invalid), a runtime exception is thrown.

ASM functions. While ASM variables have a local scope in the sense that a variable becomes non-accessible as soon as the execution leaves the rule (or the block) of its scope, **ASM functions** have a global visibility, thus they are accessible from anywhere during the entire run of an ASM machine (but not afterwards as in case of models in the VPM model space). The use of ASM functions is demonstrated in the following example.

```
machine example4
{
 // asmfunction name/arity {(Optional) Initial assignments}
 asmfunction name2hair / 1 {("John")="brown";("Jill")="red";}
 rule main() = seq {
    // We can use the value of the slot in any expression
    print(name2hair("John"));
    // We can update the value in a slot.
    update name2hair("John")="white";
    // The ASM functions have a global scope
    let  X="Jack" in update name2hair(X)="purple";
    print(name2hair("Jill"));
    print(name2hair("Jack"));
    print(name2hair("John"));
    // Slots with no value set will contain the undef value
    print(name2hair("Alice"));
 }
}
```

The output of the previous ASM machine is as follows.

```
brown
red
purple
white
```

```
undef
```

In terms of structured programming languages, an **ASM function** is an associative array (dictionary, map, etc.), which stores **data values** at **locations (slots)** defined by its **index elements**. However, a main difference is that these arrays are very dynamic in the sense that they may store an arbitrarily large number of elements, and the size of the array can be increased at any time during execution in order to store a new element at a specific location (index). Unused locations of arrays are implicitly storing the *undef* value. Mathematically, an ASM function is a partial mapping from its index domain to its value domain. (called location or slot)

First, we have to declare an ASM function at the global (machine) scope (that is, outside any rules) prior to its first use by defining an an arity (dimension of indexes). In addition, we may initially provide partial definitions of some slots initially storing some values at certain locations

7.2.5 Rules calling other rules

Rule parameters. An ASM rule may contain **input** (in), **output** (out) and **inout parameters** (input/output) which are passed in and/or out when executing the rule as demonstrated by the following example.

```
machine example5
{
 rule main (in AName, in ModelElement) = seq {
   print(AName);
   print(ModelElement);
   let X = ref(ModelElement) in print(name(X));
 }

 // rule definition with in, and out parameter
 rule double(in X, out Y) =
        update Y=2*X;

 // rule definition with in, and out parameter
 rule square(inout X) =
        update X=X*X;

}
```

Here double(in X, out Y) doubles the value passed as the input parameter, while square(inout X) calculates the square of X and writes it back to X.

As a brief summary, an ASM rule defined in a machine is allowed to have (an arbitrary number of) parameters, where each parameter is either

- **input (in) parameter**: the value of an input variable will be equal to the value passed to the rule when it is called.
- **output (out) parameter**: when the execution of the rule terminates, the value of an output variable will be copied to the variable in the caller rule.
- **inout parameter**: when the value of this variable will be passed both in and out from the called rule.

Thus, each time an ASM rule is called, all of its input (and inout) parameters should be bound to constant values. As a result of the rule call, all the output and inout parameters of the rule are assigned new values and the execution is passed back to the caller rule. Naturally, the values of variables (calculated as output and inout parameters) are available in the caller.

REMARKS If the main rule of a machine has an input parameter then this parameter can set by the user when the transformation starts execution. The actual values of the input parameters should be separated by spaces ' ' in the pop-up window.

REMARKS Currently, each input parameter is considered to be a string.

Rule calls. As many rules can be defined within the same machine, VIATRA-I allows that ASM **rule calls** to each other (or recursively to the rule itself). Calculated values are passed between these rules by using *rule parameters* as demonstrated by the following example, where factorial(in N, out Fact) calculates the factorial of its input parameter N and returns it in the output parameter Fact.

```
machine example6
{
 // This ASM rule calls itself recursively
 rule factor (in N, out Fact) = seq {
        if(N==1) update Fact=N;
        else seq {
                call factor(N-1,Fact);
                update Fact=Fact*N;
             }
        }

 rule main() = seq {
   // Output variables used in the call to factor should
   // also be predefined e.g. by assigning the "undef" value
    let F = undef in seq {
             call factor (5, F);
             print("Factor(5, F):" + F);
    }
    let X = undef in seq {
         // Rules calling rules in  other machines
         call example5.double(3,X);
```

```
        print("X: "+ X);
        // Previous value of X is overwritten
        call example5.double(2,X);
        print("X: " + X);
        // X is an inout variable in square,
        // thus its input value is compulsory
        call example5.square(X);
        print("X: " + X);
        // The following code is not a run-time exception
        // as any arithmetic operation executed on undef
        // is undef
        let Y = undef in seq {
            call example5.square(Y);
            print("Y: " + Y);
        }
    }
 }
}
```

In order to facilitate modular transformation design, rules are allowed to call rules located in other machines. In such a case, the called rules can be referred to by their fully qualified names composed of the location of its machine container (example5 in the example) and the name of the ASM rule (e.g. square() and double()).

As a result of executing the ASM machine above, the following output is generated:

```
Factor(5, F): 120
X: 6
X: 4
X: 16
Y: undef
```

- First we calculate 5!, by calling factor(5, F);
- Then example5.double() is called twice with X as output variable. Thus, this variable is updated by the second call.
- Variable X is passed as an inout parameter for square, thus its input value is 4 (current value of X), and its output value is 16.
- Finally, note that arithmetic operations on the undefined value does not cause a run-time exception but the undef value is returned as result.

REMARKS Two rules may have identical names as long as their arity (the number of parameters) is different.

REMARKS Note that if an output parameter is also passed when calling the rule, then no exception is thrown by the interpreter: it is simply ignored. Obviously, the value of this output variable will be changed as the result of the rule call.

If the undef value is assigned to an input parameter, then it is passed to the rule execution in the normal way. But (in the very rare case) if the value of an input parameter is not defined then a run-time exception is thrown.

7.2.6 Advanced ASM control structures

In addition to user-defined rules, ASMs provide a predefined set of rules for advanced control structures such as choose, iterate, forall, if-then-else and try.

Their use is demonstrated by the following ASM machine. As the first step, we define an ASM function price and an ASM rule reduce_price.

```
machine example7
{
 asmfunction price / 1 {
   ("apple")= 5;
   ("peach")= 10;
   ("grape")= 15;
   ("malone")= 18;
   ("pineapple")= 20;
 }

 rule reduce_price(in X) =
   let P = price(X) in
     if (P - 5 > 0) seq {
           update price(X) = P - 5;
           print("Price of " + X + ":" + price(X));
         }
         else seq {
           print("Final price of " + X + ":" + price(X));
           fail;
         }
  rule main () = seq { ... }
}
```

Then the main rule is executed as follows.

If-then-else. The **if-then-else** rule evaluates a Boolean condition and then branches according to the result. In our example, we use an arithmetic Boolean condition testing the value stored in the ASM function price.

```
   if (price("peach") > 5)
         print("Peach is more expensive than 5");
   else
         print("Peach is less expensive than 5");
```

Try. The **try** rule attempts to execute its body rule, and executes the (optional) else part if the execution of the body rule fails. It is conceptually similar to exception handling in Java (but without

finally block). In the example, we try to reduce the price of grapes, and since it succeeds, the else branch is not executed.

```
   try call reduce_price("peach");
// The following line is optional in this case
   else print("Unable to reduce price");
```

Iterate. The **iterate** rule applies its body rule as long as possible, i.e. until its body fails. In our example, the price of a fruit is reduced by 5 as long as it is positive.

```
   iterate call reduce_price("malone");
```

Choose. The **choose** rule tries to find one substitution of variables defined in its head, which satisfies a Boolean condition, and then the body rule is executed. If more variable substitutions satisfy the condition, then one is chosen non-deterministically. If there are no such substitutions then the *choose* rule fails. The body rule may use to the head variables of the *choose* construct. In our case, we non-deterministically select a fruit which price is over 10.

```
   choose Y with (price(Y) > 10) do
          print("Chosen: Price of " + Y + ":" + price(Y));
```

Forall. The **forall** rule finds all substitution of variables defined in its head, which satisfies a Boolean condition, and then executes the body rule for each substitution separately. If no variable substitutions satisfy the condition, then the forall rule is still successful, but nothing is changed. The body rule may use to the head variables of the *forall* construct. In our case, we select all fruits which price is over 10.

```
   forall Y with (price(Y) > 10) do
          print("Price of " + Y + ":" + price(Y));
```

As a result, the following output is printed in the Code Output View:

```
Peach is more expensive than 5
Price of peach: 5
Price of malone: 13
Price of malone: 8
Price of malone: 3
Final price of malone: 3
Chosen: Price of pineapple: 20
Forall: Price of pineapple: 20
Forall: Price of grape: 15
```

REMARKS Note that in a typical model transformation, these control constructs will drive the execution of elementary graph transformation rules. In this respect, wherever a Boolean condition is expected, we may use a graph pattern as condition, and wherever an ASM rules is executed,

we may apply a GT rule. More details on graph patterns and graph transformation rules are provided in Sec. 7.3.

