The Viatra-I Model Transformation Framework Pattern Language Specification

© 2006. OptXware Research and Development LLC. This document is property of the OptXware Research and Development LLC. To copy the whole or parts of it, or passing the contained information to third parties is allowed only with prior approval of OptXware Research and Development LLC.

Abstract

We present a specification for model transformation language of the VIATRA2 framework, which provides a rule and pattern-based transformation language for manipulating graph models by combining graph transformation and abstract state machines into a single specification paradigm. This language offers advanced constructs for querying (e.g. recursive graph patterns) and manipulating models (e.g. generic and meta transformation rules). In addition, powerful language constructs are provided for multi-level metamodeling to design modeling languages.

Date	August 26th, 2006
Document type	Specification
State	Final
Approved by	György Csertán
Contact	Dániel Varró
Version	1.0

Contents

1 Viatra-I Architecture Specification

1.1 Introduction

This chapter specifies the software architecture of the VIATRA-I model transformation framework. This is the basis of the implementation of the new framework version.

1.2 Implications of earlier assessment

As stated in the VIATRA2 implementation assessment documentation, some of the existing components have to be reimplemented, and the architecture has to be reconsidered according the experiences of the current version.

The assessment has pointed out that the original Eclipse-based architecture is mature, and it has only some problematic points that should be corrected.

1.3 Definition of the VIATRA-I architecture

1.3.1 Runtime extensions

The platform that VIATRA is running on, is the open-source Eclipse tool integration framework. This platform offers an extensible, component-based infrastructure. During the architectural planning we identified the points where VIATRA should contribute to the base platform to achieve tight integration with the existing components.

User interface extensions

The first, basic extension is the definition of an *editor* extension. Editors in Eclipse are used for implementing resource (most often file)-based access to information, using an open-modify-save paradigm. VIATRA uses this facility to display the model spaces (information stores that contain models, meta models, and transformation programs). Through the editor interface, users can ope, edit, and save the contents of model spaces. The editor implementation should use a tree control to display the model space.

Besides the editor, there are several other supplementary user interface elements. The *PropertySource* extension is implementing a property page for model elements that appears in the standard Eclipse

property pages editor. The VIATRA model space editor should also provide such an extension.

There are two components that are needed and implement the Eclipse *Views* extension point. The first one is a code output buffer that displays the code generated by model to text transformation executions. This is mainly for testing purposes. The other one is the *Frameworks view* that displays the model spaces currently in memory. The user can use this list to dispose the model space, or to invoke model importers that load additional models into the selected model space.

The framework should also provide means for creating new model spaces. Eclipse offers the *wizard* facility for this. The framework should define a wizard that guides the user through the model space creation process.

The above mentioned extensions together form the VIATRA framework GUI.

Transformation and model editing

Besides the model space editor, a *text editor* extension also must be specified that implements a textual syntax highlighting, content assist-capable editor for Viatra Textual Command Language and Viatra Textual Modeling Language definitions. The first language is used for transformation specification, and the second one for (meta) model definition.

The editor should also implement supplementary extension for *wizards* that support the creation of new transformation and model source files, and an *outline* extension that displays the outline of the currently edited file.

Extension points provided by VIATRA

The VIATRA framework itself has also to provide extension points that follow the concept of the Eclipse framework to let other developers to contribute to the project.

There must be an *interpreter* extension point that is able to handle external components that interpret a specific type of model element. For instance, if a component implements the simulation of Petri nets, it should be able to connect to the framework, and the user should be able to start the simulation by selecting the appropriate command on a Petri net object in the model space. The extension point must also be able to handle runtime parameters specified by the user as a mean of parametrization of the execution.

There must be an *importer* extension point that manages the model

importers connected to the system. Model importers convert models from the actutal syntax of an external modeling tool to abstract syntax that can be represented in the VIATRA model space.

There must be a *code formatter* extension that manages the code output components. These components are used to format and persist the output of the model-to-code transformations. One example of code formatters is the built-in code output buffer view.

There must be also a *native function* extension point that manages the external functions that are integrated into the VIATRA model interpreter. This feature supports the external definition of complex computations in Java, instead of defining it in the VIATRA GTASM language.

These extensions are present also in the existing version of VIATRA, but they have to inspected, tested, and ported to the new implementation.

1.3.2 Component structure of VIATRA-I

The component architecture of the VIATRA-I system can be seen in Figure 1.1. It is derived from the previous implementation, but with significant re-engineering to better componentize the system.

The *core* component is responsible for model space persistence and manipulation. This component offers all the extension points that have been specified in the earlier sections.

The *framework.gui* component is providing the graphical user interface of the framework, based on the core component. It provides the editor, the properties view content, and the new model space wizard.

The *model.synchronizer* bridges the model space and EMF representation of transformation programs. VIATRA uses model space for program storage and transport, as models, meta models and transformation programs can be uniformly handled that way. However, the analysis of the current implementation had shown that there is need for a common, runtime, in-memory representation of the programs that can be manipulated easier as the model space. This is the *emf.model* component that is the automatically generated implementation of the transformation language metamodel. All runtime components work on this model, but for storage it is stored in the model space. This storage and retrieval is done by the *model.synchronizer* component.

The *patternmatcher* component implements pattern matching strategies that are needed by the graph transformation interpreter. It is separated by the interpreter, because it strongly depends on the core

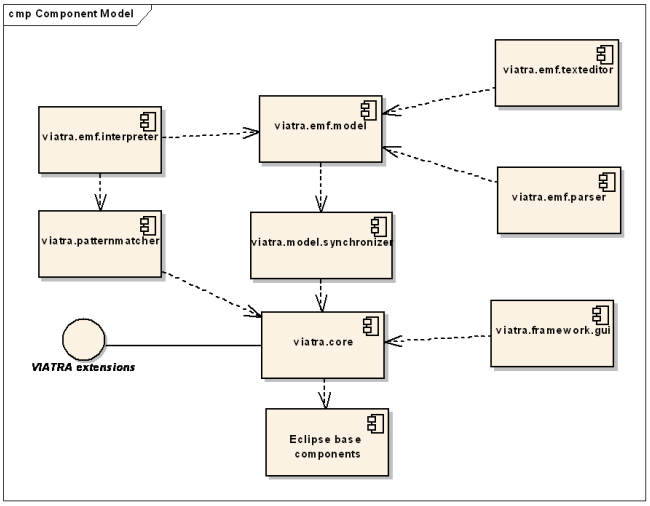


Figure 1.1: The component architecture of VIATRA2 system

implementation. If the implementation of the model management changes, the pattern matcher should also be changed (for performance reasons).

The *emf.interpreter* component is the new GTASM interpreter that works on EMF models of transformation programs.

The *emf.texteditor* component is the syntax highlight capable editor for model and transformation source files. It also uses the EMF-based transformation program representation.

THe *emf.parser* component is the new version of the VTCL parser directly building EMF transformation models instead of a custom abstract syntax tree.

1.4 Summary

This document defines the architecture of the new VIATRA implementation that is a re-engineered version of the current one. The new architecture introduces strong componentization while minimizing the duplication of code in multiple components. The architecture is also aligned with the concepts of the Eclipse framework.

2 Viatra-I Textual Editor Requirements

2.1 Introduction

In the VIATRA framework the user can create and manipulate models and metamodels in various ways. One way is to use special purpose computer languages to describe and manipulate these models. The definition of the models can be done with the VIATRA Textual Metamodeling Language (VTML), and the model manipulation processes can be described by the VIATRA Textual Command Language (VTCL). Both language uses advanced language constructs like rule definitions with parameters and variables, function and graph pattern definitions which can be quite complex. Therefore supporting these languages with a rich source editor makes the development process in the VIATRA framework more convenient and more accurate. The source editor will be called VIATRA Textual Editor and will provide rich support for both languages.

2.2 Overview

The VIATRA Textual Editor has to be an integrated part of the VIATRA framework. This means that it has to act as the default textual editor for the VTML and VTCL files in the VIATRA framework, altough the user must be able to select a different editor for these file types. The VIATRA Textual Editor must be able to handle multiple documents opened in parallel and it has to provide rich editor functionality for all of them, like syntax highlighting, content assist, a basic error reporting, model building and validation of the model.

2.3 Required Functionalities

The following chapters are describing the different functionalities that the VIATRA Textual Editor has to provide in the VIATRA framework, and they also give a short overview of their expected behaviours.

2.3.1 Syntax Highlighting

The purpose of syntax highlighting is to highlight certain elements of the source code in the editor, therefore providing a more perspicuous view of the structures in the document. The VIATRA Textual Editor

has to offer a dynamic syntax highlihgting solution which supports multiple classes of elements. The required classes to support are:

- Keywords : special, reserved character strings of a computer language
- Strings : character strings enclosed by quotation marks
- Default elements : any other element
- Comments

Highlighting an element means that the element has a different font property than a default element. The font property can be the color or the tickness of the character font. The VIATRA Textual editor has to provide a way to change these properties in a preference page and the changes must be persisted automatically. The highlighting of the elements must be updated immediately after any change in the document, this means that if the user finishes entering a keyword, the editor has to change the keyword's font settings to reflect the current state and vice versa. The VIATRA Textual Editor has to differentiate between code and comment partitions in the source and apply different highlighting rules for them. Maintaining of these partitions must also be automatically done by the editor on every change in the source.

2.3.2 Content Assist

Content Assist (also known as Intellisense) is a very useful tool which makes code editing a lot faster and more accurate. Activating this tool by a certain key combination anywhere in the code under development will result a list of element proposals that are applicable in the current position. The VIATRA Textual Editor has to offer content assist functionality with the following requirements:

- Proposing every valid keyword in the current position
- Proposing every visible model element in the current position which are in the assigned model space
- Proposing every visible model element which are declared in the source code only
- Offer an alphabetically sorted list of the collected elements
- Offer valid elements only
- If the user has entered a partial string, content assist must try to complete the already entered part if activated in the correct position which is the end of the partial string

- If activated without a partial string, content assist must provide a list of all available proposals

In the VIATRA Textual Editor the content assist function is activated by the CTRL+SPACE key combination. After activation the user is provided with a list of available proposals (if there's any), and is able to select the required one by navigating in the list with the "UP" and "DOWN" button. The selection of an element happens with the pressing of the "ENTER" button on the element. The content assist function doesn't have to work on syntactically incorrect code. In that case syntactically incorrect code means that there's an invalid sequence of elements somewhere before the position of the activation, more exactly before the position of the partial string which the user wanted to complete.

2.3.3 Code Folding

With code folding, the user can hide certain parts of the document with a single mouse click. It is useful when dealing with a computer language which contains well defined blocks of character strings. A block in the code usually represents a functionality which is coherent in some manner, a block in a document can be a paragraph. VIATRA Textual Editor has to indicate foldable elements with a small sign on the editor's sidebar. Clicking on this sign will hide or show the block that it belongs to. For the VTML language, the VIATRA Textual Editor has to provide code folding for the following language constructs:

- Import declarations
- Compound entities which have further elements in their body

Compound entities can contain further compound entities in an arbitrary depth, code folding must be available to these entities too. For the VCTL language, the VIATRA Textual Editor has to provide code folding for the following language constructs:

- Import declarations
- Machine definitions
- ASM rule definitions
- Graph transformation rule definitions
- Graph transformation pattern definitons
- Compound entities

2.3.4 Outline

The code outline is an additional view of the content in the textual editor. As it's name suggests, it provides a simple outline of the content which makes the content more perspicuous and also helps in navigation in the content. When the content is a simple text document, the outline view could show the titles of the paragraphs in a tree structure. When the content is some code written in a computer language, the outline usually shows the identifiers of the main blocks in the code in a tree structure with a small definitive icon. The identifier and the icon together is often referred as the label of the element. Different types of language elements must have different labels, which means different icons and representative identifiers.

The VIATRA Textual Editor has to provide an outline view for both VTML and VTCL languages. The outline view must be always up-to-date, which means that it has to reflect the actual structure of the code all the time. Any subtree of the main tree structure must be foldable in an arbitrary depth. For the VTML language, the following language constructs must appear in the tree:

- Namespace declaration
- Import declarations with an explicit root element
- Entity definitions
- Relation definitions
- Relationship definitions

For the VTCL language, the following language constructs must appear in the outline view:

- Namespace declaration
- Import declarations with an explicit root element
- Entity definitions
- ASM rule definitions
- ASM functions
- Graph transformation rule definitions
- Graph pattern definitions

The outline view also provides a possibility to sort the shown elements alphabetically if requested.

2.3.5 Error reporting

The VIATRA Textual Editor has to provide basic error reporting functions. Error reporting means that if there's an error anywhere in the VTML or VTLC code, it must be pointed out by the editor. Since reliable recovering of the parser from a syntactical error is a hard problem, the VIATRA Textual Editor is only expected to detect and report the first syntactical error that occurs in the source, and then it has to do it's best effort to continue parsing and maybe detecting another error. The nature of this problem makes it impossible to define more exact requirements regarding the capabilities of the error reporting function. The requirements of detecting and reporting semantical errors can be found in chapter 3.7.

2.3.6 EMF model building

The VIATRA Textual Editor has to be capable of building an internal model of the VTCL source codes using a given EMF (Eclipse Modeling Framework) metamodel. The content of the model must be always up-to-date, therefore constant sycnhronization between the document that contains the code and the internal model is required. The internal model has to be accessible by another parts of the VIATRA framework too. The model building process has to be done in a non-incremental way to reduce complexity of the software. Non-incremental build (also known as full build) means that every content change in the source will result a completely new model and the destruction of the old one. The VIATRA Textual Editor has to store an up-to-date model in the memory for every document opened with it in the given session, and it has to be able to persist a model if requested.

2.3.7 The model validation framework

The model validation function has to be implemented in a flexible, extensible way, since this functionality can be quite complex and can be extended or modified as the VIATRA framework evolves. Therefore it is a main requirement to provide a validation framework that allows VIATRA developers to easily contribute to the validation process by writing their own validation contributions. These contributions, when placed into the proper directory, must be discovered and executed automatically by the VIATRA Textual Editor. The developer of the validation contribution must be able to define the ways of interaction with the editor, namely the desired execution mode of the contribution and the type of the language element it has to validate. The VIATRA Textual Editor has to differentiate between validation contributions that are executing for every change in the source and contributions that are executing only before importing the source to

a model space. The VIATRA Textual Editor has to make sure that every contribution gets executed for every language element with the corresponding type.

A validator has to be able to use the same error reporting functions as the parser uses to report syntactical errors in the editor and has to be able to access the model and the assigned model space of the source it is currently validating through a well-defined interface.

2.3.8 Integration with the VIATRA framework

The VIATRA Textual Editor has to provide ways of interaction with the VIATRA framework. It has to provide a dialog for selecting a model space for every VTML and VTCL document in the workspace and it has to maintain this relation till the end of the session, but it doesn't have to persist this assignment information. In the editor, it must be possible to import the content of the source into the model space by right-clicking on the document in the Navigator View and selecting the corresponding action. The editor must be able to run the VIATRA framework's machine interpreter with an internal model what is built and stored in that session.

3 Pattern Language Concepts

3.1 Introduction

In this chapter, we provide an informal introduction and a formal specification of the pattern language of the VIATRA-I model transformation framework.

In VIATRA-I , we have chosen to integrate two intuitive, and mathematically precise rule-based specification formalisms, namely, graph transformation (GT) [2] and abstract state machines (ASM) [1] to manipulate graph based models.

In the current paper, we present a brief introduction to the multilevel metamodeling (Sec. 3.2), while the main focus is put on the pattern specification (Sec. 3.3.3), and the specification of graph transformation rules.

3.2 Metamodeling Language

Metamodeling is a fundamental part of model transformation design as it allows the structural definition (i.e. abstract syntax) of modeling languages. Metamodels are represented in a metamodeling language, which is another modeling language for capturing metamodels.

We have chosen to use the VPM (Visual and Precise Metamodeling) [7] metamodeling approach in the VIATRA-I framework, which can support different metamodeling paradigms by supporting multilevel metamodeling with explicit and generalized instance-of relations. Models, metamodels and transformations are stored uniformly in VIATRA-I in a VPM *model space*.

As only an abstract syntax was defined for VPM in [7], we developed a textual concrete syntax for the metamodeling environment called *VTML (Viatra Textual Metamodeling Language)* for specifying metamodels and models. The syntax of the VTML language has a certain Prolog flavor, but it offers support for well-founded typing and hierarchical model libraries. Our experience showed that this textual format was more usable in case of large, or automatically generated metamodels compared to a graphical language.