7.2.7 Model manipulation rules

Now we extend standard ASMs with rules for manipulating the VIATRA-I model space. These rules can be used directly in ASM programs; however, more typically, model manipulation is carried out by graph transformation rules.

New, Rename, SetValue, Move, SetFrom, SetTo. First we discuss how to create and alter model elements continuing the pedigree example of Sec. 7.1 (please recall Fig. 7.1 and 7.2).

Creation of entities and Renaming. In case of element creation (**new** rule), we can (optionally) specify the container where the new element is to be stored by using fully qualified names.

In case of the **rename** rule, the new name to be assigned to an existing model element is a local name, thus it is required to be unique within the container (in case of entities) or within the source element (in case of relations).

Such name clashes are resolved automatically by the VIATRA-I interpreter at run-time by renaming the new or renamed model elements that caused name clash.

As the first step, we create two families F1 and F2 inside people.models and rename them.

```
namespace people;
import people.metamodel;

machine model_manipulation_create
{
 rule main () =
// Definition of variables used later
 let F1 = undef, F2 = undef, M = undef, W = undef
     FNew = undef, FNew2 = undef in seq {
// Creation of new entities
  new (family(F1) in people.models);
  new (family(F2) in ref("people.models"));
// Renaming of model elements
  rename(F1, "Family1");
  rename(F2, "Family2");
  ...
 }
}
```

Creation of built-in relationships. Then we create a man M (and set the name and the value of this entity to Peter) and a woman W (renamed as Rita). Then we demonstrate how the new rule can be applied for built-in relationship (such as instanceOf).

```
  ...
  new (man(M) in F1);
  rename(M, "Peter");
  setValue(M, "Peter");

// Generic creation
  new (entity(W));
  rename(W, "Rita");
  new (instanceOf(W, people.metamodel.woman));
  ...
```

Note the difference between the two creations:

- Peter is created inside family F1, and its type is assigned right at creation;
- Rita is created in the root of the model space (as no container is specified) while its type is only assigned by an explicit creation of the instanceOf relation between W and people.metamodel.woman.

REMARKS The explicit creation of other built-in relationships (instanceOf, typeOf, supertypeOf, subtypeOf and contains) can be handled similarly.

Creation of relations (and implicit relocation of entities). While the creation of a relation is specified in the same way as in case of entities (and built-in relations), it is crucial to point out that the creation of relations has to be carried out consistently with move operations. The aggregation parameter of a relation is only checked, but no implicit moves are carried out, which may result in temporarily inconsistent models.

Deletion of model elements. The deletion of model elements can be carried out by a single delete rule, which takes (the fully qualified name of) a model element as parameter.

The following program removes all the entities and relations created previously by machine model_manipulation_create.

```
machine model_manipulation_delete
{
 pattern persons(X) =
 {
   person(X);
 }
```

```
rule main () =
let
  F1 = people.models.Family1, F2 = people.models.Family2,
  F3 = people.Family1, F4 = people.Family2 in
seq
{
 forall X below F1 with find persons(X) do delete(X);
 delete (F1);

 choose X in F2 with find persons(X) do delete(X);
 delete (F2);

 forall X below F3 with find persons(X) do delete(X);
 delete (F3);

 forall X in F4 with find persons(X) do delete(X);
 delete (F4);
}
}
```

7.3 Graph patterns and pattern matching

In VIATRA-I , complex logical conditions on (some parts of) the model space can be expressed by using the powerful means of **graph patterns**.

A *graph pattern* expresses that there should be a part of the model space, which "resembles" to the graph pattern it self. This "resemblance", when all the (graph) elements in a pattern are tried to be mapped to elements in the model space, is called *graph pattern matching*.

7.3.1 Definition of simple graph patterns

A **graph pattern** expresses a complex (derived) structural condition (or constraint) on model elements captured by a prototypical instance model.

A graph pattern is handled as a (user-defined) Boolean predicate with variables. However, unlike ASM variables, we do not need to declare in advance if a variable in a pattern is input or output. In other terms, if a pattern variable is bound when the pattern is matched, it is handled as an input variable, otherwise, if the variable is unbound then it is handled as an (existentially quantified) output variable, which gets instantiated when the pattern is matched.

The definition of a simple graph pattern expressing the brother relationship in the pedigree domain of Sec. 7.1 can be observed in the following example (also depicted in a graphical way in Fig. 7.4).

```
machine graph_patterns
{
 // B is a brother of X
 pattern brother(X, B) =
 {
        person(X);
        person.parent(P1, X, P);
        person(P);
        person.parent(P2, B, P);
        man(B);
        check (X != B)
 }
    ...
}
```

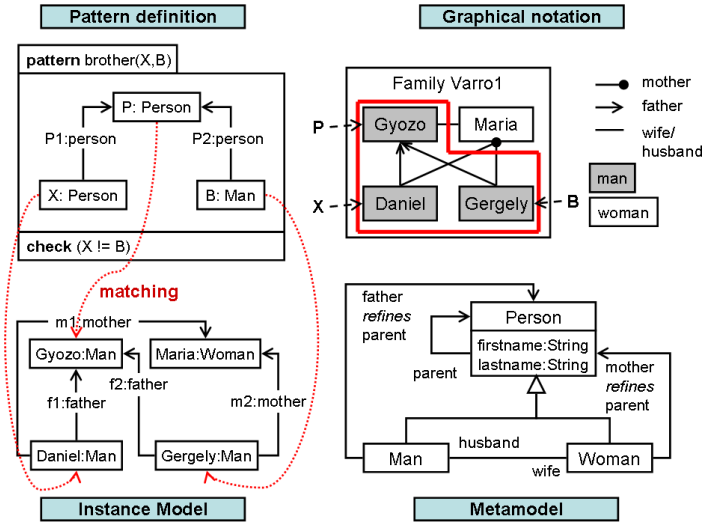


Figure 7.4: Patterns and graph pattern matching

The pattern expresses that in order to find a brother B of a person X, one needs to find a person X, and a man B who have a common parent P as expressed by the parent relation. Furthermore, B and X is not allowed to be the same persons (i.e. nobody is a brother of himself) as expressed by the logical condition of the pattern (after the check keyword).

Note that the family model in Sec. 7.1.2 contained only father and mother relations between different persons. However, since parent is a generalization (supertype) of both father and mother, it is sufficient to use this more general parent term in order to obtain a more succinct pattern. The same argument also holds when searching for the more general person instead of a man or a woman.

7.3.2 Graph pattern matching

When a pattern is used in an ASM program, **graph pattern matching** is performed, which tries to bind all variables of a pattern to element from the model space which satisfy type and structural consistency constraints. A corresponding variable substitution (in a certain part of the model space) is called a **matching**.

- **Type consistency** means that the type of a model element should be identically typed or a subtype of the pattern element it is matched to.
- **Structural consistency** means that if a pattern relation (edge) is matched to a model relation, then the source and target elements of this model relation need to be matched successfully to the source and target elements of the pattern relation (edge) in accordance with the *graph view* of the model space.

In addition, further constraints in a graph pattern may impose a certain inheritance, containment and instance-of restrictions on entities and relations in the model space.

By default, the free variables in the pattern will be substituted by *existential quantification*, which means that only one (non-deterministically selected) matching will be obtained. This can be overruled by parallel pattern matching using the forall construct (to be discussed later).

For instance, a matching of pattern brother inside model (entity) people.models is X = 'Daniel' and B = 'Gergely' (i.e. Gergely is a brother of Daniel — see also Fig. 7.4). In fact, there is a symmetric matching of the same pattern in the same model, namely, X = 'Gergely' and B = 'Daniel' (Daniel is a brother of Gergely).

REMARKS **Side effects of pattern matching**: There are no side effects of pattern matching other than instantiating the variables in the head of the pattern (e.g. X and B in brother(X, B)).

REMARKS **Attribute conditions:** The check keyword refers to a logical condition which should be satisfied in addition to the successful matching of the pattern. Typically, arithmetic conditions are checked in the context of the matched model elements this way.

REMARKS **Injectivity of pattern matching**: Currently, each variable of in a pattern should be matched to different model elements during pattern matching (in technical terms, the pattern matching is injective).

Use of graph patterns in ASM programs. A graph pattern can be used in ASM programs exactly where a Boolean expression is expected using the find keyword as demonstrated by the following example.

```
rule main () =
   choose X below people.models, B below people.models
      with find brother(X, B) do
      print(name(X) + "->" + name(B));
```

Here we non-deterministically select an assignment for X and B from the subtree of the model space starting from the container entity people.models, which satisfies the pattern brother(X,B), and then print the names of both persons.

Parallel pattern matching. Patterns most frequently are used together with the *forall* construct, which processes all matchings of a pattern in parallel. Note that while matchings are not required to be completely disjoint (i.e. certain model elements may overlap), it is forbidden that such parallel actions contradict with each other (e.g. one removes a model element while the other preserves it).

```
rule main () =
   forall X below people.models, B below people.models
      with find brother(X, B) do
      print(name(X) + "->" + name(B));
```

The above piece of VTCL code prints both symmetric pairs of X and B, namely:

```
Daniel -> Gergely
Gergely -> Daniel
```

Use of pattern parameters. The parameters of graph patterns are different from that of ASM rules in the sense that the parameters of a graph pattern do not have directions (in, out, in/out). In fact, the direction of a pattern parameter is only determined at execution time when pattern matching is initiated for that pattern. In this way, the same pattern can be called with different settings for parameter directions at various parts of a VTCL program.

In order to seamlessly interact with ASM rules, non-deterministic variable assignments can be carried out by pattern matching using the choose construct. All pattern variables which are intended to be used in the do part of the choose rule should be declared within the choose rule (or before) as demonstrated by the following example.

```
rule main () =
   choose X below people.models
      with find brother(X, B) do
      print(name(X) + " has a brother");
```

In this example, we only declare variable X to be used as an ASM variable. As a consequence, a model element gets assigned to variable B when matching pattern brother(X,B), but only the predeclared variable X can be used within the do block.

Obviously, input parameters can also be passed to the pattern matching process as seen in the example below.

```
rule main () =
  let B = people.models.'Varro1'.'Daniel' in
   choose X below people.models
      with find brother(X, B) do
      print(name(X) + " has a brother called Daniel");
```

7.3.3 Scope of pattern matching

Scope of pattern variables: Below vs. In. The scope of a variable to be instantiated by pattern matching can be restricted to take values only from a subtree of the model space (**below scope**), or only from the direct children of a certain entity (**in scope**).

For instance, the following piece of VTCL code fails to find any brothers in our example model space of Sec. 7.1.2 as all persons are direct children of a family entity.

```
rule main () =
   choose X in people.models, B in people.models
      with find brother(X, B) do
      print(name(X) + "->" + name(B));
```

Here people.models is called the **scope entity**.

REMARKS From a pure specification point of view, one can safely use the *below* scope instead of the *in* scope as all the direct children of the scope entity. However, the pattern matching process can be significantly faster if one is more (but not too) restrictive concerning the scope.

Global scope for pattern matching. As an alternate solution, we may restrict the scope of pattern matching globally, i.e. for each element matched within a pattern in the following way.

```
rule main () =
   choose X, B
      with find brother(X, B) below people.models do
      print(name(X) + "->" + name(B));
```

In this solution, the scope variables X and B becomes people.models since the scope is specified globally when initiating the pattern matching (find pattern below scope). However, a main difference is that also the internal variables of the pattern have a scope restriction (i.e. variable P in case of pattern brother) unlike in the case of local scoping of the ASM variables themselves.

Patterns spanning across multiple domains (models). Graph patterns are not restricted to take values from a single subtree of the model space. In fact, in case of model transformations, which are the most typical applications of the VIATRA-I framework, models are frequently taken from models of different modeling languages. Such scope restrictions can be specified in the pattern as well, and not only in the caller of that pattern.

In the example below, we search for cousins in two families.

```
// X of family F1 is a cousin of Y in family F2
pattern cousin(X, F1, Y, F2) =
{
        family(F1);
        family(F2);
        person(X) in F1;
        person.parent(Par1, X, P1);
        person(P1);
        person.parent(Par2, P1, GP);
        person(GP);
        person.parent(Par3, P2, GP);
        person(P2);
        person.parent(Par4, Y, P2);
        person(Y) in F2;
        check (F1 != F2)
}
```

Here we specified that X is only considered to be a cousin of Y if they have a common grandparent GP, but they are not in the same family. This is prescribed by constraining the scope of person(X) to be a subentity of family F1 (and person(Y) likewise for family F2).

7.3.4 Negative, Recursive and OR-patterns

VIATRA-I supports the use of some advanced constructs in graph patterns such as e.g. negative, recursive and OR-patterns, which will be demonstrated in the sequel.

Patterns calling other patterns. In order to facilitate the reuse of graph patterns, patterns may initiate the matching of other patterns in turn using the find keyword. For instance, a cousin pattern could also be defined by matching a grandparent pattern twice. As a result, we obtain a much higher level of reuse in transformation design.

```
// GP is a grandparent of Z
pattern grandparent(Z, GP) =
{
        person(Z);
        person.parent(Par1, Z, P1);
        person(P1);
        person.parent(Par2, P1, GP);
```

```
        person(GP);
}

// X of family F1 is a cousin of Y in family F2
pattern cousin(X, F1, Y, F2) =
{
        family(F1);
        family(F2);
        person(X) in F1;
        find grandparent(X, GP)
        person(GP);
        find grandparent(Y, GP)
        person(Y) in F2;
        check (F1 != F2)
}
```

Here person GP should be a common grandparent of both X and Y, thus a joint matching satisfying both grandparent(X,GP) and grandparent(Y,GP) needs to be found.

REMARKS Patterns may also refer to patterns defined in other machines by using its fully qualified name.

Negative patterns. A pattern may restrict its own applicability by specifying negative application conditions by negative patterns. A negative pattern is an ordinary pattern preceded with the neg keyword.

The semantics of negative patterns prescribes that if a matching of a pattern is found, and this matching can be extended to a matching of (one of) its negative patterns then the original matching of the pattern is rejected, and the pattern matching process continues to find a new variable assignment.

Unsurprisingly, an existing pattern can also be called as a negative pattern using the find construct. Furthermore, negative patterns may also contain negative patterns in turn to yield a pattern language equivalent with first order logic.

In the example below, we define patterns for an only child (has neither brother nor sisters) and an orphan (who has no parents).

```
//  Z is an only child
 pattern onlyChild(Z) =
 {
    person(Z);
// Calling predefined patterns as negative patterns
    neg find brother(Z, B);
    neg find sister(Z, S);
 }

 // X is an orphan
```

```
pattern orphan(X) =
{
    person(X);
// Defining a local negative pattern,
// which is not accessible from outside
    neg pattern hasParent(X) =
    {
        person(X);
        parent(Par, X, P);
        person(P);
    }
}
```

REMARKS Each pattern itself has to be a well-formed graph, i.e. (i) all source and target nodes of relations, and (ii) parameters of called patterns have to be included explicitly in the local pattern.

OR-patterns. In many cases, the notion of a single pattern covers alternative situations. OR-patterns provide a primary means to uniformly capture such different patterns under a single name.

The following piece of VTCL code demonstrates the use of OR-patterns by defining the notion of a sister in a slightly different way compared to brother (in Sec. 7.3.1).

```
// S is a sister of S
pattern sister(X, S) =
{
        person(X);
        person.father(P1, X, F);
        man(F);
        person.father(P2, S, F);
        woman(S);
        check (X != S)
}
or
{
        person(X);
        person.mother(P1, X, M);
        woman(M);
        person.mother(P2, S, M);
        woman(S);
        check (X != S)
}
```

This way, a sister of a person X is a woman S, if there is a common father F of X and S or if there is a common mother M of X and S. As a consequence of using OR-patterns, both cases can be handled identically when aiming to find a matching of sister(X,S).

REMARKS While subpatterns of an OR-pattern are evaluated in the given order, it is still a good practice for transformation engineering to create

mutually exclusive subpatterns so that the same matching only fulfills one subpattern.

Visibility and parameter passing in patterns. Each graph pattern is local in the sense that it can directly access locally defined elements only (those in one subpattern in case of OR-patterns). This means that variables can be named identically in different patterns (or different subpatterns of an OR-pattern).

In case of pattern calls, local pattern variables are passed as parameters for the called pattern in a bidirectional way, i.e. the pattern matcher engine imposes no restrictions whether a variable is already substituted or not when the other pattern is called.

Since negative patterns are treated as special pattern calls, parameter passing is also very similar. The main difference is that parameter passing is unidirectional, i.e. the (external) positive pattern may pass parameters to the (internal) negative pattern. Furthermore, in the case when a negative pattern is defined inside a pattern and not just called by find (compare patterns orphan and onlyChild), common elements in the positive and the negative pattern share the same names, but they are still coupled by parameter passing from the positive to the negative pattern.

Recursive patterns. By combining pattern calls and OR-patterns, we can specify recursive patterns, which are true generalizations of traditional path expressions used in many graph transformation tools. A recursive pattern is also very similar to a recursive Prolog-program (being evaluated over a set of dynamic predicates describing the model).

The following recursive pattern defines the transitive descendant pattern.

```
// D is a descendant of X
pattern descendants(X, D) =
{
	person(X);
	person.parent(P1, D, X);
	person(D);
}
or
{
	person(X);
	person.parent(P2, Ch, X);
	person(Ch);
	find descendants(Ch, D)
	person(D);
}
```

As defined by this pattern, person D is a descendant of person X, if D is a child of X or if there is a person Ch who is a child of X, and D is a descendant of Ch.

7.4 Graph Transformation Rules

Graph transformation (GT) is the primary means for elementary model transformation steps in VIATRA-I . Graph transformation provides a rule and pattern-based manipulation of graph-based models. The application of a GT rule on a given VPM model (i.e. part of the model space) replaces an image of its precondition (left-hand side, LHS) pattern with an image of its postcondition (right-hand side, RHS) pattern, and additional actions can be executed after that as further side effects.