Standard metamodeling paradigms can be integrated into VIATRA-I by import plugins and exporters defined by code generation transformations. While VIATRA-I offers the VTML language for constructing

models and metamodels, the main usage scenario is to bridge heterogeneous off-the-shelf tools by flexible model imports and exports.

3.2.1 Metamodeling concepts in VTML

Entities and relations

As shown in the metamodel of Fig. 3.1, the VPM model space consists of two basic elements: the entity (a generalization of MOF package, class, or object) and the relation (a generalization of MOF association end, attribute, link end, slot). *Entities* represent basic concepts of a (modeling) domain, while *relations* represent the relationships between model elements.[1] Furthermore, entities may also have an associated string value which contains application-specific data.

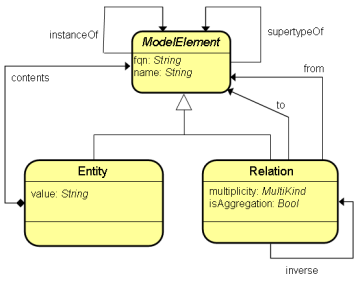


Figure 3.1: The metamodel of the VPM model space

Relations have additional properties. (i) Property *isAggregation* tells whether the given relation represents an aggregation in the metamodel, when an instance of the relation implies that the target element of the relation instance also contains the source element. (iii) *Multiplicities* impose a restrictions on the model structure. Allowed multiplicities in VPM are one-to-one, one-to-many, many-to-one, and many-to-many.

In VTML, an entity can be declared in the form *type(name)*, where type is the type of the entity and name is a fresh (unused) name for the new entity. In a relation definition takes the form: *type(name, source, target)*, where source and target are either existing elements in the model space, or they are defined (or to be defined) in the same VTML file. Type declarations are mandatory for all model elements with the basic VPM entity and relation model elements in the top of the type hierarchy.

Containment hierarchy and namespace imports

Model elements are arranged into a strict containment hierarchy. Within a container entity, each contained model element has a unique local name. In addition, a globally unique identifier called a fully qualified name (FQN) is defined for each model element. An FQN is derived by concatenation along the containment hierarchy of local names using dots (".") as separators. Relations are automatically assigned to their source model element within this containment hierarchy, which corresponds to the design decision in many

[1] Most typically, relations lead between two entities to impose a directed graph structure on VPM models, but the source and/or the target end of relations can also be relations.

modeling tools. For example, the FQN of the entity Association in Fig. 3.2 is UML.Association, while the FQN of the relation src is UML.Association.src.

VTML also allows to *import namespaces* from the model space for the given VTML file so that model elements in the imported namespaces can be referred to using their local names instead of their FQNs.

Inheritance and instantiation

There are two special relationships between model elements (i.e. either between two entities or two relations): the supertypeOf (inheritance, generalization) relation represents binary superclass-subclass relationships (like the UML generalization concept), while the instanceOf relation represents type-instance relationships (between meta-levels). By using explicit instanceOf relationship, metamodels and models can be stored in the same model space in a compact way. Note that both multiple inheritance and multiple typing are allowed (where the latter concept has a significant role in domain-specific languages supporting multi-domain editing, which is an ongoing development for VIATRA-I).

Finally, it is worth pointing out that many advanced modeling concepts of MOF 2.0 can be easily modeled in VPM. For instance, ordered properties can be represented as (ordering) relations leading between relations, while the subset property can be modeled by subtypeOf relationship between the two relations representing the properties.

3.2.2 Demonstrating example

The technicalities of VTML are demonstrated on a (simplified) UML metamodel.

The UML dialect used throughout the paper contains Classes, which may have attributes typed by a Classifier which is either a PrimitiveDataType or by other user defined Classes. Furthermore, Associations may lead from a source (src) class to a target (dst) class.

3.3 The Pattern Language

Graph patterns (GP) are the atomic units in VIATRA-I for capturing well-formedness rules of a modeling language and, especially, for specifying common patterns used in model transformation rules. Graph patterns are integral part of the *Viatra Textual Command Language (VTCL)*, which is the main textual language in VIATRA-I to specify model transformations.

```
entity(UML)
{

 entity(Classifier);
 entity(PrimitiveDataType);
 supertypeOf(Classifier,PrimitiveDataType);

 entity(Class);
 supertypeOf(Classifier,Class);
 relation(parent,Class,Class);

 entity(Association);
 relation(src,Association,Class);
 relation(dst,Association,Class);

 entity(Attribute);
 relation(type,Attribute,Classifier);
 relation(attrs,Class,Attribute);

 multiplicity(attrs,many-to-many);
 isAggregation(attrs, true);

}
```

Figure 3.2: A sample VTML metamodel for UML

Graph patterns represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. A model (i.e. part of the model space) can satisfy a graph pattern, if the pattern can be matched to a subgraph of the model using a generalized *graph pattern matching* technique presented in [6]. The formal semantics of graph patterns in VTCL will be discussed in Sec. 4.

3.3.1 Simple patterns, negative patterns

Patterns are close to predicates in Prolog, as they have a name, a parameter list, and a body. The body of a simple pattern contains model element and relationship definitions using VTML language constructs.

In the left column of Fig. 3.3, a simple pattern can be fulfilled by class which has an attribute with a user-defined type Here Class(C) declares a variable C to store an entity of type Class while Class.attrs(X,C,A) denotes a relation of type Class.attrs, which has an identifier X, and leads from class C to attribute A. Note that these predicates can be listed in an arbitrary order (unlike in Prolog) to yield a fully declarative specification.

The keyword *neg* marks a subpattern that is embedded into the current one to represent a negative condition for the original pattern. The negative pattern in the right column of Fig. 3.3 can be satisfied, if there is a class (CP) for the class in the parameter (C) that is the parent of C. If this condition can be satisfied, the outer (positive) pattern matching will fail. Thus the pattern matches to top-most classes in parent hierarchy.

```
import UML;

/*C is a UML class with an
  attribute A; A is of type T*/

pattern isClassAttribute(C, A) =
{
 Class(C);
//X is the relation between C and A
//it is an internal variable, its
//value is not present on the interface
 Class.attrs(X,C,A);
 Attribute(A);
 Attribute.type(Y,A,T);
 Classifier(T);
}
```

```
import UML;
/* C is a class without parents
     and with non-empty name */
pattern isTopClass(C, M) =
{
 Class(C) below M;

 neg pattern negCondition(C) =
 {
   Class(C);
   Class.parent(P,C,CP);
   Class(CP);
 }

 check (name(C)!="")
}
```

Figure 3.3: Basic patterns and negative patterns

Each entity in a pattern may be *scoped* by using the in or below keywords by a container entity. This means that the corresponding pattern element should be matched to a model element which resides directly inside (in) or somewhere below (below) its scope entity in the containment hierarchy of the model space.

There are also *check* conditions that are Boolean formulae which must be satisfied in order to make the pattern true. In our example, pattern can be matched to classes with non-empty names only.

A unique feature of the VTCL pattern language among graph transformation tools is that negative conditions can be embedded into each other in an arbitrary depth (e.g. negations of negations), where the expressiveness of such patterns converges to first order logic [4].

3.3.2 Pattern calls, OR-patterns, recursive patterns

In VTCL, a pattern may call another pattern using the find keyword. This feature enables the reuse of existing patterns as a part of a new (more complex) one. The semantics of this reference is similar to that of Prolog clauses: the caller pattern can be fulfilled only if their local constructs can be matched, and if the called (or referenced) pattern is also fulfilled.

Alternate bodies can be defined for a pattern by simply creating multiple blocks after the pattern name and parameter definition, and connecting them with the or keyword. In this case, the pattern is fulfilled if at least one of its bodies can be fulfilled. Naturally, OR-patterns can be called from other patterns, thus, allowing disjunction only on the top-level is not a real limitation.

The two features (pattern call and alternate (OR) bodies) can be used together for the definition of *recursive patterns*. In a typical recursive pattern, the first body (or bodies) define the halt condition for the recursion, while subsequent bodies contain a recursive call to itself. However, VIATRA-I supports general recursion, i.e. multiple recursive

calls are allowed from a pattern. Note that general recursion is not supported by any of the existing graph transformation tools up to now. The following example illustrates the usage of recursion.

```
// Parent is an ancestor (transitive parent) of Child
pattern ancestorOf(Parent,Child) =
{
   Class(Parent);
   Class(Child);
   Parent = Child
}
or
{
   Class(Parent);
   Class.parent(X,C,Parent);
   Class(C);
   find parentOf(C,Child);
   Class(Child);
}
```

A class Parent is the parent of an other class Child, if classes Parent and Child are identical, or Parent has a direct child (C), which is the parent of the child class (second body). The pattern uses recursion for traversing multi-level parent-child relationships, and uses multiple bodies to create a halt condition (base case) for the recursion.

3.3.3 Graph Transformation Rules

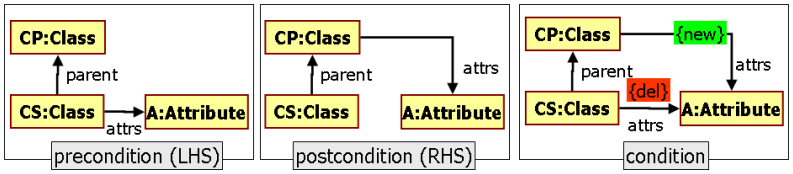
Transformation descriptions in VIATRA-I consist of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph transformation (GT) [2] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [1] rules can be used for the description of control structures. In the current section, we define graph transformation rules.

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules [2], which heavily rely on graph patterns for defining the application criteria of transformation steps. The application of a GT rule on a given model replaces an image of its left-hand side (LHS) pattern with an image of its right-hand side (RHS) pattern. The VTCL language allows different notations for defining graph transformation rules using the gtrule keyword.

3.3.4 Traditional notation of graph transformation rules

The sample graph transformation rule in Fig. 3.4 defines a refactoring step of lifting an attribute from child to parent classes. This means that if the child class has an attribute, it will be lifted to the parent.

The first syntax of a GT rule corresponds to the traditional notation (see Fig. 3.4(a)). It contains a *precondition* pattern for the LHS, and a *postcondition* pattern that defines the RHS of the rule. In general, elements that are present only in (the image of) the LHS



(a) Traditional notation (b) Compact notation

Figure 3.4: Sample graph transformation rule

are deleted, elements that are present only in RHS are created, and other model elements remain unchanged. Moreover, further actions can be initiated by calling any ASM rules within the action part of a GT rule, e.g. to report debug information or to generate code. This action part is executed *after* the model manipulation part is carried out according to the difference of the precondition and postcondition part.

```
import UML;
gtrule liftAttrsR(inout CP, inout CS, inout A) =
{
   precondition pattern lhs(CP,CS,A,Par,Attr) =
   {
      Class(CP);
      Class.parent(Par,CS,CP);
      Class(CS);
      Class.attrs(Attr,CS,A);
      Attribute(A);
   }
   postcondition pattern rhs(CP,CS,A,Par,Attr,Attr2) =
   {
      Class(CP);
      Class.parent(Par,CS,CP);
      Class(CS);
      Class.attrs(Attr2,CP,A);
      Attribute(A);
   }
   action {
      print("Rule liftAttrR is applied on attribute " + name(A));
   }
}
```

Negative conditions are also commonly used in precondition patterns, especially, for model transformations between modeling languages in order to prevent the application of a GT rule twice on the same matching, and to enable incremental transformations.

3.3.5 Informal semantics of graph transformation rules

Parameter passing

A main difference with the traditional GT notation is related to the use of parameter passing preconditions and postconditions. More precisely, matchings of the precondition pattern are passed to the postcondition via pattern parameters, which act as an explicit interface between the precondition and the postcondition.

- All the parameters of the postcondition which is (i) also a precondition parameter or (ii) passed to the entire GT rule as an

input parameter are treated as *input parameters* for the postcondition. Note that a simple lexical match decides if a precondition parameter also appears as a postcondition parameter regardless of the order of parameters. These parameters are already bound before calculating the effects of the postcondition. In our example, input parameters of the postcondition are CP, CS, A, Par and Attr.

- 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 of our example is Attr2.

On the one hand, we can reduce the size of the patterns with respect to traditional GT rules by information hiding. For instance, precondition elements which are left enact by the rule need not be passed to the postcondition, which is very convenient for large patterns.

The negative side of this solution is that the execution mechanism of a GT rule becomes slightly more complex than in case of traditional GT rules. More specifically, the postcondition of a GT rule may prescribe three different operations on the model space.

- *Preservation.* If an input parameter of the postcondition appears in the pattern body, then the matching model element is preserved. The elements matched by variables CP, CS, and A are thus preserved above.
- *Deletion.* If an input parameter of the postcondition does not appear in the body of the postcondition pattern then the corresponding model element is deleted. For instance, element matched by variable Attr is deleted.
- *Creation.* If a variable which appears in the body of the postcondition pattern 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. In our example above, variable Attr2 is not an input parameter of the postcondition, thus it prescribes the creation of a new attrs relation between class CP and attribute A. This Attr2 is an output parameter of the postcondition but not passed back to the GT rule itself.

Based upon these principles, rule liftAttrsR can be further compacted by simply omitting the parent relation from the postcondition and from the pattern parameter lists as well.

Pattern calls in GT rules

In order to reduce the size and to support the modular creation of GT rules, pattern calls (i.e. the find construct) are allowed in both the precondition and postcondition pattern. Its use in the precondition pattern was already discussed in Sec. 3.3.2.

However, pattern calls in the postcondition (RHS) of GT rules is a unique feature compared to other GT tools. Currently, VIATRA-I handles non-recursive and non-negative calls in the postcondition pattern, which allows a macro-like substitution of the called pattern in the body of the postcondition. This way, repetitive parts of a postcondition can be modularized into predefined patterns, which can be used in various GT rules afterwards.

3.3.6 Generic and meta-transformations

To provide algorithm-level reuse for common transformation algorithms independent of a certain metamodel, VIATRA-I supports generic and meta-transformations, which are built on the multilevel metamodeling support. For instance, we may generalize rule liftAttrsR as lifting something (e.g. an Attribute) one level up along a certain relation (e.g. parent). The following example is the generic equivalent of the previous GT rule parameterized by types taken from arbitrary metamodels during execution time.

```
gtrule liftUp(inout CP, inout CS, inout A, inout ClsE, inout AttE, inout ParR,
    inout AttR) = {
   precondition pattern lhs(CP,CS,A, ClsE, AttE, ParR, AttR, Attr) =  {
      // Pattern on the meta-level
      entity(ClsE);
      entity(AttE);
      relation(ParR,ClsE,ClsE);
      relation(AttR,ClsE,AttE);
      // Pattern on the model-level
      entity(CP);
      // Dynamic type checking
      instanceOf(CP,ClsE);
      entity(CS);
      instanceOf(CS,ClsE);
      entity(A);
      instanceOf(A,AttE);
      relation(Par,CS,CP);
      instanceOf(Par,ParR);
   postcondition pattern rhs(CP,CS,A, ClsE, AttE, ParR, AttR, Attr) =  {
      // Pattern on the meta-level
      entity(ClsE);
      entity(AttE);
      relation(ParR,ClsE,ClsE);
      relation(AttR,ClsE,AttE);
      // Pattern on the model-level
      entity(CP);
      // Dynamic type checking
      instanceOf(CP,ClsE);
      entity(CS);
      instanceOf(CS,ClsE);
      entity(A);
      instanceOf(A,AttE);
      relation(Par,CS,CP);
      instanceOf(Par,ParR);
      // relation (Attr, CS, A) removed as not present in RHS
      relation(Attr2,CP,A);
      instanceOf(Attr2,AttR);
   }
}
```

Compared to liftAttrsR, this generic rule has four additional input parameters: (i) ClsE for the type of the nodes containing the thing to be lifted (Class previously), (ii) AttE for the type of nodes to be lifted (Attribute previously), and (iii) ParR (ex-parent) and (iv) AttR (ex-attrs) for the edge types.

When interpreting this generic pattern, the VIATRA-I engine first binds the type variables (ClsE, ParR, etc.) to types in the metamodel of a modeling language and then queries the instances of these types. Internally, this is carried out by treating subtype-of and instance-of relationships as special edges in the model space, which enables the easy generalization of traditional graph pattern matching algorithms.

3.3.7 Invoking graph transformation rules

The basic invocation of a graph transformation rule is done using the apply keyword within a choose or a forall construct. In each case, the actual parameter list of the transformation has to contain a valid value for all input parameters, and an unbound variable for all output parameters.

A rule can be executed for all possible matches (in a single, pseudo-parallel step) by quantifying some of the parameters using the forall construct. Finally, a GT rule can be applied as long as possible by combining the iterate and the choose constructs. The following example illustrates some possible invocations of our sample rule.

```
//execution of a GT rule for one attribute of a class
//variables Class1 and Class2 must be bound
choose A apply liftAttrsR(Class1,Class2,A);

//calling the rule for all attributes of a class
//variables Class1 and Class2 must be bound
forall A do apply liftAttrsR(Class1,Class2,A);

//calling the rule for all possible matches in parallel
forall C1, C2, A do apply liftAttrsR(C1,C2,A);
//Apply a GT rule as long as possible for the entire model space
iterate
    choose C1, C2, A apply liftAttrsR(C1,C2,A)
```

Note the difference between the *as long as possible* and the *forall* execution modes: the former applies the rule once and only then does it select a next match as long as a match exists, while the latter collects all matches first, and then it applies the rule for each of them one by one in a single compound step.

3.3.8 Control Structure

To control the execution order and mode of graph transformation the VTCL language includes language constructs that support the definition of complex control flow. As one of the main goals of the development of VTCL was to create a precise formal language, we included the basic set of Abstract State Machine (ASM) language

constructs [1] that correspond to the constructs in conventional programming languages.

The basic elements of an ASM program are the rules (that are analogous with methods in OO languages), variables, and ASM functions. ASM functions are special mathematical functions, which store values in associative arrays (dictionaries). These values can be updated by ASM rules.

In VTCL, a special class of functions, called *native function*s, is also defined. Native functions are user-defined Java methods that can be called from the transformations. These methods can access any Java library (including database access, network functions, and so on), and also the VIATRA model space. This allows the implementation of complex calculations during the execution of model transformations.

ASMs provide complex model transformations with commonly used control structures in the form of built-in ASM rules, which are overviewed in Fig. 3.5.

```
// variable definition
let X = 1 in ruleA
// variable (and ASM function) updates
update X = X + 1;
// print and log rules print a term to standard output or into the log
print("Print X: " + X + "\n");
log(info, "Log X: " + X);
// conditional branching by a logical condition or by pattern matching
if (X>1) ruleA else ruleB
if (find myPattern(X)) ruleA else ruleB
// exception handling: rule2 is executed only if rule1 fails
try rule1 else rule2
// calls the user defined ASM rule myRule with actual parameter X
call myRule(X)
// the sequencing operator: executes its subrules in the given order;
seq { rule1; rule2; }
// executes a non-deterministically selected rule from a set of rules
random { rule1; rule2; }
// iterative execution by applying rule1 as long as possible
iterate rule1;
//executes rule1 for a (non-deterministic) substitution of variable X
//which satisfies the pattern (or location) condition with X
choose X below M with (myAsmFun(X) > 0) do rule1
choose X below M with find myPattern(X) do rule1
//pseudo-parallel execution of rule1 for all substitution of variable X
//which satisfies the pattern (or location) condition with X
forall X below M with (myAsmFun(X) > 0) do rule1
forall X below M with find myPattern(X) do rule1
```

Figure 3.5: Overview of built-in ASM rules in VIATRA-I

As a summary, ASMs provide control structures including the sequencing operator (seq), rule calls to other ASM rules (call), variable declarations and updates (let and update constructs) and if-then-else structures, non-deterministically selected (random) and executed rules (choose), iterative execution (applying a rule as long as possible iterate), and the deterministic parallel rule application at all possible matchings (locations) satisfying a condition (forall).

In addition to these core ASM rules, the VIATRA-I dialect of ASMs also includes built-in rules for manipulating the model space. As a re-

sult, elementary model transformation steps can be specified either in a declarative way (by graph transformation rules) or in an imperative way by built-in model manipulation rules. Main model manipulation rules are summarized in Fig. 3.6.

```
// create a new entity C of type class and place it inside M
// the local name of C is automatically generated
new (class(C) in M);
// rename class C to "Product"
rename(C, "Product");
// create a new relation Attr of type attrs between C and A
// Attr is placed under its source C
new(attrs(Attr, C, A));
// Explicitly moves entity C (and all of its contents) to NewContainer
move(C, NewContainer);
// Retargets relation Attr to A1
  setTo(Attr, A1);
// copy entity C and all of its contents directly under Container together with
// ALL incoming and outgoing relations; the entity is accessed by CNew
copy(C, Container, CNew, keep_edges);
// copy entity C and all of its contents directly under Container but
// only copy relations between entities of the containment subtree of C
copy(C, Container, CNew, drop_edges);
// removes model element M together with its contents
delete(M);
```

Figure 3.6: Overview of model manipulation rules in VIATRA-I

Most of these rules are rather straightforward, we only give more insight into the copy and move rules. The copy rule aims at copying an entity and all the recursive contents (i.e. the subtree of the entity) to a new parent. VIATRA-I provides two kinds of semantics for that copy operation: keep_edges and drop_edges. In the first case, all relations leading out from or into an entity placed anywhere below the copied entity are copied as well. In the latter case, only those relations are copied where both the source and the target entity is below the copied entity. In case of the move rule, an entity is moved (by force) to a new container by decoupling it from its old container. While this step may break the invariants of the old container, this problem is not as critical as in case of EMF as our constraints are checked at the end of the transformation.

These basic built-in ASM rules, combined with graph patterns and graph transformation rules, form an expressive, easy-to-use, yet mathematically precise language where the semantics of graph transformation rules are also given as ASM programs (see [6] for more details). The following example demonstrates some of the main control structures.

```
// An ASM rule is defined using the 'rule' keyword
rule main(in Model) =
 // Conversion from strings to model elements
 let M = ref(Model) in seq {
   //Print out some text
   print("The transformation of model " + M + " has started...\n");
   //Find all top-level classes below M and call rule printTopLevel
   forall C1 below M with find isTopClass(C1) do
      call printTopLevel(C1);
   //Apply a GT rule as long as possible for the entire model space
   iterate
          choose C1, C2, A apply liftAttrsR(C1,C2,A) do
             print("Attribute "+ name(A) + " is lifted from class " + C1 +
                   " to class " + C2 +"\n");
```

```
    //Write to log
    log(info,"Transformation terminated successfully.");
}
rule printTopLevel(in C) =
{
    print("Class " + name(C) + " is a top-level class.\n");
}
```

4 Formal Semantics of Graph Patterns in VTCL

As the main conceptual novelties of the VIATRA-I framework are related to the rich pattern language with recursive (and non-recursive) pattern calls, and arbitrary depth of negation, below we provide a formalization of the semantics of this crucial part. In order to avoid lengthy formal descriptions, we disregard from the formalization of other aspects of the language, like ASM or GT rules, which already have a rich theoretical background, and VIATRA-I 's contribution are less significant. For instance, a formal semantics of ASMs are given in [1], while an ASM formalization of GT rules is listed in [6].

4.1 Formal representation of VPM models

A formal logic representation can be easily derived for VPM models. The vocabulary (signature) of VPM models can be derived directly from the Prolog-like syntax to include (i) *predicates* (i.e. boolean function symbols) entity/1, relation/3, supertypeOf/2, instanceOf/2, in/2, below, and (ii) *function symbols* value/1 (the value of an entity) fqn/1, name/1 (the FQN and local name of a model element), and (iii) traditional *boolean and arithmetic function symbols*. A *state of the model space* (denoted as $\mathcal{A}$) can be defined by an evaluation of these predicates and function symbols.

- *Entity.* A predicate $\mathsf{entity}(v)$ is evaluated to true in state $\mathcal{A}$ (denoted as $[\![\mathsf{entity}(v)]\!]^{\mathcal{A}}$), if entity uniquely identified by v exists in the model space in state $\mathcal{A}$. Otherwise the predicate is evaluated to false: $[\![\neg\mathsf{entity}(v)]\!]^{\mathcal{A}}$.
- *Relation.* A predicate $\mathsf{relation}(v, s, t)$ is true (denoted as $[\![\mathsf{relation}(v, s, t)]\!]^{\mathcal{A}}$), if the relation identified by v exists in the model space in state $\mathcal{A}$, and it leads from model element s to model element t. Otherwise, $[\![\neg\mathsf{relation}(v, s, t)]\!]^{\mathcal{A}}$.
- *SupertypeOf* A predicate $\mathsf{supertypeOf}(sup, sub)$ is evaluated as true (denoted as $[\![\mathsf{supertypeOf}(sup, sub)]\!]^{\mathcal{A}}$), if sup is a supertype of sub in the model space in state $\mathcal{A}$. Otherwise, $[\![\neg\mathsf{supertypeOf}(sup, sub)]\!]^{\mathcal{A}}$.
- *InstanceOf* A predicate $\mathsf{instanceOf}(ins, typ)$ is evaluated as true (denoted as $[\![\mathsf{instanceOf}(ins, typ)]\!]^{\mathcal{A}}$), if ins is instance of typ in the model space in state $\mathcal{A}$. Otherwise, $[\![\neg\mathsf{instanceOf}(ins, typ)]\!]^{\mathcal{A}}$.

- *In* A predicate $\mathsf{in}(chi, par)$ is evaluated as true (denoted as $[\![\mathsf{in}(chi, par)]\!]^{\mathcal{A}}$), if *chi* is directly contained by *par* in the model space in state $\mathcal{A}$. Otherwise, $[\![\neg\mathsf{in}(chi, par)]\!]^{\mathcal{A}}$. Furthermore, we define predicate $\mathsf{below}(chi, anc)$ as the reflexive and transitive closure of *in*.

VTML predicates of the form *type(id)* are treated as $\mathsf{entity}(type) \wedge \mathsf{entity}(name) \wedge \mathsf{instanceOf}(()id, type)$ while typed relations are handled accordingly. Furthermore, we assume the existence of (an infinite pool) of fresh object identifiers, which will be assigned to newly created model elements. Finally, well-formedness rules of VPM models are formalized by axioms in [6].

4.2 Semantics of graph pattern matching

The formal semantics of graph patterns in VTCL will be defined as the set of all matches of a given pattern in a model space (see Table 4.1). For this purpose, VTCL patterns are first translated into a logic program (i.e. Prolog-like predicates). Then the semantics of this logic program will be defined as all the solutions which will be derived using standard relation database operations (like selection, filtering, natural join, etc.).[1].

Parameter passing. The semantics of a pattern is defined with respect to a given model space $\mathcal{A}$ and a set of input parameters passed to the pattern, which is encoded as a selection criterion $F_0 = \bigwedge_i X_i = v_i$.

Elementary predicates. First of all (Rows 1-3 in Table 4.1, the semantics of *elementary VPM predicates* (i.e. entity, relation, etc.) is simply the set of all corresponding tuples of the model space filtered by the selection criterion F_0 to restrict the result to the input parameters.

Simple patterns. Then a *simple pattern* (Row 4) is represented as a conjunction of VPM predicates. The semantics of a simple pattern is defined as the inner join of matches retrieved by each predicate. In order to resolve name clashes of variables prior to the inner join operation, we rename all conflicting variable names by introducing new variables and extending the selection criterion F with corresponding equality constraints of (previously clashing) variables. For instance, if variable X is clashing, and thus it is renamed to variable Y, an equality constraint $X = Y$ is added to F.

Non-recursive calls. In case of non-recursive pattern calls (Row 5), we simply derive the inner join of each (non-recursive) pattern call by appropriate variable renaming as above.

[1] We assume that the reader is familiar with the basic database operations.

	VTCL grammar (simplified)	Derived predicates	Semantics
1	$fact$ = entity(V) (or relation(V,S,T))	$fact(V) \leftarrow \mathsf{entity}(V)$ $fact(V,S,T) \leftarrow \mathsf{relation}(V,S,T)$	$[\![fact(V)]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_F(\{v \mid [\![\mathsf{entity}(v)]\!]^{\mathcal{A}}\})$ $[\![fact(V,S,T)]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_F(\{(v,s,t) \mid [\![\mathsf{relation}(v,s,t)]\!]^{\mathcal{A}}\})$
2	$fact$ = supertypeOf(A,B) (or instanceOf(A,B))	$fact(A,B) \leftarrow \mathsf{supertypeOf}(A,B)$ (or instanceOf(A,B))	$[\![fact(A,B)]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_F(\{(a,b) \mid [\![\mathsf{supertypeOf}(a,b)]\!]^{\mathcal{A}}\})$ (or instanceOf(a,b))
3	$fact$ = in(A,B) (or below(A,B))	$fact(A,B) \leftarrow \mathsf{in}(A,B)$ (or below(A,B))	$[\![fact(A,B)]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_F(\{(a,b) \mid [\![\mathsf{in}(a,b)]\!]^{\mathcal{A}}\})$ (or below(a,b))
4	$simple = fact_1 \ldots fact_n$	$simple(\overline{X}) \leftarrow$ $fact_1(\overline{X_1}) \wedge \ldots \wedge fact_1(\overline{X_n})$	$[\![simple(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![fact_1(\overline{Y_1})]\!]_{F_1}^{\mathcal{A}}) \overset{F_1}{\bowtie} \ldots \overset{F_1}{\bowtie} [\![fact_n(\overline{Y_n})]\!]_{F_1}^{\mathcal{A}})$ where $F_1 = F \wedge \bigwedge_k X_{i,k} = Y_{j,k}$ for each variable renaming.
5	$poscall = simple$ **find (** $base_1$ **);** ... **find (** $base_n$ **);**	$poscall(\overline{X}) \leftarrow$ $simple(X_0) \wedge \bigwedge_j base_j(\overline{X_j})$	$[\![poscall(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![simple(\overline{Y_0})]\!]_{F_1}^{\mathcal{A}}) \overset{F_1 F_1}{\bowtie\bowtie} [\![base_j(\overline{Y_j})]\!]_{F_1}^{\mathcal{A}})$ with $F_1 = F \wedge \bigwedge_k (X_k = Y_k)$ for each var. renaming.
6	neg = **neg find (** $\{base \mid recur\}$ **);**	$neg(\overline{X}) \leftarrow base(\overline{X})$	$[\![neg(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![base(\overline{X})]\!]_F^{\mathcal{A}}$
7	$base = poscall\, neg_1, \ldots, neg_n$	$base(\overline{X}) \leftarrow poscall(\overline{X_0}) \wedge \bigwedge_j (\neg neg_j(\overline{X_j}))$	$[\![base(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} \sigma_{F_2}([\![poscall(\overline{Y_0})]\!]_{F_1}^{\mathcal{A}} \overset{F_1}{\ltimes} (\overset{F_1}{\ltimes} [\![neg_j(\overline{Y_j})]\!]_{F_1}^{\mathcal{A}}))$ where $F_1 = F \wedge \bigwedge_k X_k = Y_k$ for each variable renaming, and $F_2 = F_1 \wedge \bigwedge Y_{k,ns} = \varepsilon$ for non-shared variables of patterns neg_j and $poscall$.
8	$recur = base$ **find (** $recur_1$ **);** ... **find (** $recur_n$ **);**	$recur(\overline{X}) \leftarrow base(\overline{X_0}) \wedge recur_1(\overline{X_1}) \wedge \ldots \wedge recur_n(\overline{X_n})$	$[\![recur(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} lfp([\![base(\overline{Y_0})]\!]_{F_1}^{\mathcal{A}} \overset{F_1}{\bowtie} [\![recur_1(\overline{Y_1})]\!]_{F_1}^{\mathcal{A}} \overset{F_1}{\bowtie} \ldots \overset{F_1}{\bowtie} [\![recur_n(\overline{Y_n})]\!]_{F_1}^{\mathcal{A}})$ where $F_1 = F \wedge \bigwedge_k X_k = Y_k$ for each variable renaming
9	$body = recur$ (**check** $cond$)?	$body(\overline{X}) \leftarrow recur(\overline{X}) \wedge check(\overline{X})$	$[\![body(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![recur(\overline{X})]\!]_{F_1}^{\mathcal{A}}$ where $F_1 = F \wedge check(\overline{X})$
10	**pattern** $patt$ = { $body_1$ } **or** { $body_2$ }	$patt(\overline{X}) \leftarrow body_1(\overline{X}) \vee body_2(\overline{X})$	$[\![patt(\overline{X})]\!]_F^{\mathcal{A}} \stackrel{def}{=} [\![body_1(\overline{X})]\!]_F^{\mathcal{A}} \cup [\![body_2(\overline{X})]\!]_F^{\mathcal{A}}$

Table 4.1: Deriving predicates for VTCL patterns

Negative pattern calls. Negative pattern calls (Rows 6-7) are syntactically very close to ordinary pattern calls, however, their semantic treatment is essentially different. First we assume that there are there are no recursive calls alternating between negative and positive patterns. To be more precise, if the positive patterns transitively called by a pattern p are denoted by the set P while the set of patterns transitively called by a negative pattern of is denoted by N then $P \cup N = \emptyset$.