7.4.1 Definition of graph transformation rules

VIATRA-I provides different ways for defining graph transformation rules in order to adapt to the different programming style of transformation developers.

The traditional way of defining GT rules is to provide a pair of graph patterns: the precondition (left-hand side, LHS) pattern and the postcondition (right-hand side, RHS) pattern. In VIATRA-I , there is also an action part which defines additional side-effects for rules

The following example defines a GT rule for marriage, when a man and a woman founds a new family.

```
// Man M and Woman W get married to found a new family F
gtrule marry (in M, in W, out F) =
{
  precondition ...
  postcondition ...
  action ...
}
```

Precondition The precondition part is defined by a graph pattern which should be found in the instance model by pattern matching in order to enable the application of the GT rule.

In the sample marry rule, we prescribe the presence of a man M and a woman W who are (i) not married yet (see the find pattern calls as negative application conditions) and (ii) living in different families (F1 and F2, respectively).

```
precondition pattern lhs(M, W, F1, MB1, F2, MB2) =
{
      man(M);
      family(F1);
```

```
        family.members(MB1, F1, M);
        woman(W);
        family(F2);
        family.members(MB2, F2, W);
        neg find married(M)
        neg find married(W)
// This check is not required!
        check (F1 != F2)
  }
```

REMARKS Note that the check condition where additional logical conditions can be prescribed is superfluous this time, since VIATRA-I applies injective pattern matching policies, thus F1 and F2 are matched automatically to different model elements.

REMARKS Note that precondition and postcondition (as well as negative conditions) of a GT rule can also be defined using the predefined pattern by the find construct.

Postcondition The postcondition pattern describes what conditions should hold as result of applying the GT rule. The result of GT rule application is calculated as the difference of the postcondition and precondition in the following way:

```
gtrule marry (in M, in W, out F) = {
 precondition pattern lhs(M, W, F1, MB1, F2, MB2) = {...}
 postcondition pattern rhs(M, W, F1, MB1, F2, MB2, F) =
 {
        man(M);
        family(F1);
        woman(W);
        family(F2);
        family(F);
        family.members(MB3, F, M);
        family.members(MB4, F, W);
        man.wife(WF1, M, W);
 }
}
```

Parameter passing. Matchings are passed to the postcondition as parameters, thus a parameter of the precondition pattern can be (but not compulsory to be) used in the postcondition pattern.

- All the parameters of the postcondition which are (i) shared by the precondition or (ii) by an input parameter of the entire GT rule are treated as **input parameters** for the postcondition. These parameters are already bound before calculating the effects of the postcondition.

  In the example above, input parameters of the postcondition are derived either from the GT rule parameters (such as M and

W) or the precondition (e.g. F1, F2, MB1 or MB2).

- Additional parameters of the postcondition are **output parameters**, which will be bound as the direct effect of the postcondition.

  The single output parameter of the postcondition is F.

The postcondition may prescribe three different operations on the model space.

- *Preservation.* If an input parameter of the postcondition also appears in the pattern itself, then the matching model element is preserved.

  Model elements matched by variables M, W, F1, and F2 in the example above are thus preserved.

- *Deletion.* If an input parameter of the postcondition does not appear in the postcondition pattern itself then the matching model element is deleted.

  Model elements matched by variables MB1 and MB2 are thus deleted.

- Creation. If a variable which appears in the postcondition pattern itself is not an input parameter of the postcondition, then a new model element is created, and the variable is bound to this new model element. Naturally, this variable can be used as an output parameter of the postcondition.

  In our example above, variable F is not an input parameter, thus it prescribes the creation of a new family F in accordance with the postcondition pattern. This F is an output parameter of both the postcondition and also the GT rule itself.

Actions After resolving the difference of the precondition and postcondition patterns, the GT rule may execute a sequence of additional actions defined as ordinary ASM rules.

As the postcondition pattern is not compulsory in a GT rule, we can define the effects of a GT rule directly in the action part as well as done in the following example.

```
action
{
   delete(MB1);
   delete(MB2);
   // Family F is created in the root of the model space
   new(family(F));
   // Same effects defined with different notation
   new(family.members(MB3, F, M));
```

```
    new(people.metamodel.family.members(MB4, F, W));
    new(man.wife(WF1, M, W));
}
```

REMARKS Note that all parameters of both the precondition and the postcondition can be used in the action part, but internal pattern variables (i.e. those that does not appear as parameter of the pattern) are not visible.

7.4.2 Calling GT rules from ASM programs

Graph transformation rules can be invoked by using the choose and forall ASM constructs as demonstrated in the following examples.

First, rule marry is applied for the marriage of Szilvia and Daniel.

```
rule main () =  seq {
 let X = ref("people.models.Varro1.Daniel") in
  let Y = ref("people.models.Gyapay1.Szilvia") in
// Variable definition is required for all variables
   let F = undef in
// Marriage of Daniel and Szilvia, new family F is created
     choose with apply marry(X, Y, F) do
     seq {
       rename(F, "Varro2");
       move(F, people.models);
     }
}
```

Syntactically, the GT rule is called in the with part of the choose (and forall), which seems to introduce logical conditions (i.e. pattern matching) with side effects at first sight.

However, on the semantic level, the application of the GT rule starts with an ordinary find pattern call for the precondition (see pattern matching in Sec. 7.3.2), which is directly followed by the actual GT rule application (i.e. resolution of the postcondition and the execution of the action part) in an atomic step prior to executing the do part of the choose (forall) construct.

Thus our example, we first apply rule marry in an atomic step, and only after that do we start executing the sequence of rename and move rules.

REMARKS When applying a GT rule using the choose and forall constructs, scoped variable declarations (e.g. X below M) are allowed there as well (none of such are present in the example above).

Iterate vs. forall. In model transformations, we frequently use the iterate and forall constructs to initiate the multiple application of a

GT rule. Note, however, that the choice between these two constructs in the context of GT rules have some semantic consequences.

The iterate construct combined with a single choose execution of a GT rule (as its body) applies the GT rule as long as possible, i.e. as long as a matching of the GT rule can be found by the choose construct. In other terms, first we apply the GT rule on a single (non-deterministically selected) matching and only after rule application do we select the next available matching.

As a consequence, the incorrect precondition of the GT rule may cause *non-termination* when combined with the iterate construct. For instance, the most typical problem is when a transformation designer does not prevent to apply the GT rule twice on the same matching (by using an appropriate negative condition).

```
rule main () =
 let F2 = undef in
 // Random marriage of unmarried people
 iterate
  choose M below people.models, W below people.models
    with apply marry(M, W, F2) do move(F2, people.models);
```

On the other hand, non-termination is normally not a critical issue when using the forall construct, which first collects all available matchings, and then applies the rule for all of them in a single deterministic (parallel) step.

REMARKS Note that the forall construct always succeeds, even if no matchings were found. Thus it directly causes non-termination if we switch to forall instead of choose in the example above.

However, if different matchings of a GT rule are overlapping, parallel rule application may result in conflicts when conflicting operations are issued on a model element (e.g. delete vs. preserve). This is demonstrated by the following example:

```
// The following call is not erroneous since one
// application of marry may disable other matchings
// of the same rule (rule conflicts)
  forall M below people.models, W below people.models
     with apply marry(M, W, F2) do move(F2, people.models);
```

Here there are multiple wife (husband) candidates (matchings) for each man (woman), but these matchings are conflicting since the negative condition of rule marry prevent the rule from being applied to already married couples. Thus if we marry someone along one matching, other matchings are invalidated.

REMARKS VIATRA-I currently does not provide support for detecting such con-

flicts, thus it is the role of the transformation designer to assure that a GT rule applied in forall mode is not conflicting with itself.

8 Sample Transformation: The Object-Relational Mapping

This chapter discusses a classic transformation task, the problem of object-relational mapping: how to transform a UML class diagram into a relational database schema. The example will only partially elaborate this task as our intention is to demonstrate the workings of VIATRA-I on a simple example, not to give a complete and production-ready transformation. Nevertheless, this transformation has practical significance as it is altogether not uncommon to use object oriented modeling in the initial design phase of databases.

8.1 Scope of the Chapter

In this chapter we demonstrate a whole transformation development life cycle starting with the theoretical formulation of the problem and reaching a transformation ready for full automation (see 8.1 for the intended functionality on a specific input-output pair). On the technical side, we first develop the metamodels describing the source and the target domain; in addition, we also define a trace metamodel for recording mappings between the domains. Second, we develop a set of transformations realizing the tasks. Accordingly, the following files constitute the example:

- Metamodels:
  - `uml2.vtml`: the metamodel of UML 2.0
  - `reldb_meta.vtml`: a simplified metamodel of relational database structures
  - `ref_uml2reldb_meta.vtml`: a reference (also commonly called trace) metamodel for recording mappings between the two domains
- Transformations:
  - `uml2reldb_xform.vtcl`: a transformation realizing the mapping from the UML domain to the relational domain
  - `orderingOfCols.vtcl`: an auxiliary transformation
  - `modelManagement.vtcl`: an auxiliary transformation
  - `codegen.vtcl`: SQL code generator transformation

Full code listings can be found in the Appendix, here we will discuss only the conceptually important parts. For instructions on how to process the metamodeling and control language files and how to run transformations please refer to the technical introductory part of this guide.