After that the left outer join of the positive pattern *poscall* and the negative patterns is calculated. Successful matchings of the pattern are identified as rows where all non-shared variables of the negative patterns neg_j (i.e. non-shared with the positive pattern *poscall*) take the NULL value ($Y_{k,ns} = \varepsilon$).

Recursive pattern calls. In VTCL, arbitrary recursive calls are allowed (as long as recursion and negation is not alternating), which is a rich functionality. Obviously, we require the presence of a *base pattern* which is a simple pattern extended with non-recursive calls and negative patterns. The semantics of recursive calls is defined in a bottom up way as the least fix point of the inner join operation $[\![base]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} [\![recur_1]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} \ldots \overset{F_1}{\bowtie} [\![recur_n]\!]_F^{\mathcal{A}})$. Initially, $[\![recur_i^{(0)}]\!]_F^{\mathcal{A}} = \emptyset$ for all i. Then in the first iteration, it collects all matchings derived by the base cases of recursive patterns, i.e. $[\![recur_i^{(1)}]\!]_F^{\mathcal{A}} = [\![base_i]\!]_F^{\mathcal{A}}$ (i.e. its own base case). In all upcoming iterations, $[\![recur^(i+1)]\!]_F^{\mathcal{A}}$ is defined as $[\![base]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} [\![recur_1^{(i)}]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} \ldots \overset{F_1}{\bowtie} [\![recur_n^{(i)}]\!]_F^{\mathcal{A}}$ with the appropriate variable renaming. This step is executed until a fixpoint is reached, which might cause non-termination for ill-formed programs.

Check condition and OR patterns. Check conditions (Row 9) in a pattern simply extend to the selection criteria F, while the result of OR-patterns (Row 10) is defined as the union of the results for each pattern body.

4.3 Semantics of calling graph patterns from ASMs

In case of model transformations, graph patterns are called from ASM programs by (i) supplying input parameters, and (ii) defining whether pattern matching should be initiated in choose or forall mode by quantifying free variables of the pattern. Note that the same pattern can be called with different *adornment* (i.e. variable binding). For instance, once a pattern parameter can be an input parameter, while at a different location, it can be quantified by the forall construct.

When using the choose construct to initiate pattern matching, the free variables will be substituted by *existential quantification*, i.e. only

one (non-deterministically selected) matching will be retrieved from result set of the pattern.

However, if the pattern is called using the forall construct, this means *universal quantification* for the pattern parameters (head variables), thus all possible values of the head variables that satisfy the pattern will be retrieved, and the ASM body of the forall rule will be executed on each pattern one by one.

Finally, patterns may also contain internal variables which appear in the body but not in the formal parameter list (head), which variables are quantified existentially. Note that universally quantified variables take precedence over existentially quantified ones, i.e. we try to find one substitution of existentially quantified variables for each valid substitution of universally quantified ones.

Formalization. In order to formalize this behavior, let $\overline{X} = \overline{X^{in}} \cup \overline{X^f} \cup \overline{X^b}$ denote all the variables in the body of a pattern *patt* where $\overline{X^{in}}$ is supplied as input parameters (with assignments $X_i^{in} = v_i$ that constitute the initial filtering condition F), $\overline{X^f}$ denote the free variables in the pattern head, while $\overline{X^b}$ denote the variables appearing only in the pattern body.

Then the semantics (i.e. result set) retrieved by each construct is defined by projecting the result set to the columns of pattern variables only. As a consequence, variables of the body $\overline{X^b}$ are always evaluated in an existential way, no matter how many matchings they have.

- $[\![$**choose** $\overline{X^f}$**with find** $patt(\overline{X^{in}} \cup \overline{X^f})]\!]_F^{\mathcal{A}} \stackrel{def}{=}$
  any $v \in \pi_{\overline{X^{in}} \cup \overline{X^f}}([\![patt(\overline{X})]\!]_F^{\mathcal{A}})$,
  i.e. *any value* in the result set projected to columns of the pattern variables only. If $[\![patt(\overline{X})]\!]_F^{\mathcal{A}} = \emptyset$, then the choose construct becomes inconsistent to cause backtracking in ASM.
- $[\![$**forall** $\overline{X^f}$**with find** $patt(\overline{X^{in}} \cup \overline{X^f})]\!]_F^{\mathcal{A}} \stackrel{def}{=} \pi_{\overline{X^{in}} \cup \overline{X^f}}([\![patt(\overline{X})]\!]_F^{\mathcal{A}})$,
  i.e. the (possibly empty) result set projected to the columns of pattern variables only.

Finally, note that while a forall rule collects all the matches for a pattern in a single step, its body executed sequentially one by one on the matches. For the moment, only partial checks are carried out during run-time to detect conflicts in the execution of different bodies in a forall rule.

4.4 Notes on algorithmic considerations

Note that this formal semantics, which is quite close to that of used in various deductive databases [5], also highlights the main strategy of the implementation of pattern matching in VIATRA-I . Unfortunately,

the actual algorithms used in VIATRA-I are far too complicated, thus their detailed description is out of scope for the current paper. Below, we only highlight the main novelties.

Since pattern predicates are not ordered in VIATRA-I (nor in deductive databases), their proper ordering is a crucial step for the overall performance of pattern matching. The VIATRA-I implementation improves performance compared to deductive databases by carrying out global optimization for predicate ordering by flattening pattern nonrecursive pattern calls using a generic search plan representation [3].

The costs of elementary matching operations can be defined based on the metamodel (i.e. navigating along a relation with at-most-one multiplicity is cheaper than along an arbitrary multiplicity), or using adaptive pattern matching with model-specific search plans [8].

In case of recursive calls, we build on *magic sets* [5], a well-known technique for deductive databases, which combines bottom-up fixpoint evaluation (by iteratively extending existing matchings based on the merging of globally optimized flattened patterns until a fixpoint is reached) with a top-down strategy for input parameters.

5 Viatra Textual Command Language Specification

The Viatra Textual Command Language (VTCL) is the primary transformation definition platform for VIATRA. Developers can use this language for the definition of metamodels, model transformations and code generators. The language offers both graph transformation rule and Abstract State Machine (ASM) rule definition capabilities. This chapter introduces the constructs of the language.

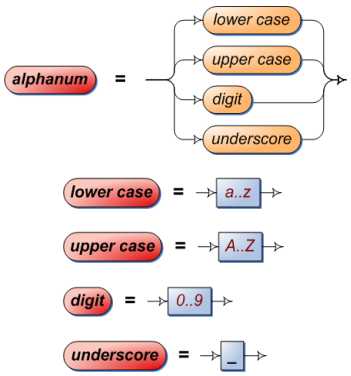
5.1 Naming and Coding Conventions

5.1.1 Base Elements

Alphanumerical characters

Description *alphanum* represents the alphanumerical characters. In general, names, variables and type names are composed of these characters.

Syntax



Semantics An alphanumerical character can be a lowercase letter (*lower case*), an uppercase letter (*upper case*), a numerical digit (*digit*) or an underscore (*underscore*).

See also *name*, *variable*, *type name*

Arbitrary Characters

Description In strings representing names, variables, type names or string constant almost an arbitrary character can appear. These are defined here.

Syntax

Semantics *anychar I.* can be an arbitrary ASCII character except the single quotation mark (’). Names, variables and type names can be composed of these between single quotation marks.

anychar I. can be an arbitrary ASCII character except

- backslash (\ - 0x5C),
- double quotation mark (” - 0x22),
- newline character (\n - 0x0D),
- linefeed character (\r - 0x0A).

String constants can be composed of these between double quotation marks.

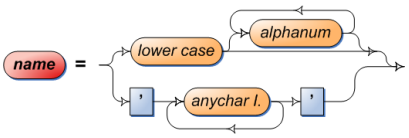
See also *name*, *variable*, *type name*, *string constant*

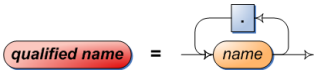
5.1.2 Names and Types

Name

Description VTCL related model elements such as machines, GT rules, GT patterns, action patterns, ASM functions and ASM rules are identified by their names.

Syntax





Semantics A *name* begins with a lowercase letter that can be followed by an arbitrary number of alphanumerical characters. A name can contain other or special characters, too, but in that case it should be inserted between single quotation marks. The single quotation marks are not parts of the name.

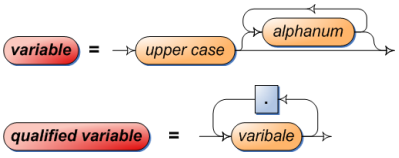
Since a *name* is an identifier for VTCL related model elements, it is also part of the containment hierarchy. Thus, the name in itself –which is often referred as *short name*– can be used when referencing the related model element only if its namespace is imported or the reference is in the same namespace. In all other cases the VTCL related model element must be referenced with its *fully qualified name*. A fully qualified name is a *qualified name* that consists of all containers of a model element according to the containment hierarchy starting from a child of the root element. A *qualified name* is a sequence of *name*s separated with dots.

See also *asm def*, *gtrule def*, *gtpattern def*, *action pattern def*, *asmfunction def*, *asmrule def*, *namespace def*

Variable

Description A *variable* can be a pattern variable or an ASM variable. A *pattern variable* must refer to a model element (either entity or relation), whereas an *ASM variable* can contain the result of an arbitrary ASM term.

Syntax



Semantics A *variable* begins with an uppercase letter that can be followed by an arbitrary number of alphanumerical characters.

When referencing pattern variables in the definition of a relation or relationship in a pattern body or action pattern body it can be necessary to qualify a variable with other variables if its definition was

nested to the definition of the other variables (i.e. as part of the *contained elements* of other entities). A *qualified variable* is composed of the names of the variables separated with dots.

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, for instance.

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

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. As a result, ASM variables that are undefined within a certain scope cause compilation errors.

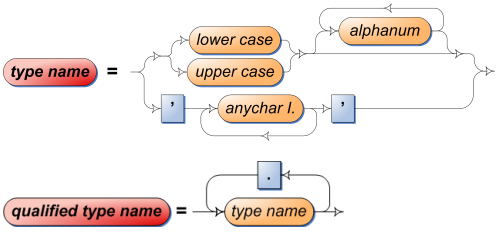
Constraints ASM variables cannot be redefined within an internal scope.

See also *model element*, *relation*, *relationship*, *contained elements*

Type Name

Description A type name is the name of a model element.

Syntax



Semantics A model element is identified by its name called *type name*. A *type name* can contain arbitrary ASCII characters except the single quotation mark since the sequence of characters must be inserted between single quotation marks. The single quotation marks are not parts of the name. The quotation marks can be omitted if the name starts with a letter (either upper case or lower case) and contains only alphanumerical characters.

The type name is the *short name* of the model element; the *fully qualified name* is a *qualified type name* that consists of all containers of the model element according to the containment hierarchy starting from a child of the root element. A *qualified type name* is a sequence of *type name*s separated with dots.

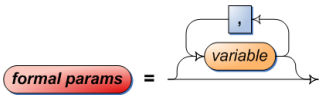
Remarks Note that name constants starting with an upper-case initial may clash with variables in ASM programs. In case of such a conflict, the name constants have to be surrounded with apostrophes (').

5.1.3 Parameters and Scopes

Formal Parameters

Description The interface of the GT patterns and action patterns are defined by the formal parameters, i.e. by a list of pattern variables.

Syntax



Semantics *formal params* is a list of pattern variables separated by comas. It can contain any number of variables including zero. In contrast to the *directed formal params* it does not define the direction of the parameters. The variables can be used both for input and output purposes; it depends on the usage, i.e. on the invoking environment. Those variables that are bound before the invocation of the pattern are the input variables and those that are unbound are output variables. As mentioned above, these two sets can change from invocation to invocation.

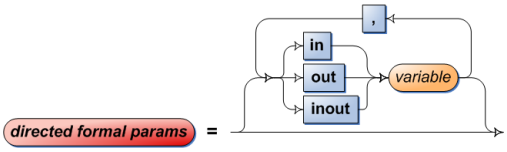
Used by *gtpattern def*, *action pattern def*

See also *directed formal params*, *actual params*

Directed Formal Parameters

Description The interface of the GT rules and ASM rules are defined by directed formal parameters, i.e. by a list of pattern variables associated with a direction.

Syntax



Semantics *directed formal params* is a list of direction tag–pattern variable pairs separated by comas. It can contain any number of variables including zero. In contrast to the *formal params* it defines the direction of the parameters. It can be either ***in***, ***out*** or ***inout***.

A pure input variable (with tag ***in***) takes the value of the actual parameter, but at the end of the execution of the rule the value of the actual parameter is not updated.

A pure output variable (with tag ***out***) does not take the value of the actual parameter: it is considered as undefined having the *undefined value* ***undef***. At the end of the execution of the rule the value of the actual parameter is overwritten with the value of the variable defined as formal parameter.

An input-output variable (with tag ***inout***) takes the value of the actual parameter, and at the end of the execution of the rule the value of the actual parameter is updated with the value of the variable defined as formal parameter.

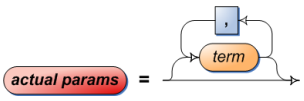
Used by *gtrule def*, *asmrule def*

See also *formal params*, *actual params*, *undefined value*

Actual Parameters

Description Actual parameters describe variables, values and formulas that are associated with the formal parameter list of rules or patterns during their invocation.

Syntax



Semantics *actual params* is a comma separated list of ASM terms that are associated with the formal parameters of a pattern or rules.

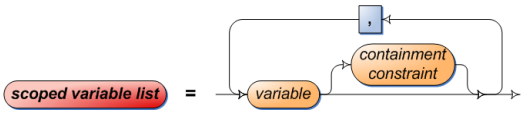
In case of GT or ASM rules the formal parameters have direction; this implicates that actual parameters that are associated with output formal parameters can be only variables. Actual parameters that are associated with pure input formal parameters can be an arbitrary ASM *term*.

Used by *gtrule call*, *gtpattern call*, *action pattern call*, *rule invocation rule*

Scoped Variable List

Description A *scoped variable list* is a list of variables where the variables contains model elements and the possible values of variables are restricted with respect to their location in the containment hierarchy.

Syntax



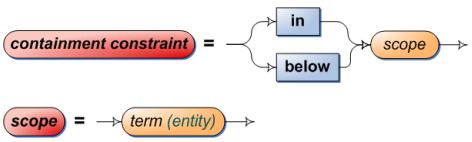
Semantics *scoped variable list*s are used in a *choose rule* or in a *forall rule*, i.e. the variables of the list are quantized with the existential or with the universal quantor. These variables can contain only model elements and the *containment constraint* restricts the possible location of model elements with respect to their location in the containment hierarchy.

Used by *choose rule*, *forall rule*

Containment Constraint

Description A containment constraint applies to the domain of a variable that refers to an entity. It restricts the variable domain with respect to the containment hierarchy.

Syntax



Semantics The *containment constraint* is formulated with a special model ele-

ment called *scope*. The *scope* defines an entity in the modelspace in an arbitrary way, i.e. it is an ASM *term* that evaluates to an entity (for instance it can be a direct model element reference or a variable that refers to an entity).

In case of the ***in*** keyword the domain of the constrained variable is restricted to the elements that are directly contained by the scope, i.e. the domain contains the children of the entity defined by the scope.

In case of the ***below*** keyword the domain of the constrained variable is restricted to the elements that are directly or indirectly contained by the scope, i.e. the domain contains the children of the entity defined by the scope and the children of the children in an arbitrary depth.

CONSTRAINTS A containment constraint can be applied only for variables that refer to entities (i.e. it cannot be applied for variables that refer to relations!).

USED BY *scoped variable list*, *entity*

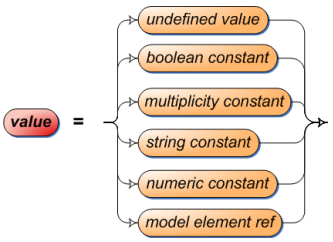
SEE ALSO *variable*

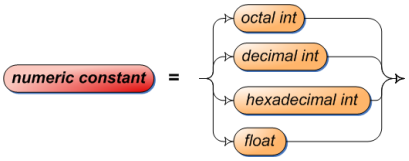
5.1.4 Values and Constants

VALUE

DESCRIPTION The possible values described here determines the domain of the variables. In addition variables can be assigned to entities.

SYNTAX





SEMANTICS A *value* can be an *undefined value*

- an *undefined value*,

- a constant (*boolean constant*, *multiplicity constant*, *string constant*, *numeric constant*),
- a reference to a model element (*model element ref*).