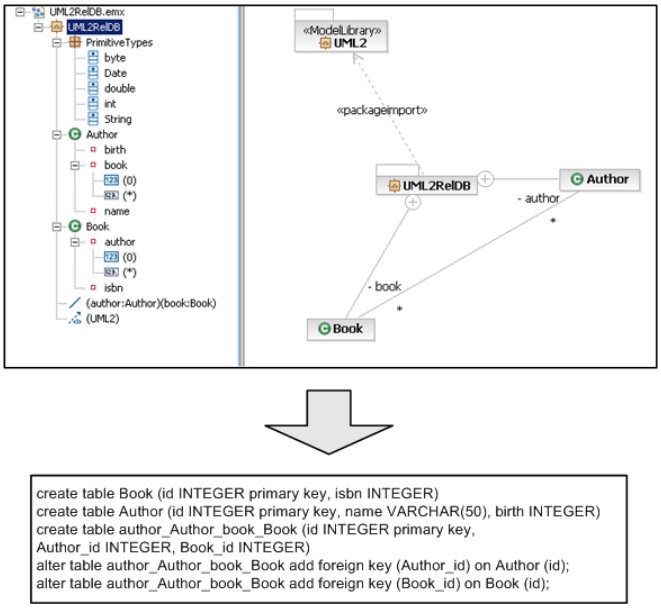


Figure 8.1: The object-relational mapping task

8.2 Theoretical Considerations

The main 'theoretical' question is that how can we map the concepts of object oriented modeling to relational modeling. In the following we summarize the main ideas commonly used in the literature and in the industrial practice.

1. *UML classes* are mapped to tables (from a more theoretical point of view, object types are mapped to relation types).

2. *Object structure.* If classes are mapped to tables, then it is straightforward that atomic attributes are mapped to simple relational attributes. (We omit the handling of list- and set-valued attributes.)

3. *Object identity.* In object-oriented modeling we assume that all objects have a unique identity. However, this is not captured explicitly in the model (on the conceptual level – implementations certainly use object identifiers), therefore the identity of an object mapped into a table row needs to be assured by a so-called surrogate key (an artificially established primary key). This is

crucial for establishing foreign keys for the mapping of object associations.

4. *Associations.* The most natural way to map (undirected) associations into a relational model is to use association tables - a relation type reflecting the binary relation between object pairs. Hence, the respective association table of an association type will have as columns a surrogate key and two foreign key columns to identify the according two rows in the class tables. Note that this is not the only solution - however, it is one of the simplest ones.

5. *Inheritance.* The relational model does not contain any notion of inheritance or subtyping. Therefore there are multiple possible solutions to mimic inheritance; each with its own quirks and constraints. As this is only a simple example, we refrain from handling inheritance.

6. *Information hiding, methods and other concepts of object-oriented modeling.* There are concepts and notions in object-oriented modelling – or specifically, in UML – that simply can not be expressed naturally in relational modeling. This is a fairly trivial consequence of the discrepancy of the domains intended to be modeled by the two paradigms; certainly there are some special cases where problem-specific solutidirectons can be given (for instance one can map object methods to database stored procedures under special sets of constraints), but generally when discussing the transformation task at hand we restrict its scope to pure information modeling without encapsulation.

8.3 Metamodels

Two different modeling domains play a direct role in the transformation: UML for the input models and the domain of relational databases for the 'result' models. In addition it is usual to maintain a model of the mappings established by the executed transformations, a so-called trace model. These three modeling languages have to be defined in VIATRA via their metamodels before we can begin elaborating the transformations themselves. Due to its size and complexity, here we refrain from discussing the UML2 metamodel – in fact, the usual way to use it is beginning the development with a model space already containing it.

8.3.1 The Relational Database Metamodel

The relational database metamodel (`reldb_meta.vtml`) is quite simple - or better to say, simplified; it defines tables, columns and for-

eign keys, all of which are 'DBElements'. The relations defined between them are self-explanatory and reflect our usual concept of relational databases. To note is the containment hierarchy defined in the file: the entity `reldb` will appear as a root-child level element in the model space, containing the entities `metamodel`, `models` and `transformations`. Arranging the model space structure for a newly defined domain this way is certainly by no means compulsory or the only possible way – it is just a sensible solution. For a visual representation of the `reldb` namespace see 8.2.

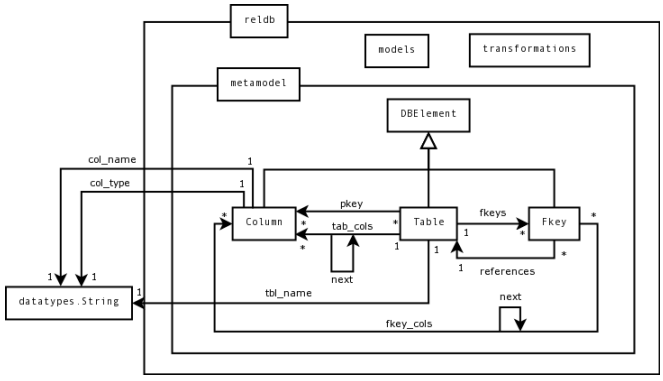


Figure 8.2: The relational database schema metamodel

8.3.2 The Trace Metamodel

The `uml2reldb` metamodel (`ref_uml2reldb_meta.vtml`) defines a metamodel for models that can capture 'mapped to' relations between model elements in the two domains. Technically, this means special 'mapping' entities - for example, `class2table` - in relation (in the VPM sense) with one `uml2.metamodel.Element` (for example, `uml2.metamodel.Class`) and one `reldb.metamodel.DBElement` (for example, `reldb.metamodel.Table`). For a visual representation of the `uml2reldb` namespace see 8.3.

8.4 Transformations

We define four ASM machines: `ordering`, `modelManagement`, `uml2reldb_xform` and `codegen`. The machine `uml2reldb_xform` contains the transformation for mapping a UML model in the model space to the relational domain and recording the mappings; the machine `codegen` generates code from a given database model in the relational domain, thus it is actually independent from the fact that we reach the relational model via a transformation from UML. The machines `ordering` and `modelManagement` have auxiliary functions.

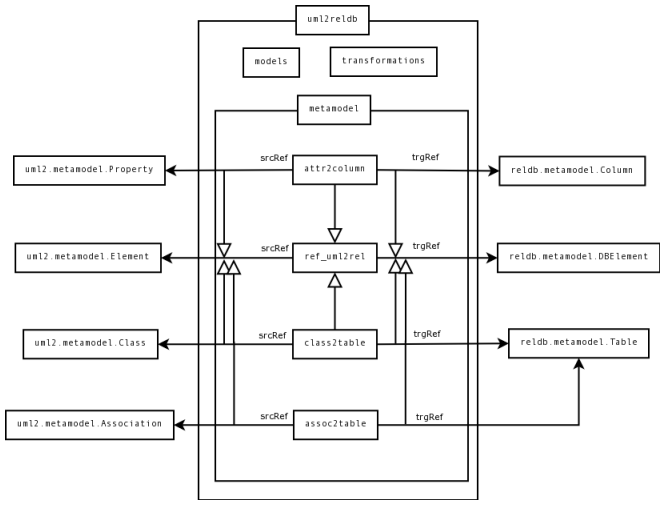


Figure 8.3: The trace metamodel

8.4.1 The Object-Relational Mapping

The file `uml2reldb_xform.vtcl` defines the ASM machine `uml2reldb_xform` for the actual transformation. The body of the main rule is as follows:

```
//main rule of the machine uml2reldb_xform
rule main (in UMLStr, in RefStr, in DBStr) = seq
{
  call initModels(UMLStr, RefStr, DBStr);                    //(1.1)
  forall C below models("uml") with apply class2tableR(C) do skip;  //(1.2)
  forall C below models("uml"), A below models("uml")
                        with apply attr2columnR(C, A) do skip; //(1.3)
  forall A below models("uml") with apply attrOfStringTypeR(A) do skip; //(1.4)
  forall A below models("uml") with apply attrOfIntTypeR(A) do skip; //(1.5)
  forall A below models("uml") with apply assoc2tableR(A) do skip; //(1.6)
}
```

Input parameters are:

- the name of the UML class model (*inside* VIATRA – i.e. it has been already imported from file or constructed in the VPM model space by other means);
- the name of the 'reference' model to be created, and
- the name of the database model to be created.

The call `initModels` (1.1) checks for the existence of the UML model under the entity `uml2` and creates the according entities under `reldb.models` and `uml2reldb.models` in the model space. These

three entities are placed in the indexed global array `models` with indices `uml`, `db` and `ref`. These indices are used as references to the according subtrees in the upcoming calls.

Now let us revise what graph transformation rules are applied in the forall constructs. The `class2tableR` graph transformation rule (1.2, 2.1) - after matching the name of the input class onto the unbound variable `ClsNM` - creates a new table in `db` with this name, creates in this table a column `id` and sets it as a primary key, creates the mapping reference entity in `ref` and its source and target relations and prints out a notification with the fully qualified name of the class and the table.

```
//graph transformation rule for mapping classes to tables
gtrule class2tableR(in Cls) = {          //(2.1)
  precondition pattern lhs (Cls, ClsNM) = {
   Class(Cls) below models("uml");
   NamedElement.name(N1, Cls, ClsNM);
   String(ClsNM) below models("uml");
  }
  action {
   let T = undef in
   let R = undef in
   let RS = undef in
   let RT = undef in seq {
    call createNewTable(value(ClsNM), T);
    call createPrimaryKeyInTable(T);
    new (class2table(R) in models("ref"));
    new (class2table.srcRef(RS, R, Cls));
    new (class2table.trgRef(RT, R, T));
    print("Class "+ fqn(Cls)+ "-> Table" + fqn(T) +"\n");
   }
  }
}
```

The `attr2columnR` graph transformation rule (1.3, 3.1) is called for all entity pairs under `uml` - the precondition filters out all pairs that are not associated class-property pairs. Using the mappings recorded in `ref`, the precondition identifies the table created in `db` from the class of the attribute. The action part of the graph transformation rule creates a new column in the table with the respective name and also establishes the mapping reference entity.

```
//graph transformation rule for mapping attributes (not association type)
gtrule attr2columnR(in Cls, in Attr) = {                          //(3.1)
  precondition pattern lhs (Cls, Attr, NMAttr, Tab) = {
   Class(Cls) below models("uml");
   StructuredClassifier.ownedAttribute(F, Cls, Attr);
   Property(Attr) below models("uml");
   NamedElement.name(N1, Attr, NMAttr);
   String(NMAttr) below models("uml");
   class2table.srcRef(RS, R, Cls);
   class2table(R) below models("ref");
   class2table.trgRef(RT, R, Tab);
   Table(Tab) below models("db");
   neg pattern assocProperty(Attr) =
   {
           Property(Attr) below models("uml");
           Association.memberEnd(ME, Asc, Attr);
           Association(Asc) below models("uml");
   }
  }
  action {
  let Col = undef in
  let R = undef in
  let RS = undef in
  let RT = undef in seq {
    call createNewColumn(value(NMAttr), Tab, Col);
    new (attr2column(R) in models("ref"));
    new (attr2column.srcRef(RS, R, Attr));
    new (attr2column.trgRef(RT, R, Col));
    print("Attribute: "+ fqn(Attr)+ "-> Column: " + fqn(Col) +"\n");
   }
  }
 }
```

The graph transformation rules `attrOfStringTypeR` (1.4) and `attrOfIntTypeR` (1.5) search for attributes in the UML model with types 'String' and 'int' and determine the columns to which these attributes are mapped (via the mapping model). As action a new instance of the VIATRA built-in datatype `String` is created with the respective type name as its name and is bounded to the respective column in a new `Column.col_type` relation instance.

Last but not least, the graph transformation rule `assoc2tableR` (1.6) is called. Here the precondition part searches for the table-pairs to which the source and the target classes of associations were mapped earlier; in the action part the association is mapped into a new table with two foreign key columns (referring to the class-tables). Additionally, a surrogate key column is also created. The mapping is recorded in `ref`. As the structure of this transformation is almost identical to the previous two, we do not discuss it in detail.

8.4.2 Code Generation

As a last step we want to generate SQL code that generates the database structure we gained with the transformation; the code generator ASM machine is defined in `codegen.vtcl` under the name `codegen`. Note that before calling this machine the machine `ordering` must be run on database models reached with the transformation above for establishing a column- and foreign key column-ordering that `codegen` implicitly expects. The body of the main rule of the machine is as follows:

```
//The main rule of the code generator
rule main (in A) = seq
{
        let DBModel = ref(A) in
        forall T below DBModel, TName with find isTable(T,TName) do seq
        {
        print("create table "+value(TName)+" (");
        forall C below DBModel, Cn, Ct with
                find notLastColumn(T,C,Cn,Ct) do
                call printTableColumn(T,C,Cn,Ct);
        choose C below DBModel, Cn, Ct with
                find lastColumn(T,C,Cn,Ct) do
                call printLastTableColumn(T,C,Cn,Ct);
        print(")\n");

        let TRefName = undef in
        let RefTable = undef in
        forall Fk below DBModel with find isFKey(T,Fk,TRefName,RefTable) do
                forall FC below DBModel, Col, ColNm with
                        find isFKeycolumn(Fk,Col,ColNm) do
                        choose PC below DBModel, Col2, ColNm2 with
                                find isPKeycolumn(RefTable,Col2,ColNm2) do
                                print("alter table "+value(TName)+
                                        " add foreign key (" + value(ColNm) +
                                        ") on "+value(TRefName)+" (" +
                                    value(ColNm2)+ ");\n");
        }
}
```

Input parameter is a string identifying the VPM model space subtree where database model elements are to be searched for. The rule iterates on all tables in this subtree; for each table, the `CREATE TABLE` statement is formulated first (note that we know a priori that only one primary key is possible per table, so the syntactical peculiarities of compound key building do not have to be addressed). After the `CREATE TABLE` statement the foreign key declarations are generated.

8.4.3 The Auxiliary Transformations

The `orderingOfCols.vtcl` file defines the ASM machine `ordering` that establishes

- `reldb.metamodel.Table.tab_cols.next` and
- `reldb.metamodel.FKey.fkey_cols.next`

relations on all tables and foreign keys in the model space subtree given as input parameter. The `modelManagement.vtcl` file defines the ASM machine `modelManagement` which can be used for model space management purposes as creating/deleting elements given by name and fully qualified parent name. Both machines have auxiliary functions; discussing them in detail is out of the scope of this chapter. For further details please refer to the code listings in the Appendix.

Appendix A Object-Relational Transformation Source Code Listings

A.1 reldb_meta.vtml

```
entity(reldb)
{
  entity(metamodel)
  {
    entity('DBElement');
    entity('Table')
    {
            relation(fkeys, 'Table', 'FKey');
            multiplicity(fkeys, one_to_many);
            isAggregation(fkeys, true);

            relation(pkey, 'Table', 'Column');
            multiplicity(pkey, many_to_many);

            relation(tab_cols, 'Table', 'Column');
            multiplicity(tab_cols, one_to_many);
            isAggregation(tab_cols, true);

            relation(next, reldb.metamodel.Table.tab_cols,
                                    reldb.metamodel.Table.tab_cols);

            relation(tbl_name, 'Table', datatypes.'String');
            multiplicity(tbl_name, one_to_one);
            isAggregation(tbl_name, true);
    }
    supertypeOf('DBElement','Table');
    entity('FKey');
    supertypeOf('DBElement','FKey');
    entity('Column')
    {
            relation(col_name, 'Column', datatypes.'String');
            multiplicity(col_name, one_to_one);
    isAggregation(col_name, true);

            relation(col_type, 'Column', datatypes.'String');
            multiplicity(col_type, one_to_one);
            isAggregation(col_type, true);
    }
    supertypeOf('DBElement','Column');

    relation(references, 'FKey', 'Table');
    multiplicity(FKey.references, many_to_one);

    relation(fkey_cols, 'FKey', 'Column');
    multiplicity(FKey.fkey_cols, many_to_many);
    relation(next, FKey.fkey_cols, FKey.fkey_cols);

  }
  entity(models);
  entity(transformations);
}
```

A.2 ref_uml2reldb_meta.vtml

```
import uml2.metamodel;
import reldb.metamodel;

entity(uml2reldb)
{
  entity(metamodel)
  {
        entity(ref_uml2rel)
        {
                relation(srcRef, ref_uml2rel, Element);
                relation(trgRef, ref_uml2rel, DBElement);
        }
        entity(class2table)
        {
                relation(srcRef, class2table, Class);
                relation(trgRef, class2table, Table);
            supertypeOf(ref_uml2rel.srcRef, class2table.srcRef);
            supertypeOf(ref_uml2rel.trgRef, class2table.trgRef);
        }
    supertypeOf(ref_uml2rel, class2table);

    entity(attr2column)
    {
                relation(srcRef, attr2column, Property);
                relation(trgRef, attr2column, Column);
            supertypeOf(ref_uml2rel.srcRef, attr2column.srcRef);
            supertypeOf(ref_uml2rel.trgRef, attr2column.trgRef);
    }
    supertypeOf(ref_uml2rel, attr2column);

        entity(assoc2table)
        {
                relation(srcRef, assoc2table, Association);
                relation(trgRef, assoc2table, Table);
            supertypeOf(ref_uml2rel.srcRef, assoc2table.srcRef);
            supertypeOf(ref_uml2rel.trgRef, assoc2table.trgRef);
        }
  }
  entity(models);
  entity(transformations);
}
```