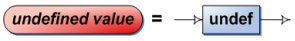
A numeric constant can be either an integer value with different representation forms (*octal int*, *decimal int*, *hexadecimal int*) or a floating point value (*float*).

Used by *term*

Undefined Value

Description Undefined value is a concrete value; a variable has this value if it is known that the value of the variable is not known.

Syntax



Semantics The undefined value is identified by the ***undef*** keyword.

A pattern variable has the undefined value if it is unbound.

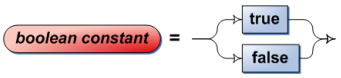
Remarks It can also be used in the *variable definition rule* if it does not make sense to assign a value to the variable at the moment of the definition.

Defined in *value*

Boolean Constant

Description Variables can store boolean values – these are defined by boolean constants. It can be used also in ASM logical terms.

Syntax



Semantics A boolean constant can have two value: ***true*** and ***false***.

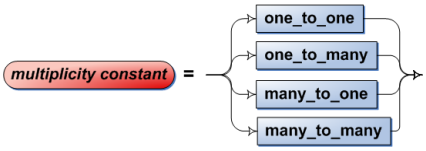
Defined in *value*

Used by *base logical term*

Multiplicity Constant

Description A property of a relation is its multiplicity. The multiplicity constants determine the domain of this property.

Syntax



Semantics Both end of a relation can have either the multiplicity *one* or the multiplicity *many*. In the modelspace only a single property determines the multiplicity of a relation that holds the multiplicity information for both ends. The possible values of a *multiplicity constant* are ***one_to_one***, ***one_to_many***, ***many_to_one*** and ***many_to_many***.

Defined in *value*

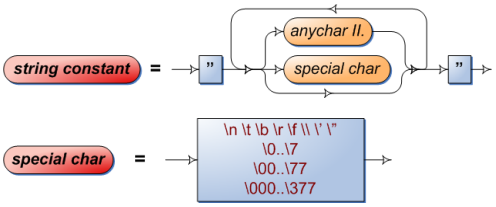
Used by *element update rule* (***setMultiplicity***),

built in function (***getMultiplicity***)

String Constant

Description The value property of entities are represented by strings, variables can store string values, too and strings can be the elements of arithmetic terms. The representation of strings are defined by string constants.

Syntax



Semantics A string is a sequence of characters. In the constant representation it is surrounded with double quotation marks (**"**). A character can be a normal ASCII character defined by *anychar II.* or a special character defined by *special char*.

A *special char* is not a single ASCII character; it is defined by a pair, triplet or quartet of characters, see Table 5.1.

Table 5.1: Definition of special characters in string constants

Special character	*Description*
\"	a double quotation mark character
\\	a backslash character
\n	a new line character
\t	a tabulator character
\b	a backspace character
\r	a carriage return character
\f	a formfeed character
\0..\7 \00..\77 \000..\377	an ASCII character with its code

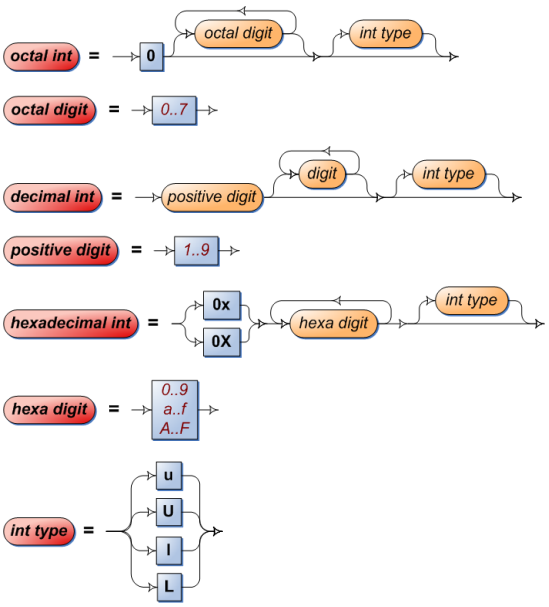
Defined in *value*

See also *entity value*, *element update rule* (***setValue***), *variable*, *arithmetic term*

Integer Constants

Description Integer values can be represented as octal, decimal or hexadecimal constants.

Syntax



Semantics The different representations of integers are differentiated by their first characters:

- octal integers start with a ***0*** character that is followed by an *octal digit*,
- decimal integers start with a positive digit (***1***..***9***),
- hexadecimal integers start with a ***0*** character that is followed by a small or large *X* (***x***/***X***).

The first character or characters are followed by the possible numerical characters:

- ***0***..***7*** for octal integers,
- ***0***..***9*** for decimal integers,
- ***0***..***9***, ***a***..***f***, ***A***..***F*** for hexadecimal integers.

Each representation of an integer can have three different types related to the internal representation:

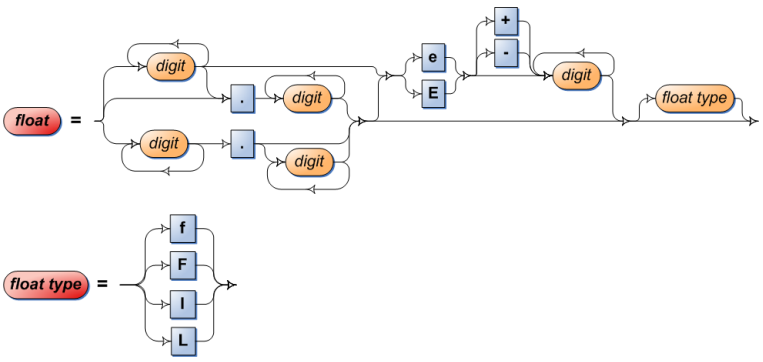
- *normal* (there is no type modifier at the end of the number),
- *unsigned* (there is a ***u*** or ***U*** character at the end of the number),
- *long* (there is an ***l*** or ***L*** character at the end of the number).

Defined in *value – numeric constant*

Float Constant

Description A float is a type of values that variables can store. Float values can be represented explicitly as float constants.

Syntax



Semantics A float constant can be represented in an arbitrary combination of

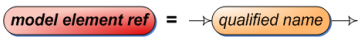
the floating point representation and the scientific representation.

Defined in *value – numeric constant*

Model Element Reference

Description A model element reference is a type of values that variables can store. Typically, pattern variables store model element references.

Syntax



Semantics A *model element ref* can reference either an entity or a relation in the actual modelspace. It is formulated as a *qualified name*: it can be either the short or the fully qualified name of the referenced model element.

Defined in *value*

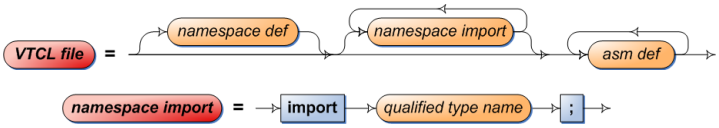
5.2 Main Structure

5.2.1 The VTCL File

VTCL file

Description A VTCL file contains the definition of one or more Abstract State Machines (ASM) in the default or in the explicitly defined namespace. Other namespaces can be imported for direct accessibility of the elements defined there.

Syntax



Semantics Namespaces are identified by *qualified type names*.

If a namespace is defined explicitly all ASMs defined in the file are interpreted as parts of this namespace, otherwise ASMs are part of the default namespace. The default namespace for a *VTCL file* is the *root namespace*, which is identified by the empty identifier.

Arbitrary number of other namespaces can also be imported; in this case all model elements defined in these namespaces can be referenced with their simple names (last part of the fully qualified name) inside of all the machines defined in the file.

Remarks The *vpm* and *vpm.entity* namespaces are imported implicitly to every VTCL file.

Namespace Definition

Description Like in modern programming languages, namespaces are used also in VTCL. Namespaces makes it easier to identify model elements uniquely in a hierarchical way.

Syntax

Semantics Namespaces are identified by *qualified type name*s.

The namespace structure corresponds to the containment hierarchy of the VPM model space. Model elements can be referenced either by their fully qualified names or by their short names (identifiers) if the namespace in which they are contained is reachable (either because it is the currently defined namespace or because it is imported). Import instructions must be placed at the beginning of the file.

Constraints All element names must be unique within the scope of their direct containers.

Remarks The recommended structure of namespaces is to divide the root namespace into technological spaces (like ASM, UML, BPM, MOF etc.). In these technological namespaces define the following containers (Fig. 5.1):

- *metamodel* for containing the metamodel of the given technology,
- *models* for all the models that are instances of the above mentioned metamodel,
- *patterns* for containing the frequently used metamodel related patterns,
- *transformations* for transformations that either modifies the models in the given technological space or transforms these to other technological spaces,

- *references* for all the additional information that is related to transformations transforming to other technological spaces organized into subcontainers:
  - *metamodels* for the metamodels of reference structures that defines the relationship between the elements of the metamodels of the current and the target technology,
  - *patterns* for patterns that identifies the corresponding model elements in the different technological spaces based on the reference structure,
  - concrete reference models grouped by the target models.

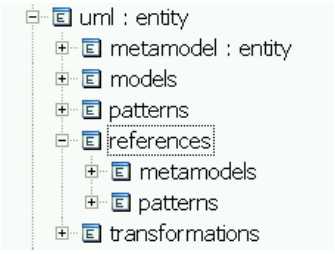


Figure 5.1: Recommended structure of a technological namespace

This structure clearly separates the various types of metamodels, models, transformations and other related model elements.

EXAMPLE An example container structure can be seen in Fig. 5.2 of the UML technological space with a transformation to the VMM technological space and with the related reference elements.

SEE ALSO *VTCL file*.

5.2.2 Machine Definition

GRAPH TRANSFORMATION ABSTRACT STATE MACHINE DEFINITION

DESCRIPTION A Graph Transformation Abstract State Machine (GTASM) is the unit of execution. It consists of ASM rules, ASM functions, graph transformation rules, graph transformation patterns and action patterns.

SYNTAX

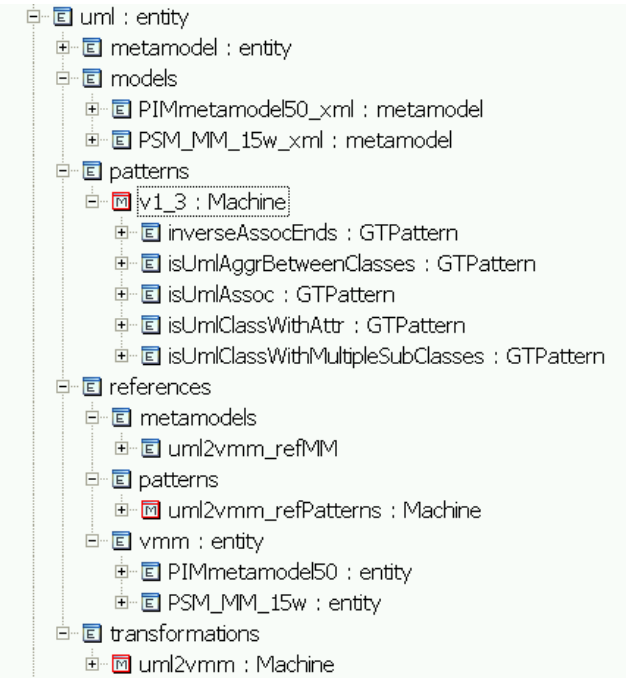
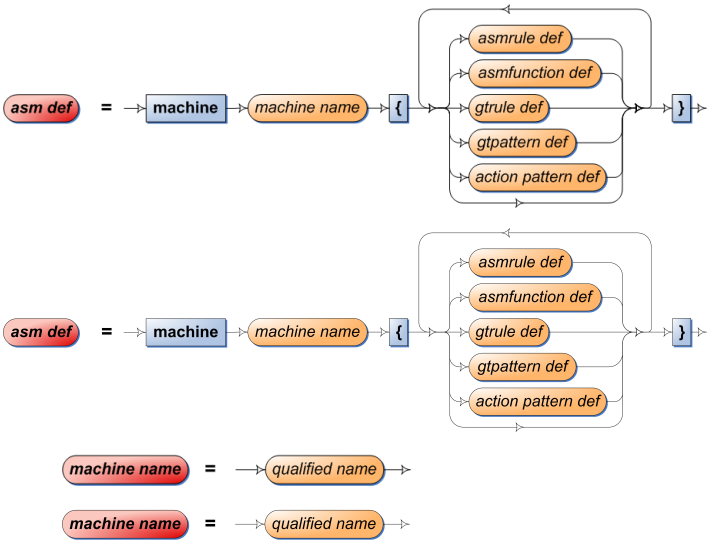


Figure 5.2: Example structure of a technological namespace



SEMANTICS

A GTASM is identified by its name; the *machine name* is a *qualified name*, whereas the fully qualified name of a machine is composed of the namespace defined in the container *VTCL file* and this qualified name.

A GTASM contains two types of components: abstract state machine (ASM) related components and graph transformation (GT) related components.

An ASM related component can be a rule or a function; the set of these compose the control flow part of the GTASM. The entry point of the control flow is a special rule called *main* just similarly to the C language.

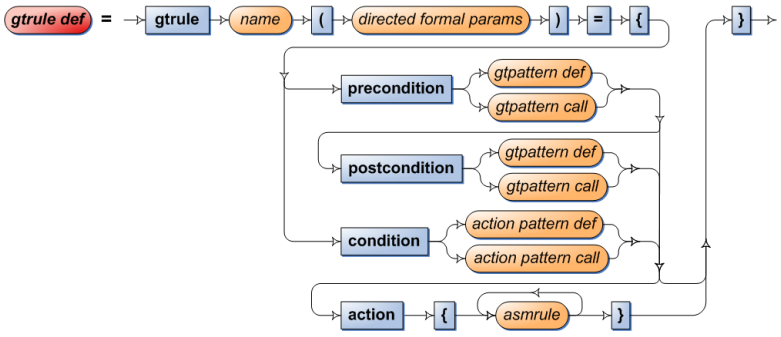
A graph transformation related component can be a transformation rule or a pattern; the set of these compose the model manipulation part of the GTASM. Patterns have two types: a gtpattern defines a pattern that is used just for searching the modelspace, whereas the action pattern extends it with model manipulation features. The later is used only in the context of a graph transformation rule, whereas a simple pattern can also be the part of an ASM rule.

5.3 Graph Transformation Rules

Graph Transformation Rule Definition

Description 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.

Syntax



Semantics *name* is the identifier of the graph transformation rule.

Variables defined as part of the *directed formal params* are the input or output parameters of the rule depending on its type (***in***/***out***/***inout***). Parameters can be bound before the rule was called; these bindings are kept in case of input parameters (***in***/***inout***). Pure output parameters (that has the tag ***out***) are always handled as

unbound variables. Unbound variables are bound either in the ***precondition*** or in the ***condition*** part of the rule.

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 this case the model manipulation is defined by the difference of the precondition and the postcondition.
- Using the ***condition*** part of the rule instead of the pre- and postcondition parts represents a more compact way for defining GT rules. The condition pattern is the union of the pre- and postcondition patterns; in this case the model manipulation actions (i.e. the difference between the pre- and the postcondition) are indicated directly with *gtrule action* tags.
- In VIATRA-I , there is also an ***action*** part where additional side-effects for rules can be defined with ASM rules. Among other things model manipulation can also be realized here.

Thus, a GT rule can be composed of

- a ***precondition***, a ***postcondition*** and an optional ***action*** part,
- a ***condition*** and an optional ***action*** part,
- a ***precondition*** and an ***action*** part.

Each of the precondition, postcondition and condition parts of the GT rule can be defined either inline or somewhere else, i.e. the appropriate keyword is followed by either a pattern definition or a pattern call (action patterns are extensions of normal patterns).

The parameters of the patterns or pattern calls can be input or output parameters. *Input parameters* are those variables that are input parameters of the GT rule or an arbitrary parameter of a preceding part of the GT rule (i.e. for instance an output parameter of a precondition can be the input of the postcondition). All others are called *output parameters* of the given part of the rule, i.e. those that are output parameters of the GT rule or have not been defined before the given part. Output parameters are bound as the direct effect of the given part.

Precondition The precondition part –if present– defines the condition for the application of the GT rule. Input parameters of this pattern

can be bound or unbound before the execution of this part. Execution of a precondition means pattern matching, thus after it all parameters are bound.

Postcondition The postcondition pattern describes what conditions should hold as result of applying the GT rule. Input parameters of this pattern have to be bound before the execution of this part.

The result of GT rule application is calculated as the difference of the postcondition and precondition. 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.
- *Deletion.* If an input parameter of the postcondition does not appear in the postcondition pattern itself then the matching model element is 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.

Condition Condition part consists of both pattern matching and model manipulation. *Local pattern body element*s in the *action pattern body* can have three states after the execution of the condition:

- *Preserved* if an element does not have a *gtrule action* tag before it. These elements are only parts of the pattern matching.
- *Deleted* if it has ***del*** *gtrule action* tag before it. These elements are parts of the pattern matching and are deleted after it from the model.
- *Created* if it has ***new*** *gtrule action* tag before it. These elements are NOT parts of the pattern matching, but are created after it in the model.

Action After pattern matching and a non-compulsory model manipulation the GT rule may execute a sequence of additional actions defined as ordinary ASM rules.

All parameters of both the precondition and the postcondition or of the condition can be used in the action part, but internal

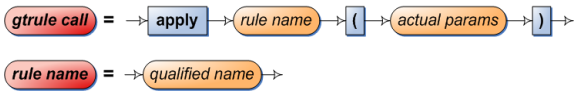
pattern variables (i.e. those that does not appear as parameter of the pattern) are not visible. Of course, all parameters of the GT rule itself can also be used in the action part.

See also *choose rule*, *forall rule*, *gtrule call*.

GTRule Call

Description Graph transformation rules can be invoked by using the *choose* or *forall* ASM rules to execute model manipulation.

Syntax



Semantics The rule is identified by the *rule name*. It can be the short name of the rule if

- the rule is defined in the same file or
- the rule is defined in another file but the namespace of the rule is imported at the beginning of the file.

If the rule is defined in another file and its namespace is not imported then the *rule name* should be the fully qualified name of the rule.

The *actual params* part is the list of actual parameters.

During the execution of the rule variables are getting bound according to the precondition or condition part of the rule and model manipulation is performed according to the postcondition or condition and the action parts.

The call fails, if there are no matches.

Constraints Pure input parameters of the rule –i.e. the parameters that are tagged with ***in*** in the formal definition of the parameter list– can be an arbitrary ASM *term*, but output parameters must be variables.

All variables mentioned in the *actual params* part have to be already defined before the invocation of the rule, although these can be unbound.

See also *Namespace definition*, *choose rule*, *forall rule*,

5.4 Graph Patterns

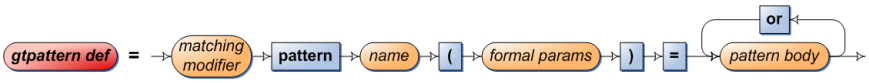
5.4.1 Pattern Definition

GTPattern Definition

Description A graph transformation pattern is the tool/leverage/mean for searching the model space. Graph patterns provide a way for defining logic formulae for the model structure.

Graph patterns can be defined either stand-alone or as part of a graph transformation rule or as part of a negative pattern in the body of another pattern.

Syntax



Semantics *name* is the identifier of the pattern.

formal params is the list of formal 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.

The *matching modifier* controls if the variables in the pattern body or bodies are bound in injective or non-injective way.

A graph pattern definition consists of at least one *pattern body* definition. If multiple bodies are present, there is an or condition between them (if at least one of the bodies can be matched, the pattern is considered to be matched).

See also *asm def*, *gtrule def*, *negative pattern*

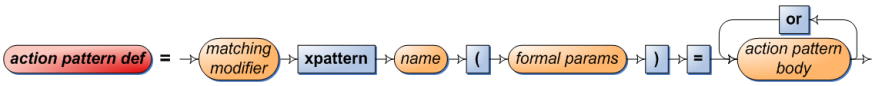
Action Pattern Definition

Description An action pattern is an extension of graph patterns. It is considered in the context of graph transformation rules: in such a pattern model

manipulation related operations (*creation* and *deletion* of model elements or relationships) are indicated explicitly.

Action patterns can be defined either stand-alone or inline in a GT rule.

Syntax



Semantics *name* is the identifier of the action pattern.

formal params is the list of formal 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.

The *matching modifier* controls if the variables in the action pattern body or bodies are bound in injective or non-injective way.

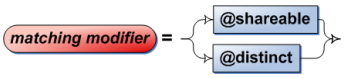
An action pattern definition consists of at least one *action pattern body* definition. If multiple bodies are present, there is an or condition between them (if at least one of the bodies can be matched, the pattern is considered to be matched and the corresponding model manipulation operations are executed).

See also *gtpattern def*

Matching Modifier

Description The matching modifier is interpreted in the context of a single graph pattern (either normal or action pattern). It determines if the variables in the body or bodies of the pattern are bound in injective or non-injective way.

Syntax



Semantics The keyword ***@distinct*** identifies the *injective* variable bounding

method. In this case different variables (i.e. those that have different names) *cannot* be bound to the same model element.

The keyword **@sharable** identifies the *non-injective* variable bounding method. In this case different variables (i.e. those that have different names) *can* be bound to the same model element.

See also *gtpattern def*, *action pattern def*

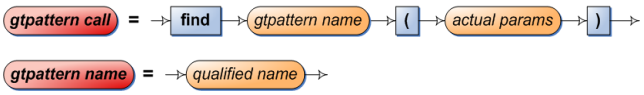
GTPattern Call

Description

A graph transformation pattern can be called in the following contexts:

- as the precondition or postcondition part of a *GT rule definition*,
- from inside another GT or action pattern, i.e. as part of a *pattern body* or *action pattern body*,
- as part of a *negative pattern*,
- where *base logical terms* can be used.

Syntax



Semantics

A GT pattern is identified by *gtpattern name*. It can be the short name of the GT pattern if

- that is defined in the same file or
- that is defined in another file but the namespace of the pattern is imported at the beginning of the file.

If the pattern is defined in another file and its namespace is not imported then the *gtpattern name* should be the fully qualified name of the pattern.

The *actual params* part is the list of actual parameters. Variable parameters evaluating to model elements can be both bound or unbound.

During the call of the pattern constraints defined by the pattern are checked on the parameters. If unbound variables are present in the *actual params* list then these are getting bound some way to satisfy the constraints if possible. If it is not possible or parameters do not

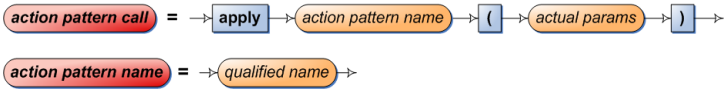
satisfy the constraints then pattern matching fails and returns *false* value. If pattern matching succeeds the returned value is *true*.

See also *GT rule def*, *pattern body*, *action pattern body*, *negative pattern*, *base logical terms*

Action Pattern Call

Description An action pattern can be called only in the context of a GT rule. It is called either directly as the condition part of a *GT rule definition* or indirectly from inside another action pattern, i.e. as part of an *action pattern body*.

Syntax



Semantics An action pattern is identified by *action pattern name*. It can be the short name of the action pattern if

- that is defined in the same file or
- that is defined in another file but the namespace of the pattern is imported at the beginning of the file.

If the pattern is defined in another file and its namespace is not imported then the *action pattern name* should be the fully qualified name of the pattern.

The *actual params* part is the list of actual parameters. Variable parameters evaluating to model elements can be both bound or unbound.

During the call of the pattern constraints defined by the pattern are checked on the parameters. If unbound variables are present in the *actual params* list then these are getting bound some way to satisfy the constraints if possible. If it is not possible or parameters do not satisfy the constraints then pattern matching fails. If *pattern matching succeeds* the model manipulation is performed as part of the execution of the graph transformation rule that initiated the call of the action pattern (either directly or indirectly through another action pattern). If *pattern matching fails* the corresponding GT rule also fails.

See also *GT rule def*, *action pattern body*

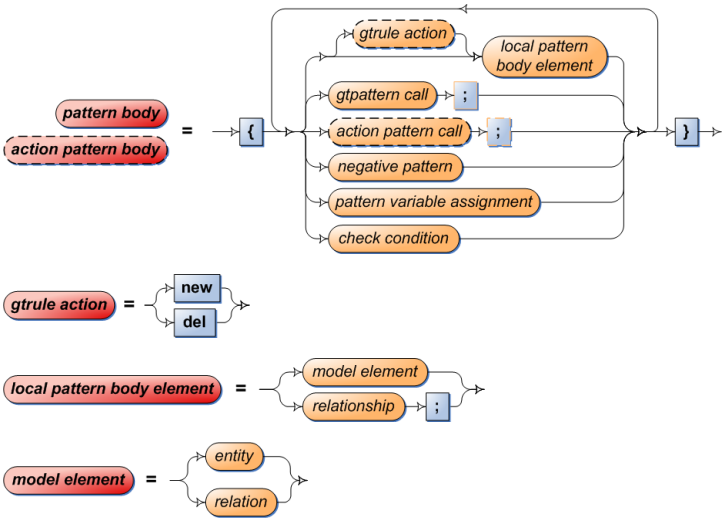
Pattern Body and Action Pattern Body

Description

Pattern body is the basic part of the *gtpattern definition*, whereas *action pattern body* is the basic part of the *action pattern definition*. These are constructed very similarly – that is why these are considered together. However, *action pattern body* contains some model manipulation related extensions compared to the base *pattern body*.

Both determine a constraint set that identifies a part of the modelspace. In addition, *action pattern body* defines how this part should be modified.

Syntax



Semantics

In the syntax notation the elements that have dashed border line are only part of the *action pattern body*. All other elements are part of both bodies. An *action pattern body* specific element appears during the definition of the subelements, see *contained elements*.

A *pattern body* mainly consist of *local pattern body element*s. These defines the typed structure of the pattern. A *local pattern body element* can be either a *model element* or a *relationship* definition between model elements. A *model element* is either an *entity* or a *relation*. *Model element*s are identified by variables: these are either the parameters of the pattern or internal variables.

In an *action pattern body local pattern body elements* can be tagged with *gtrule action*s ***new*** and ***del***. In pattern matching only those elements are taken into account that have no tag or have the ***del*** tag. After pattern matching as part of the execution of the initiator GT

rule those elements and relationships are deleted that have the ***del*** tag and those are created that have the ***new*** tag.

From within a *pattern body* another pattern can be invoked; within an *action pattern body* also an *action pattern* can be called. In these cases the constraints or actions defined by the invoked patterns are added to the locally defined constraints and actions.

A negative pattern defines negative constraints: the pattern body can match only if these are not satisfied.

Within a *pattern body* the equality of variables can be defined: it is called *pattern variable assignment*.

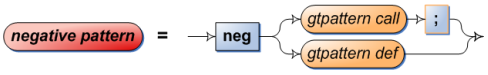
Extra conditions can be states in addition to those that defines the typed structure. These are called *check condition*s and are formulated by ASM terms.

See also *gtpattern def*, *action pattern def*

Negative Pattern

Description Negative patterns define negative constraints in a *pattern body* or in an *action pattern body*: if these can be matched then the match of the container pattern fails.

Syntax



Semantics A *negative pattern* is composed of the ***neg*** keyword and a pattern definition or pattern call.

See also *pattern body*, *action pattern body*

Pattern Variable Assignment

Description A *pattern variable assignment* is part of a *pattern body* or *action pattern body*. It defines the equality of two locally visible variable.

Syntax

Semantics If two pattern variable is declared to be equal these are assigned to the same *model element*.

See also *pattern body*, *action pattern body*

Check Condition

Description A *check condition* is part of a *pattern body* or *action pattern body*. It defines a boolean condition as an ASM term that must be true in order the pattern match to succeed.

Syntax

Semantics A *check condition* is composed of the ***check*** keyword and an ASM *term* in braces. The ASM *term* can contain those variables that are parameters or are defined as a *model element*. It can invoke ASM functions, too.

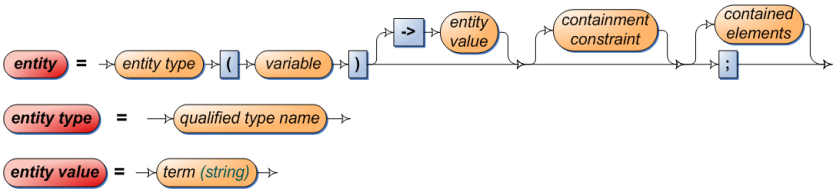
See also *pattern body*, *action pattern body*

5.4.2 Entity Description

Entity

Description An *entity* is a model element. It is the base component of the VPM; a modelspace is composed of typed entities and relations between them.

Syntax



Semantics The description of an *entity* can appear in the context of a *pattern body* or *action pattern body*. It either references an existing entity if it is used in pattern matching or defines a new entity if it has a ***new*** tag before it in the *action pattern body* or it is in the difference of a postcondition and precondition.

It is identified by a pattern variable (*variable*).

It can have a type (*entity type*), which is either a short name or a fully qualified name. A type identifies another entity of the modelspace; it can be referenced with the short name only if its namespace is imported. In the simplest case the type is ***entity***, which is also part of the modelspace (its FQN is *vpm.entity*). It can be referenced with its short name since the *vpm* namespace is imported implicitly to every VTCL file.

An entity can have a value (*entity value*). Optionally, it can be the part of the description of the entity after the *variable* prefixed by a textual arrow (**-¿**). If *entity value* is omitted it can be anything during pattern matching. If the description is the definition of the entity its value is set to the *empty string* if *entity value* is omitted.

The description of an entity can contain a *containment constraint*. During pattern matching the referenced entity must be located in the containment hierarchy according to the constraint. In case of the ***in*** constraint the entity must be contained directly by the entity referenced by the scope, whereas in case of the ***below*** constraint the containment can be recursive. If the description is the definition of the entity it will be created as the child of the entity defined by the *scope* regardless of that the constraint is ***in*** or ***below***.

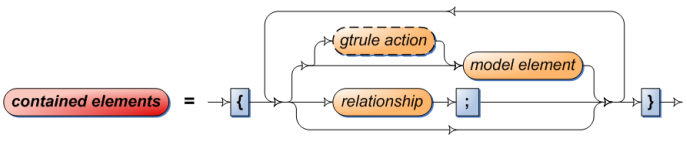
An entity can be defined either as a single entity or together with its *contained elements*.

See also *model element*, *pattern body*, *action pattern body*, *local pattern body element*, *import namespace*

Contained Elements

Description In the modelspace an entity can contain other entities; it is described by the containment hierarchy. In the description of an *entity* it can be taken into account either by a *containment constraint* or by listing the *contained elements* explicitly.

Syntax



Semantics Szerintem ez felesleges, mivel van containment constraint. De legalbbis a relation-t s relationship-et itt megengedni.

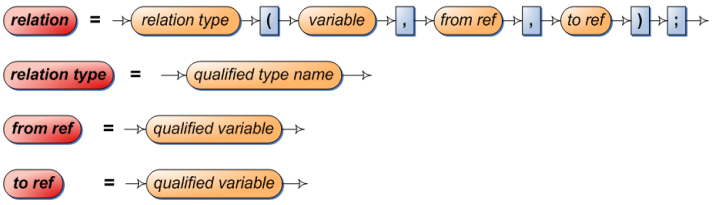
See also *entity*

5.4.3 Relation Description

Relation

Description A *relation* is a model element. It is one of the base components of the VPM; a modelspace is composed of typed entities and relations between them.

Syntax



Semantics The description of a *relation* can appear in the context of a *pattern body* or *action pattern body*. It either references an existing relation if it is used in pattern matching or defines a new relation if it has a ***new*** tag before it in the *action pattern body* or it is in the difference of a postcondition and precondition.

It is identified by a pattern variable (*variable*).

It can have a type (*relation type*), which is either a short name or a fully qualified name. A type identifies another relation of the modelspace; it can be referenced with the short name only if its namespace is imported. In the simplest case the type is ***relation***, which is also part of the modelspace (its FQN is *vpm.entity.relation*). It can be referenced with its short name since the *vpm.entity* namespace is imported implicitly to every VTCL file.

The description of a relation contains two references to model elements . The first is the source of the relation (*from ref*) and the second is the target of the relation (*to ref*). Both are identified by pattern variables. These variables are qualified by other variables if these are defined as part of those as *contained elements*. Both source and target of a relation can be an entity or another relation.

Remarks The containment of a relation cannot be defined or constrained explicitly since every relation is contained by its source model element.

See also *model element*, *entity*, *pattern body*, *action pattern body*, *local pattern body element*, *import namespace*

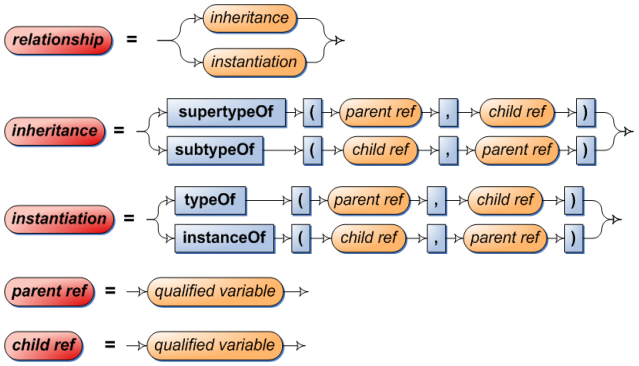
5.4.4 Relationship Description

RELATIONSHIPS

DESCRIPTION

A relationship is a special relation that is model independent and is defined as part of the VPM. It describes either inheritance (i.e. supertype-subtype) or instantiation (i.e. type-instance) relation.