A.3 orderingOfCols.vtcl

```
namespace reldb.transformations;
import reldb.metamodel;
import datatypes;

machine ordering
{
        pattern selectFirst_TabCols(T, CL) =
        {
                Table(T);
                Table.tab_cols(CL, T, C);
                Column(C);
                neg pattern noNext(T, CL, C) =
                {
                        Table(T);
                        Table.tab_cols(CL, T, C);
                        Column(C);
                        Table.tab_cols(CL2, T, C2);
                        Column(C2);
                        Table.tab_cols.next(N1, CL, CL2);
                }
        }

        pattern selectNext_TabCols(T, CL, PrevCL) =
        {
                Table(T);
                Table.tab_cols(CL, T, C);
                Column(C);
                Table.tab_cols(PrevCL, T, PrevC);
                Column(PrevC);
                neg pattern noNext(T, CL, C) =
                {
                        Table(T);
                        Table.tab_cols(CL, T, C);
                        Column(C);
                        Table.tab_cols(CL2, T, C2);
                        Column(C2);
                        Table.tab_cols.next(N1, CL, CL2);
                }
        }

        rule collectNext_TabCols(in T) =
          try
                choose FirstCol with find selectFirst_TabCols(T, FirstCol) do
                        call setNext_TabCols(FirstCol, T);
          else
            skip;
```

```
rule setNext_TabCols(in OldCol, in T) =
  try
        choose NextCol with find selectNext_TabCols(T, NextCol, OldCol)
 do
                let Next = undef in seq
                {
                        print(fqn(NextCol));
                        new ('Table'.tab_cols.next(Next, OldCol, NextCol));
                        call setNext_TabCols(NextCol, T);
                }
  else
    skip;

pattern isTable(Table) =
{
        Table(Table);
}

pattern isFKey(FKey) =
{
        FKey(FKey);
}

pattern selectFirst_fkeyCols(F, CL) =
{
        FKey(F);
        FKey.fkey_cols(CL, F, C);
        Column(C);

        neg pattern noNext(F, CL, C) =
        {
                FKey(F);
                FKey.fkey_cols(CL, F, C);
                Column(C);
                FKey.fkey_cols(CL2, F, C2);
                Column(C2);
                FKey.fkey_cols.next(N1, CL, CL2);
        }
}
```

```
        pattern selectNext_fkeyCols(F, CL, PrevCL) =
        {
                FKey(F);
                FKey.fkey_cols(CL, F, C);
                Column(C);
                FKey.fkey_cols(PrevCL, F, PrevC);
                Column(PrevC);

                neg pattern noNextFKey(F, CL, C) =
                {
                        FKey(F);
                        FKey.fkey_cols(CL, F, C);
                        Column(C);
                        FKey.fkey_cols(CL2, F, C2);
                        Column(C2);
                        FKey.fkey_cols.next(N1, CL, CL2);
                }
        }

        rule collectNext_fkeyCols(in F) =
          try
                choose FirstCol with find selectFirst_fkeyCols(F, FirstCol) do
                        call setNext_fkeyCols(FirstCol, F);
          else
            skip;

        rule setNext_fkeyCols(in OldCol, in F) =
          try
                choose NextCol with find selectNext_fkeyCols(F, NextCol,
         OldCol) do
                        let Next = undef in seq
                        {
                                new ('FKey'.fkey_cols.next(Next, OldCol, NextCol));
                                call setNext_fkeyCols(NextCol, F);
                        }
          else
            skip;

        rule main (in A) =
                let DBModel = ref(A) in seq {

                forall T below DBModel with find isTable(T) do
                        call collectNext_TabCols(T);

                forall F below DBModel with find isFKey(F) do
                        call collectNext_fkeyCols(F);
                }
}
```

A.4 modelManagement.vtcl

```
machine modelManagement
{
 pattern entityExists(M) =
 {
        entity(M);
 }

 rule cleanupModel(in M) = seq
 {
        forall X below M with find entityExists(X) do
                delete( X );
 }

rule lookupAndCreate(in ParentStr, in ModelStr, in Mode, out Model) =  seq
 {
  print("Processing:" + ParentStr + ":" + ModelStr + ":" + Mode + "\n");
  let PARENT = ref(ParentStr) in
        if (PARENT == undef) seq
        {
                log(error,"Container entity " + ParentStr+ " does not exist.");
                fail;
        }
        else seq
        {
                update Model = ref(ParentStr+"."+ModelStr);
                if (Model == undef)
                        if (Mode == "LOOKUP_AND_CLEAN" || Mode ==
                                                "LOOKUP_NO_CLEAN")
                seq
                        {
                                log(error,"Model element " + ModelStr + " does not
                                                                        exist.");
                                fail;
                        }
                        else if (Mode == "CREATE_ONLY" || Mode ==
                                        "CLEAN_OR_CREATE")
                seq
                        {
                                new(entity(Model) in PARENT);
                                rename (Model, ModelStr);
                        }
                        else seq
                        {
                                log(error,"Unknown model creation mode: "+Mode);
                                fail;
                        }
```

```
                else
                        if (Mode == "LOOKUP_AND_CLEAN" || Mode ==
                                        "CLEAN_OR_CREATE")
                seq
                        {
                                call cleanupModel(Model);
                        }
                        else if  (Mode == "CREATE_ONLY") seq
                        {
                                log(error,"Model element " + ModelStr + " already
                                        exists and not overwritten.");
                                fail;
                        }
                        else if (Mode == "LOOKUP_NO_CLEAN" )
                                log(info,"Model element " + ModelStr + "
                                initialized successfully.");
                        else seq
                        {
                                log(error,"Unknown model creation mode: "+Mode);
                                fail;
                        }
        }
 }
}
```