SYNTAX



SEMANTICS

The description of a *relationship* can appear in the context of a *pattern body* or *action pattern body*. It either references an existing relationship if it is used in pattern matching or defines a new relationship if it has a ***new*** tag before it in the *action pattern body* or it is in the difference of a postcondition and precondition.

A *relationship* is either an *inheritance* or an *instantiation*.

In each case two model element is referenced by pattern variables: a parent (*parent ref*) and a child (*child ref*). These variables are qualified by other variables if these are defined as part of those as *contained elements*.

In case of inheritance the parent is the *supertype* and the child is the *subtype*. Both ***supertypeOf*** and ***subtypeOf*** refers to the same relationship; only the order of the parameters are different.

In case of instantiation the parent is the *type* and the child is the *instance*. Both ***typeOf*** and ***instanceOf*** refers to the same relationship; only the order of the parameters are different.

In case of both relationships both the parent and the child can be an entity or a relation.

CONSTRAINTS

The parent and the child of the relationships must refer to the same

type of model element, i.e. either both refer to entities or both refer to relations.

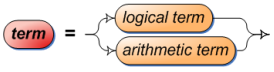
See also *pattern body*, *action pattern body*, *local pattern body element*, *entity*, *contained elements*

5.5 ASM Terms and Formulas

ASM Term

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

Syntax



Semantics ASM expressions are constructed in the traditional way from constants, variables and ASM functions. Currently, only constants have types (string, boolean, integer, double or model element reference) that are evaluated at compile-time, while the type of variables and ASM functions are determined dynamically at run-time.

Operators for different types are defined as follows:

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

An ASM expression, i.e. a *term* is either a *logical term* or an *arithmetic term*.

Remarks If an expression is evaluated to a type where a used operator is not supported (e.g. “str1” || “str2” is invalid), a run-time exception is thrown.

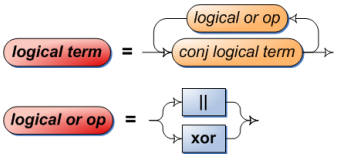
Used by *actual params*, *scope*, *check condition*, *entity value*, *location*, *value tag*, *update rule*, *print rule*, *log rule*, *variable definition rule*, *choose rule*, *forall rule*, *if rule*, *create rule*, *delete rule*, *copy rule*, *move rule*, *element update rule*

5.5.1 Logical Terms

Logical Term

Description A *logical term* is an ASM term that evaluates to a boolean value. It can be either a simple or a compound expression.

Syntax



Semantics A *logical term* can be a compound expression, i.e. it can be composed of *conj logical term*s with a *logical or op*.

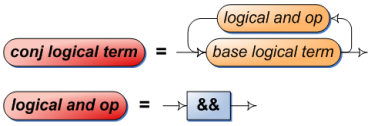
The *logical or op* can be either the || (OR) or the ***xor*** operator; these are interpreted in the classical way.

Defined in *term*

Conjunctive Logical Term

Description A *conj logical term* is an ASM term that evaluates to a boolean value. It can be either a simple or a compound expression.

Syntax



Semantics A *conj logical term* can be a compound expression, i.e. it can be composed of *base logical term*s with a *logical and op*.

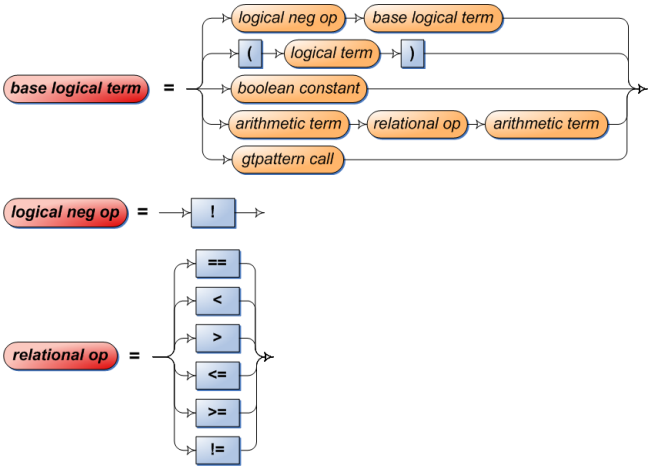
The *logical and op* is the ***&&*** (AND) operator; it is interpreted in the classical way.

Defined in *logical term*

Base Logical Term

Description A *base logical term* is an ASM term that evaluates to a boolean value. It can be either a simple or a compound expression.

Syntax



Semantics A *base logical term* can be the followings:

- a *boolean constant*,
- the negation of a *base logical term*,
- a general *logical term* surrounded by parenthesis,
- a relational expression or
- a *gtpattern call*.

A *relational expression* is a relation between two *arithmetic terms*; the relational operator can be ==, >, <, >=, <= or ! =. Operators for different types are defined as follows:

- String operators: $<, >, ==, !=, <=, >=$
- Integer and float operators: $<, >, ==, !=, <=, >=$
- Boolean operators: $==, !=$
- Multiplicity operators: $==, !=$
- Name operators $==, !=$

A *gtpattern call* results ***true*** if a matching is found and ***false*** otherwise.

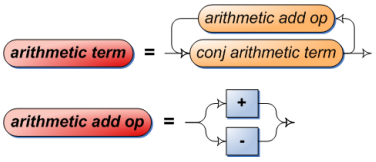
Defined in *conj logical term*

5.5.2 Arithmetic Terms

Arithmetic Term

Description An *arithmetic term* is an ASM term that evaluates to an arbitrary value (string, integer, float, boolean, model element ref). It can be either a simple or a compound expression.

Syntax



Semantics An *arithmetic term* can be a compound expression, i.e. it can be composed of *conj arithmetic term*s with an *arithmetic add op*, but not for all types.

The *arithmetic add op* can be

- the + operator; it is allowed
  - for integers and floats; the interpretation is the classical (i.e. addition),
  - for strings; it is interpreted as concatenation,
- the - operator; it is allowed
  - for integers and floats; the interpretation is the classical (i.e. substraction).

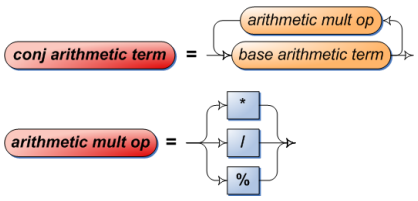
Defined in *term*

Used by *base logical term*

Conjunctive Arithmetic Term

Description A *conj arithmetic term* is an ASM term that evaluates to an arbitrary value (string, integer, float, boolean, model element ref). It can be either a simple or a compound expression.

Syntax



Semantics A *conj arithmetic term* can be a compound expression, i.e. it can be composed of *base arithmetic term*s with an *arithmetic mult op*, but only in case of numerical types (for integers and floats).

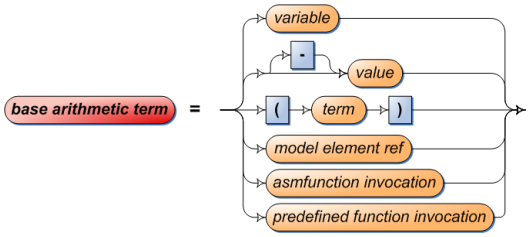
The *arithmetic mult op* can be either the *, the / or the % operator; these are interpreted in the classical way; i.e. as multiplication, division or remainder calculation.

Defined in *arithmetic term*

Base Arithmetic Term

Description A *base arithmetic term* is an ASM term that evaluates to an arbitrary value (string, integer, float, boolean, model element ref). It can be either a simple or a compound expression.

Syntax



Semantics A *base arithmetic term* can be the followings:

- a *variable*,
- a *value*,
- the negative value of a *numeric constant*,
- a general *term* surrounded by parenthesis,
- a *model element ref*,
- an *asmfunction invocation* or
- a *predefined function invocation*.

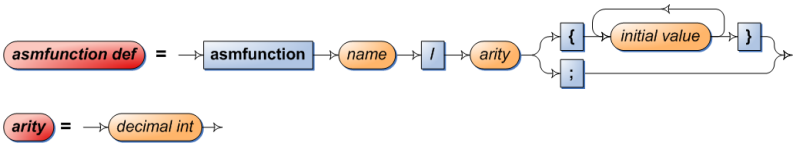
Defined in *conj arithmetic term*

5.5.3 ASM Functions

ASM Function Definition

Description In terms of structured programming languages, an *ASM function* is an associative array (dictionary, map, etc.), which stores data *value*s at *location*s (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).

Syntax



Semantics An ***asmfunction*** is identified by its *name*. In the definition of an ASM function the number of index elements, i.e. the *arity* must be determined explicitly. In addition, we may initially provide partial definitions of some slots initially storing some values at certain locations (*initial value*). Unused locations of arrays are implicitly storing the *undef* value.

Mathematically, an ASM function is a partial mapping from its index domain (called location or slot) to its value domain.

An ASM function have to be declared at the global scope (that is, at machine scope, outside any rules) prior to its first use.

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).

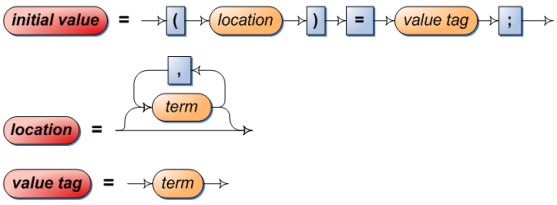
Defined in *asm def*

See also *asmfunction invocation*, *update rule*

Initial Value

Description An *initial value* can be the part of the definition of an ASM function. It provides the definition of a slot initially storing a value at a certain location.

Syntax



Semantics An *initial value* assigns a value to a *location*. The value is determined by the evaluation of the *value tag*, which can be an arbitrary ASM *term*.

The *location* is determined by the comma separated list of *index elements*, that can be arbitrary ASM *terms*.

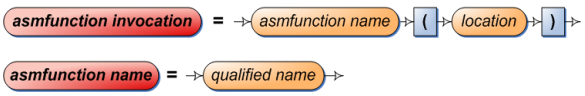
Constraints The number of the index elements must be equal to the arity determined by the definition of the ASM function.

Defined in *asmfunction def*

ASM Function Invocation

Description A slot of an ASM function, i.e. a value stored at a specific location determined by an ASM function can be reached by an *asmfunction invocation*.

Syntax



Semantics In the *asmfunction invocation* the ASM function is identified by its short or fully qualified name; in addition, a specific *location* is defined.

The *asmfunction invocation* returns the value associated with the given location. If no value has been explicitly associated to the location then the ***undef*** value is returned.

USED BY *update rule*, *base arithmetic term*

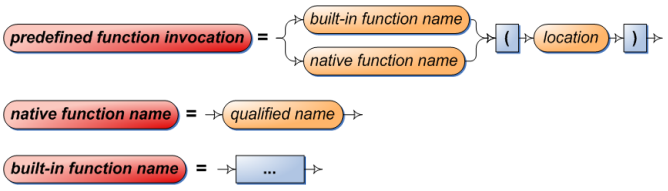
SEE ALSO *asmfunction def*

5.5.4 Predefined Functions

PREDEFINED FUNCTION INVOCATION

DESCRIPTION VIATRA-I provides a set of predefined functions: a part of it is built-in function that is interpreted by the VIATRA-I engine and another part is defined as an Eclipse plugin; these later ones are called *native functions*.

SYNTAX



SEMANTICS The predefined functions are identified by their names and are called with an actual parameter list.

For *native functions* the name and the formal parameter list is determined by the Eclipse plugin.

Built-in functions are the followings:

- *Element reference functions*
  - **ref(s)** - This function returns a reference to the element with the name in *s*.
  - **fqn(e)** - Returns the fully qualified name of the parameter element *e*.
  - **name(e)**- Returns the local (short) name of the parameter element *e*.
  - **value(e)**- Returns the value associated to the parameter entity *e*.
  - **source(r)** - Returns the source element of the relation *r*.
  - **target(r)** - Returns the target element of the relation *r*.
- *Conversion functions*

- **toBoolean(r)** - Converts the contained value to boolean (if possible).
- **toString(r)** - Converts the contained value to string (if possible).
- **toInteger(r)** - Converts the contained value to integer number (if possible).
- **toDouble(r)** - Converts the contained value to double precision floating point number (if possible).

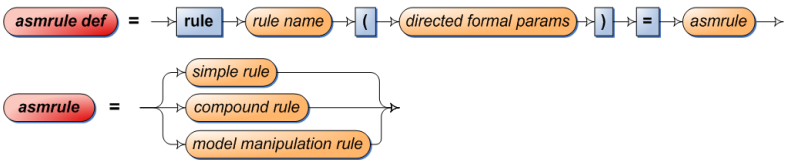
USED BY *base arithmetic term*

5.6 ASM Rules

ASM RULE DEFINITION

DESCRIPTION An ASM rule is the building block of an ASM program. It can be defined either *standalone* or *nested*, as a part of another ASM rule. The entry point of an ASM program is a special standalone rule called *main*.

SYNTAX



SEMANTICS A classical *asmrule* is a *simple rule* or a *compound rule*. In VIATRA-I the set of these rules are extended with *model manipulation rules*, which can modify the Viatra model space.

If an *asmrule* is defined as a standalone rule (*asmrule def*) it is identified by a name (*rule name*) and it can have a parameter list (*directed formal params*).

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

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

Typically the *asmrule* part of an *asmrule def* is a *sequential rule*.

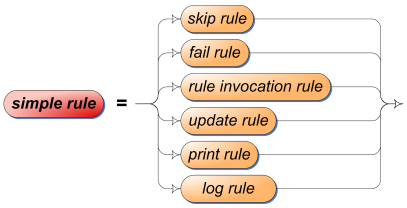
DEFINED IN *asmdef*

5.6.1 Simple Rules

SIMPLE RULE

DESCRIPTION A *simple rule* is such an ASM rule that does not contain nested rules.

SYNTAX



SEMANTICS A simple rule can be

- control flow related (*skip rule*, *fail rule*, *rule invocation rule*),
- variable related (*update rule*) or
- printing related (*print rule*, *log rule*).

DEFINED IN *asmrule*

SKIP RULE

DESCRIPTION A *skip rule* represents an "empty" or "do nothing" instruction.

SYNTAX

SEMANTICS The ***skip;*** rule do not do anything, the execution goes to the next rule.

REMARKS Typically, it is used as a nested rule if a nested rule must be defined in a rule where nothing should be done. For instance in a *choose rule* the expected actions can be the side effect of applying a GT rule and thus the *asmrule* in the do-part should do nothing.

Defined in *simple rule*

Fail Rule

Description A *fail rule* represents an "always fails" instruction, that can be used for interrupting the execution of the containing rule.

Syntax

Semantics The *fail rule* interrupts the execution of the containing rule; this interruption is propagated to upper levels as long as it is catched or the program is terminated.

For example, it can be used to explicitly interrupt an *iterate rule*.

An interrupt generated by a *fail rule* can be catched explicitly by the *try rule*.

Defined in *simple rule*

See also *try rule*, *iterate rule*

Rule Invocation Rule

Description A standalone rule can be invoked by a *rule invocation rule*.

Syntax

Semantics In the *rule invocation rule* the name of the invoked rule have to be specified and the list of the actual parameters according to the formal parameters of the invoked rule.

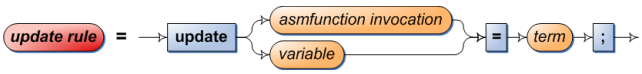
Remarks In order to facilitate modular transformation design, rules are allowed to call rules located in other machines. In such a case, the called rules can be referred to by their fully qualified names composed of the location of its machine container and the name of the ASM rule.

Defined in *simple rule*

Update Rule

Description An *update rule* is used to modify the value of an already defined variable or the value assigned to a specific location of an ASM function.

Syntax



Semantics If the *variable* referred by the update rule is defined then its value is updated to the value resulting from the evaluation of *term*. A compile-time error occurs if the *variable* is not defined.

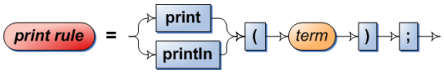
The same applies for the *asmfunction invocation*: the asmfunction referred in the invocation must be already defined otherwise a compile-time error occurs.