A.5 uml2reldb_xform.vtcl

```
namespace uml2reldb.transformations;
import uml2.metamodel;
import reldb.metamodel;
import uml2reldb.metamodel;
import datatypes;

machine uml2reldb_xform
{

 asmfunction models / 1 ;

 rule main (in UMLStr, in RefStr, in DBStr) = seq
 {
        call initModels(UMLStr, RefStr, DBStr);
        forall C below models("uml") with apply class2tableR(C) do skip;
        forall C below models("uml"), A below models("uml") with
                apply attr2columnR(C, A) do skip;
        forall A below models("uml") with apply attrOfStringTypeR(A) do skip;
        forall A below models("uml") with apply attrOfIntTypeR(A) do skip;
        forall A below models("uml") with apply assoc2tableR(A) do skip;
 }

 rule initModels(in UMLStr, in RefStr, in DBStr) =
        let UMLModel = undef in
        let DBModel = undef in
        let RefModel = undef in seq
        {
        call modelManagement.lookupAndCreate("uml2", UMLStr,
                        "LOOKUP_NO_CLEAN",  UMLModel);
      update models("uml") = UMLModel;
          call modelManagement.lookupAndCreate("uml2reldb.models", RefStr,
                        "CLEAN_OR_CREATE",  RefModel);
      update models("ref") = RefModel;
        call modelManagement.lookupAndCreate("reldb.models", DBStr,
                "CLEAN_OR_CREATE",  DBModel);
      update models("db") = DBModel;
        }

 gtrule class2tableR(in Cls) = {
  precondition pattern lhs (Cls, ClsNM) = {
   Class(Cls) below models("uml");
   NamedElement.name(N1, Cls, ClsNM);
   String(ClsNM) below models("uml");
  }
  action {
   let T = undef in
   let R = undef in
   let RS = undef in
   let RT = undef in seq {
    call createNewTable(value(ClsNM), T);
    call createPrimaryKeyInTable(T);
    new (class2table(R) in models("ref"));
    new (class2table.srcRef(RS, R, Cls));
    new (class2table.trgRef(RT, R, T));
    print("Class "+ fqn(Cls)+ "-> Table" + fqn(T) +"\n");
   }
  }
}
```

```
gtrule attr2columnR(in Cls, in Attr) = {
 precondition pattern lhs (Cls, Attr, NMAttr, Tab) = {
  Class(Cls) below models("uml");
  StructuredClassifier.ownedAttribute(F, Cls, Attr);
  Property(Attr) below models("uml");
  NamedElement.name(N1, Attr, NMAttr);
  String(NMAttr) below models("uml");
  class2table.srcRef(RS, R, Cls);
  class2table(R) below models("ref");
  class2table.trgRef(RT, R, Tab);
  Table(Tab) below models("db");
  neg pattern assocProperty(Attr) = {
          Property(Attr) below models("uml");
          Association.memberEnd(ME, Asc, Attr);
          Association(Asc) below models("uml");
 }}
 action {
 let Col = undef in
 let R = undef in
 let RS = undef in
 let RT = undef in seq {
   call createNewColumn(value(NMAttr), Tab, Col);
   new (attr2column(R) in models("ref"));
   new (attr2column.srcRef(RS, R, Attr));
   new (attr2column.trgRef(RT, R, Col));
   print("Attribute: "+ fqn(Attr)+ "-> Column: " + fqn(Col) +"\n");
  }
 }
}

gtrule attrOfStringTypeR(in Attr) = {
 precondition pattern lhs (Attr, Col) = {
  Property(Attr) below models("uml");
  TypedElement.type(TP, Attr, DTType);
  PrimitiveType(DTType) below models("uml");
  attr2column.srcRef(RS, R, Attr);
  attr2column(R) below models("ref");
  attr2column.trgRef(RT, R, Col);
  Column(Col) below models("db");
  check (name(DTType) == "String")
 }
 action {
   call createNewColumnType(Col, "VARCHAR(50)");
 }
}

gtrule attrOfIntTypeR(in Attr) = {
 precondition pattern lhs (Attr, Col) = {
  Property(Attr) below models("uml");
  TypedElement.type(TP, Attr, DTType);
  PrimitiveType(DTType) below models("uml");
  attr2column.srcRef(RS, R, Attr);
  attr2column(R) below models("ref");
  attr2column.trgRef(RT, R, Col);
  Column(Col) below models("db");
  check (name(DTType) == "int")
 }
 action {
   call createNewColumnType(Col, "INTEGER");
 }
}
```

```
gtrule assoc2tableR(in Assoc) = {
 precondition pattern lhs (Assoc, AE1, AE2, SrcC, TrgC, SrcTab, TrgTab) = {
  Association(Assoc) below models("uml");
  Association.memberEnd(C1, Assoc, AE1);
  Property(AE1) below models("uml");
  TypedElement.type(T1, AE1, SrcC);
  Class(SrcC) below models("uml");
  class2table.srcRef(RS1, R1, SrcC);
  class2table(R1) below models("ref");
  class2table.trgRef(RT1, R1, SrcTab);
  Table(SrcTab) below models("db");
  Association.memberEnd(C2, Assoc, AE2);
  Property(AE2) below models("uml");
  TypedElement.type(T2, AE2, TrgC);
  Class(TrgC) below models("uml");
  class2table.srcRef(RS2, R2, TrgC);
  class2table(R2) below models("ref");
  class2table.trgRef(RT2, R2, TrgTab);
  Table(TrgTab) below models("db");
 }
 action {
 let T = undef in
 let myIdCol = undef in
 let SrcCol = undef in
 let PK1 = undef in
 let FKSrc = undef in
 let TrgCol = undef in
 let FKTrg = undef in
 let R = undef in
 let RS = undef in
 let RT = undef in seq {

  call createNewTable(name(AE1) + "_"+ name(SrcC) + "_" + name(AE2)+ "_"+
                      name(TrgC), T);

  call createNewColumn("id", T, MyIdCol);
  call createNewColumnType(MyIdCol, "INTEGER");
  new('Table'.pkey(PK1, T, MyIdCol));

  call createNewColumn(name(SrcC)+ "_id", T, SrcCol);
  call createNewColumnType(SrcCol, "INTEGER");
  call createNewFKey(T, SrcCol, SrcTab, FKSrc);

  call createNewColumn(name(TrgC)+ "_id", T, TrgCol);
  call createNewColumnType(TrgCol, "INTEGER");
  call createNewFKey(T, TrgCol, TrgTab, FKTrg);

  new (assoc2table(R) in models("ref"));
  new (assoc2table.srcRef(RS, R, Assoc));
  new (assoc2table.trgRef(RT, R, T));

  }
 }
}
```

```
rule createNewTable(in TabName, out T) =
let N1 = "undef" in
let Str = "undef" in seq {
   new ('Table'(T) in models("db"));
   new (datatypes.'String'(Str) in models("db"));
   setValue(Str, TabName);
   new ('Table'.tbl_name(N1, T, Str));
}

rule createPrimaryKeyInTable(in T) =
let PKCol = undef in
let PK1 = undef in seq {
       print(fqn(T) + "\n");
   call createNewColumn("id", T, PKCol);
   call createNewColumnType(PKCol, "INTEGER");
   new ('Table'.pkey(PK1, T, PKCol));
}

rule createNewColumn(in ColName, in Tab, out Col) = seq {
   new ('Column'(Col) in models("db"));
   new (datatypes.'String'(StrNM) in models("db"));
   setValue(StrNM, ColName);
   new ('Column'.col_name(N3, Col, StrNM) );
   new ('Table'.tab_cols(TC1, Tab, Col) );
}

rule createNewColumnType(in Col, in ColType) = seq {
   new (datatypes.'String'(StrNM) in models("db"));
   setValue(StrNM, ColType);
   new ('Column'.col_type(N1, Col, StrNM) );
}

rule createNewFKey(in Tab, in Col, in RefTab, out FK) = seq {
  new ('FKey'(FK) in models("db"));
  new ('Table'.'fkeys'(FK1, Tab, FK) );
  new ('FKey'.'references'(RF1, FK, RefTab) );
  new ('FKey'.'fkey_cols'(FKC1, FK, Col) );
}

}
```

A.6 orderingOfCols.vtcl

```
namespace reldb.transformations;
import reldb.metamodel;
import datatypes;

machine codegen {

        pattern isTable(Table,Name) =
        {
                'Table'(Table);
                'String'(Name);
                Table.tbl_name(TN1,Table,Name);
        }

        pattern lastColumn(Table,C,Coln,Colt) =
        {
                'Table'(Table);
                Table.tab_cols(TC,Table,C);
                'Column'(C);
                Column.col_name(CN,C,Coln);
                'String'(Coln);
                Column.col_type(CT,C,Colt);
                'String'(Colt);
                neg find hasNextColumn(Table, TC, C)
        }

        pattern notLastColumn(Table, C, Coln, Colt) =
        {
                'Table'(Table);
                Table.tab_cols(TC,Table,C);
                'Column'(C);
                Column.col_name(CN,C,Coln);
                'String'(Coln);
                Column.col_type(CT,C,Colt);
                'String'(Colt);
                find hasNextColumn(Table, TC, C)
        }

        pattern hasNextColumn(Table, TC1, C1) = {
                'Table'(Table);
                Table.tab_cols(TC1,Table,C1);
                'Column'(C1);
                Table.tab_cols(TC2,Table,C2);
                'Column'(C2);
                Table.tab_cols.next(N1, TC1, TC2);
        }


        pattern isPrimKey(Table,C) =
        {
                'Table'(Table);
                'Column'(C);
                Table.pkey(TC,Table,C);
        }
```

```
pattern isFKey(Table,FK,Tname,Table2) =
{
        'Table'(Table);
        'Table'(Table2);
        'FKey'(FK);
        Table.fkeys(TC,Table,FK);
        FKey.references(FKR,FK,Table2);
        'String'(Tname);
        Table.tbl_name(TN1,Table2,Tname);
}

pattern isFKeycolumn(FK,Col,CName) =
{
        'FKey'(FK);
        'Column'(Col);
        FKey.fkey_cols(FC,FK,Col);
        'String'(CName);
        Column.col_name(CN,Col,CName);
}

pattern isPKeycolumn(Table,Col,CName) =
{
        'Table'(Table);
        'Column'(Col);
        Table.tab_cols(TC,Table,Col);
        Table.pkey(PK,Table,Col);
        datatypes.'String'(CName);
        Column.col_name(CN,Col,CName);

}

rule printTableColumn(in T, in C, in Cn, in Ct) =
    if (find isPrimKey(T,C))
        print("    " + value(Cn) + " " + value(Ct)+ " primary key,\n");
    else
        print("    " + value(Cn) + " " + value(Ct) + ",");

rule printLastTableColumn(in T, in C, in Cn, in Ct) =
    if (find isPrimKey(T,C))
        print("    " + value(Cn) + " " + value(Ct)+ " primary key\n");
    else
        print("    " + value(Cn) + " " + value(Ct));

rule printFKeyColumn(in F, in C, in Cn, in Ct) =
        print("    " + value(Cn) + " " + value(Ct)+ " primary key,\n");

rule printLastFKeyColumn(in F, in C, in Cn, in Ct) =
        print("    " + value(Cn) + " " + value(Ct)+ " primary key\n");
```

```
   rule main (in A) = seq
   {
       let DBModel = ref(A) in
       forall T below DBModel, TName with find isTable(T,TName) do seq
       {
           print("create table "+value(TName)+" (");
           forall C below DBModel, Cn, Ct with
                   find notLastColumn(T,C,Cn,Ct) do
                      call printTableColumn(T,C,Cn,Ct);
           choose C below DBModel, Cn, Ct with
                   find lastColumn(T,C,Cn,Ct) do
                      call printLastTableColumn(T,C,Cn,Ct);
           print(")\n");

           let TRefName = undef in
           let RefTable = undef in
           forall Fk below DBModel with find isFKey(T,Fk,TRefName,RefTable) do
               forall FC below DBModel, Col, ColNm with
                     find isFKeycolumn(Fk,Col,ColNm) do
                   choose PC below DBModel, Col2, ColNm2 with
                       find isPKeycolumn(RefTable,Col2,ColNm2) do
                           print("alter table "+value(TName)+
                                 " add foreign key (" + value(ColNm) +
                                 ") on "+value(TRefName)+" (" +
                                 value(ColNm2)+ ");\n");
            }
       }
}
```

Bibliography

[1] E. Börger and R. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[2] Object Management Group. *Object Constraint Language Specification (Version 2.0)*, May 2006. http://www.omg.org.

[3] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.

[4] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.