Defined in *simple rule*

Print Rule

Description The *print rule* is used for printing output from ASM programs to the code output.

Syntax



Semantics The ***print*** rule evaluates its argument *term* and prints the resulted value to the *Code Output View*, which is a Viatra-I specific Eclipse view. The ***println*** rule adds a newline character to the output string.

Remarks The *print rule* can be used for instance for code generation or for printing status messages.

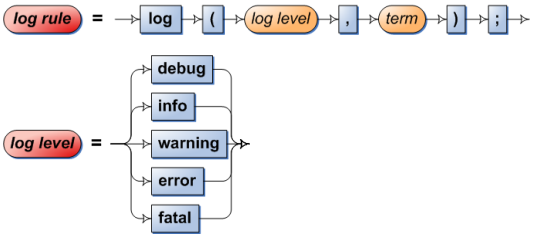
Defined in *simple rule*

See also *log rule*

Log Rule

Description The *log rule* prints a message into the Eclipse Error Log.

Syntax



Semantics The *log rule* evaluates its argument *term* and prints the resulted value to the *Error Log View*, which is a general Eclipse view.

The type of the message (*log level*) can be ***debug***, ***info***, ***warning***, ***error*** or ***fatal***. This determines how Eclipse handles and represents the message.

Defined in *simple rule*

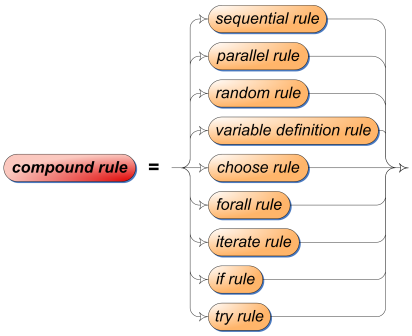
See also *print rule*

5.6.2 Compound Rules

Compound Rule

Description A *compound rule* is such an ASM rule that contains nested rules.

Syntax



Semantics A compound rule can be

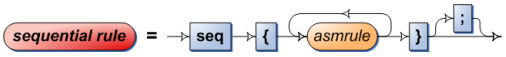
- for defining a set of rules (*sequential rule*, *parallel rule*, *random rule*),
- variable related (*variable definition rule*) or
- pattern matching or graph transformation related (*choose rule*, *forall rule*),
- control flow related (*iterate rule*, *if rule*, *try rule*).

Defined in *asmrule*

Sequential Rule

Description The *sequential rule* is a rule-container that defines the sequential execution of multiple rules.

Syntax



Semantics The *sequential rule* executes *all* rules listed inside in the *defined* order.

A *sequential rule* can contain an arbitrary number of *asmrule*s.

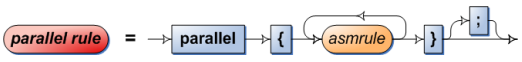
Defined in *compound rule*

See also *parallel rule*, *random rule*

Parallel Rule

Description The *parallel rule* is a rule-container that defines the parallel execution of multiple rules.

Syntax



Semantics The *parallel rule* executes *all* rules listed inside in an *arbitrary* order. If parallel processing is supported on the host machine rules can be executed simultaneously.

A *parallel rule* can contain an arbitrary number of *asmrule*s.

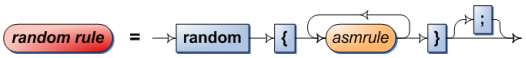
Defined in *compound rule*

See also *sequential rule*, *random rule*

Random Rule

Description The *random rule* is a rule-container; it executes a non-deterministically chosen rule from the rules listed inside.

Syntax



Semantics The *random rule* executes *only one* rule from the contained rules.

A *random rule* can contain an arbitrary number of *asmrules*.

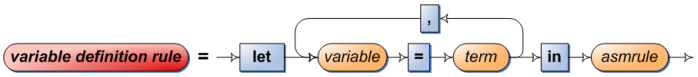
Defined in *compound rule*

See also *sequential rule*, *parallel rule*

Variable Definition Rule

Description The *variable definition rule* is for defining a variable for a given scope.

Syntax



Semantics An ASM variable needs to be defined explicitly prior to its first use. A possible way of it is using the *variable definition rule* or, for short, the *let rule* (a variable can be defined also with the *forall rule* and the *choose rule* or as a parameter of an ASM rule).

So the let rule defines a variable, initializes it with the value resulting from the evaluation of the ASM *term* and then calls its internal *asmrule*. The variable is accessible inside the scope of the rule, i.e. only in its body, in the defined *asmrule*.

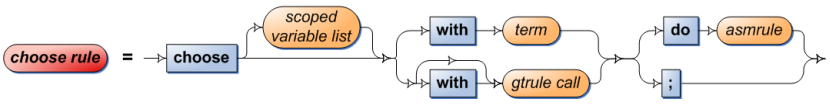
Defined in *compound rule*

See also *forall rule*, *choose rule*, *directed formal params*

Choose Rule

Description The *choose rule* is used to execute a rule for an element in the model space that has a specific property.

Syntax



Semantics 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.

For a combination of values in the variables that make the formula true, the rule is executed.

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

Remarks Note that in a typical model transformation, the *forall rule* and the *choose rule* 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 rule is executed, we may apply a GT rule.

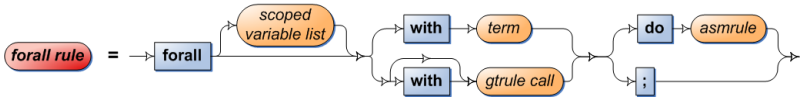
Defined in *compound rule*

See also *forall rule*

Forall Rule

Description The forall rule is used to execute a rule for every element in the model space that has a specific property.

Syntax



SEMANTICS

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

Note that the forall construct always succeeds, even if no matchings were found.

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). VIATRA-I currently does not provide support for detecting such conflicts, thus it is the role of the transformation designer to assure that a GT rule applied in *forall rule* is not conflicting with itself.

The first form of a *forall rule* means that the *asmrule* is executed for all (combination of) variable substitutions that satisfy the formula *term*. The second one means that the graph transformation rule in the *gtrule call* is executed for all possible matches of its pattern and an optional *asmrule* is executed. Theoretically a third form is also possible (taking the variables that satisfy the *term* and doing nothing) but it makes no sense.

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

REMARKS

Note that in a typical model transformation, the *forall rule* and the *choose rule* 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 rule is executed, we may apply a GT rule.

DEFINED IN

compound rule

See also *choose rule*, *iterate rule*

Iterate Rule

Description Using an *iterate rule* is a possible way to manage loop execution in a GTASM.

Syntax

Semantics The *iterate rule* applies its body rule as long as possible, i.e. until its body fails.

A rule can fail in the following situations:

- it is the *fail rule* itself,
- it is the *choose rule* and no matching is found,
- it contains a rule that can fail as a nested rule, this nested rule is executed and fails and the fail exception is not catched by a nested *iterate rule* or *try rule*.

Remarks Since the execution of an *asmrule* can succeed always it can happen that the *iterate rule* does *not terminate*. It can be avoided only with careful program design.

The *iterate rule* construct combined with a single *choose rule* 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).

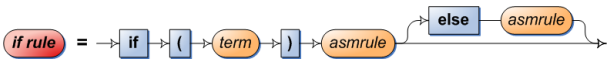
Defined in *compound rule*

See also *forall rule*, *choose rule*, *fail rule*, *try rule*

If Rule

Description An *if rule* is a conditional rule that defines a binary branch in the flow of execution similar to the conditional instructions of programming languages.

Syntax



Semantics The conditional rule has a condition, which is an ASM *term*. The term is evaluated and if it is true then the first *asmrule* is executed. If the condition evaluates to false and the else branch exists then the second *asmrule* is executed.

Defined in *compound rule*

Try Rule

Description The *try rule* is for catching the fail exception of a rule.

Syntax

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

A rule can fail in the following situations:

- it is the *fail rule* itself,
- it is the *choose rule* and no matching is found,
- it contains a rule that can fail as a nested rule, this nested rule is executed and fails and the fail exception is not catched by a nested *iterate rule* or *try rule*.

Defined in *compound rule*

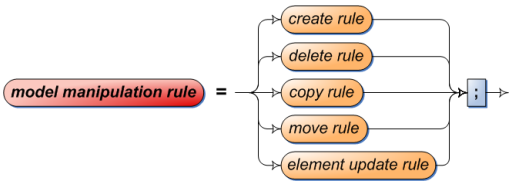
See also *fail rule*, *choose rule*, *iterate rule*

5.6.3 Model Manipulation Rules

Model Manipulation Rule

Description Standard ASM is extended with rules for manipulating the VIATRA-I model space. These rules can be used directly in ASM programs; however, model manipulation can be carried out by graph transformation rules, too.

Syntax



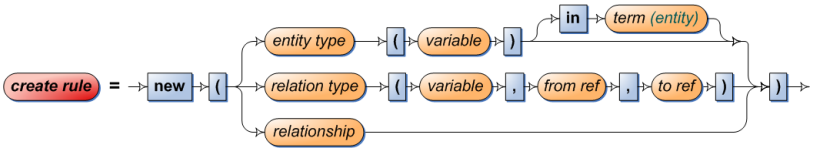
Semantics The rules introduced in this section are special extensions to the ASM language. They provide an interface to the model space to allow

- creation (*create rule*, *copy rule*),
- change (*move rule*, *element update rule*), or
- deletion of model elements (*delete rule*).

Create Rule

Description The *create rules* can be used for model element or element relationship creation.

Syntax



Semantics In case of creation of an entity the optional "in" parameter defines a namespace, where the new element has to be created. The *term* has to be evaluated to *model element ref* that references an entity.

In case of creation of a relation *from ref* and *to ref* is a variable that references the source and target model elements.

If a new model element is created, a reference to it is stored in the *variable*.

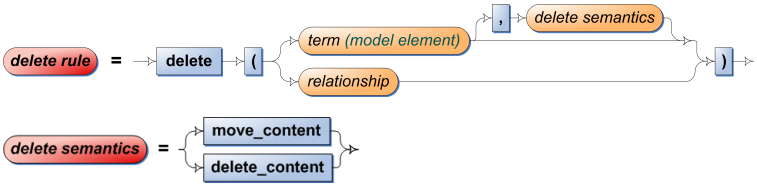
The ***new*** rule can also be used for creating relationships between model elements.

DEFINED IN *model manipulation rule*

DELETERULE

DESCRIPTION The *delete rule* removes a model element or a relationship from the model.

SYNTAX



SEMANTICS The parameter of a *delete rule* is either a *term* that evaluates to a model element or a *relationship*.

If a model element is to be deleted then all relations that have this element either as source or as target are deleted implicitly, too.

If an entity is to be deleted then optionally the *delete semantics* can be defined:

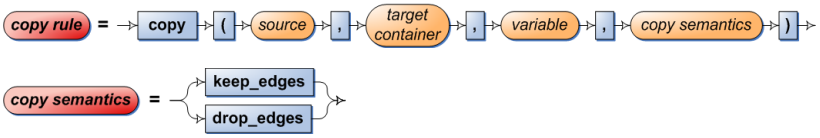
- ***move_content*** means that only the referenced element is deleted and the contained elements –if any– remain in the model; the new container for these is the *root* element. This behavior corresponds to the view when the model is interpreted as a *graph*: in each deletion operation only a single node is deleted with corresponding edges.
- ***delete_content*** means that both the referenced element and its contained elements –if any– are deleted from the model. This behavior corresponds to the view when the model is interpreted along its *containment hierarchy*: in each deletion operation the node with its whole subtree is deleted with the corresponding edges. This is the default behavior.

DEFINED IN *model manipulation rule*

CopyRule

Description This rule copies a subtree of the current model space to an other location.

Syntax



Semantics The first parameter (*source*) is the root of the sub-model to be copied, and the second parameter is the *target container*. Both *source* and *target container* is an ASM *term* that evaluates to a *model element ref* referencing an entity.

The reference to the root element of the newly created sub-model is stored in *variable*.

The *copy semantics* defines whether to keep (***keep_edges***) or to delete the edges (***drop_edges***) that are going to or coming from outside the sub-model under copy.

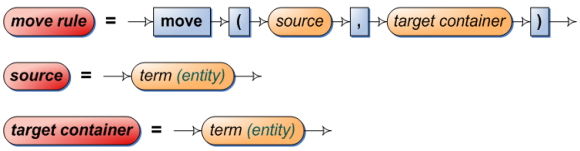
Defined in *model manipulation rule*

See also *move rule*

Move Rule

Description The *move rule* changes the location of an entity in the containment hierarchy.

Syntax



Semantics The *move rule* moves the element specified by the first parameter (*source*) to the container specified by the second parameter (*target container*). Both *source* and *target container* is an ASM *term* that evaluates to a *model element ref* referencing an entity.

The *move rule* performs changes only in the containment hierarchy, i.e. no model element or relationship is deleted from the model space.

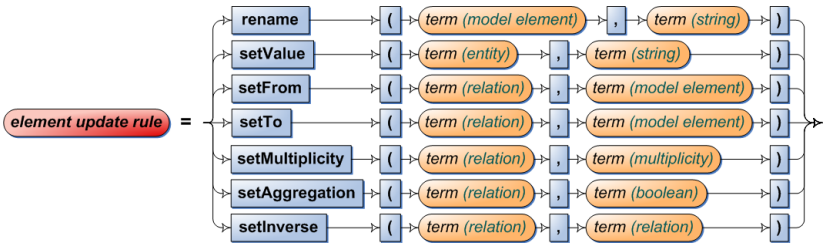
Defined in *model manipulation rule*

See also *copy rule*

Element Update Rule

Description An *element update rule* changes a property of a model element.

Syntax



Semantics The ***rename*** rule modifies the *name* of a model element. Both parameters are ASM *terms*; the first must evaluate to a *model element ref* referencing an entity or relation, the second must evaluate to a string that will be the short name of the model element. Since the new name is required to be unique within its container (in case of entities) or within the source element (in case of relations), a possible name clash is resolved automatically by the Viatra-I interpreter at run-time by extending the name of the renamed model element with a randomly generated tag.

The ***setValue*** rule modifies the *value* of an entity. Both parameters are ASM *terms*; the first must evaluate to a *model element ref* referencing an entity, the second must evaluate to a string that will be the new value assigned to the entity.

The ***rename*** rule modifies the *name* of a model element. Both parameters are ASM *terms*; the first must evaluate to a *model element ref* referencing an entity or relation, the second must evaluate to a string that will be the short name of the model element.

The ***setFrom*** rule sets the source of a relation. Both parameters are ASM *terms* that must evaluate to *model element ref*s; the first references a relation, the second references an arbitrary model element that will be the new source of the relation.

The ***setTo*** rule sets the target of a relation. Both parameters are ASM *term*s that must evaluate to *model element ref*s; the first references a relation, the second references an arbitrary model element that will be the new target of the relation.

The ***setMultiplicity*** rule modifies the *multiplicity* property of a relation. Both parameters are ASM *term*s; the first must evaluate to a *model element ref* referencing a relation, the second must evaluate to a multiplicity value that will be the new value assigned to the relation.

The ***setAggregation*** rule modifies the *aggregation* property of a relation. Both parameters are ASM *term*s; the first must evaluate to a *model element ref* referencing a relation, the second must evaluate to a boolean value that will be the new value assigned to the relation.

The ***setInverse*** rule defines an *inverse relation* between two relations. Both parameters are ASM *term*s that must evaluate to *model element ref*s referencing relations; the inverse property is set for both relations having referenced the other relation as the new value.

DEFINED IN *model manipulation rule*

SEE ALSO *entity*, *relation*, *type name*, *model element ref*, *string constant*, *multiplicity constant*, *boolean constant*

Bibliography

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

[2] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.

[3] Á. Horváth, D. Varró, and G. Varró. Generic search plans for matching advanced graph patterns. In *Proc. Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2007)*. 2007. To appear.

[4] A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg (eds.), *Proc. 2nd International Conference on Graph Transformation (ICGT 2004), Rome, Italy*, vol. 3256 of *LNCS*, pp. 319–335. Springer, 2004.

[5] D. Ullman. *Principles of Database and Knowledge Base Systmes*, vol. Deductive Databases. Computer Science Press, 1988.

[6] D. Varró. *Automated Model Transformations for the Analysis of IT Systems*. Ph.D. thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2004.

[7] D. Varró and A. Pataricza. VPM: A visual, precise and multi-level metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, vol. 2(3):pp. 187–210, 2003.

[8] G. Varró, D. Varró, and K. Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In G. Karsai and G. Taentzer (eds.), *GraMot 2005, International Workshop on Graph and Model Transformations*, vol. 152 of *ENTCS*, pp. 191–205. Elsevier, 2006.