Available online at www.sciencedirect.com

ScienceDirect

Science of Computer Programming 68 (2007) 214–234

Science of Computer Programming

www.elsevier.com/locate/scico

The model transformation language of the VIATRA2 framework

Dániel Varró*, András Balogh

Budapest University of Technology and Economics, Department of Measurement and Information Systems, H-1117, Magyar tudosok krt. 2., Budapest, Hungary
OptXware Research and Development LLC, Budapest, Hungary

Received 15 August 2006; received in revised form 17 April 2007; accepted 14 May 2007
Available online 5 July 2007

Abstract

We present the 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 transformation and meta-transformation rules) in unidirectional model transformations frequently used in formal model analysis to carry out powerful abstractions.

Keywords: Model transformation; Graph transformation; Abstract state machines

1. Introduction

The crucial role of model transformation (MT) languages and tools for the overall success of model-driven system development has been revealed in many surveys and papers during the recent years. To provide a standardized support for capturing queries, views and transformations between modeling languages defined by their standard MOF metamodels, the Object Management Group (OMG) has recently issued the QVT standard.

QVT provides an intuitive, pattern-based, bidirectional model transformation language, which is especially useful for synchronization kind of transformations between semantically equivalent modeling languages. The most typical example is to keep UML models and database models (or UML models and application code) synchronized bidirectionally during model evolution.

However, there is a large set of model transformations, which are unidirectional by nature, especially, when the source and target models represent information on very different abstraction levels (i.e. the model transformation is either *refinement* or *abstraction*). Unfortunately, the current QVT Mapping Language is far less intuitive and easy-to-use in case of unidirectional transformations. Even those MT approaches that aim at implementing the standard (like ATL [5]) have chosen a more intuitive language for such transformations.

Typical examples of abstraction transformations can be identified during formal model validation [9]. The VIATRA2 model transformation framework (available as an Eclipse plugin [3]) primarily aims at designing model

* Corresponding author at: Budapest University of Technology and Economics, Department of Measurement and Information Systems, H-1117, Magyar tudosok krt. 2., Budapest, Hungary.
E-mail addresses: varro@mit.bme.hu (D. Varró), abalogh@mit.bme.hu (A. Balogh).

0167-6423/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.05.004

transformations to support the precise model-based system development with the help of invisible formal methods. In this approach, formal methods are hidden by automated model transformations which project system models into various mathematical domains. Afterwards, results of the mathematical analysis are back-annotated to the source engineering models by using additional model transformations. It is worth pointing out that since transformations for model validation are complex abstractions, bidirectional transformations rarely provide appropriate solution due to the large abstraction gap.

VIATRA2 has chosen to integrate two intuitive, and mathematically precise rule-based specification formalisms, namely, graph transformation (GT) [13] and abstract state machines (ASM) [10] to manipulate graph-based models. VIATRA2 also facilitates the separation of transformation design (and validation) and execution time. During design time, the VIATRA2 interpreter takes such MT descriptions and executes them on selected models as experimentation. Then final model transformation rules can be compiled into efficient, platform-specific transformer plugins for optimal execution [7].

In the current paper, we present the textual VIATRA2 transformation language which is composed of three sublanguages for metamodeling (Section 2), pattern specification (Section 3), and rule-based model transformations (Section 4). The current paper extends [6] by a more detailed description of the ASM control language (Section 4.2) and the formal semantics of the major VIATRA2 constructs (Sections 4.1.2 and 5). Moreover, a brief case study (Section 6) now demonstrate the use of the language constructs for a transformation used in model analysis.

While there is a large set of available tools using the paradigm of graph transformation (see Section 7) for model transformations, the specification language of VIATRA2 uniquely provides a combined support for recursive graph patterns, and generic transformations or meta-transformations.

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.

Currently, most widely used metamodeling languages (e.g. Eclipse Modeling Framework (EMF) [1]) are derived with slight variations from the Meta Object Facility (MOF) [19] metamodeling standard issued by the OMG. However, as stated in [27], the MOF standard fails to support multi-level metamodeling [4], which is typically a critical aspect for integrating different technological spaces [8] where different metamodeling paradigms are used.

Therefore, we have chosen to use the VPM (Visual and Precise Metamodeling) [27] metamodeling approach in the VIATRA2 framework, which provides a uniform representation of models, metamodels and transformations in a VPM *model space* using explicit and generalized instance-of relations.

As only an abstract syntax was defined for VPM in [27], 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 VIATRA2 by import plugins and exporters defined by code generation transformations. So far, models from very different technological spaces such as XML, EMF, semantic web and modeling languages of high practical relevance like BPEL, UML (and various domain-specific languages in the dependable embedded, telecommunication, and service-oriented domain) have been successfully integrated into VIATRA2 . While VIATRA2 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.

2.1. Metamodeling concepts in VTML

Entities and relations. As shown in the metamodel of Fig. 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

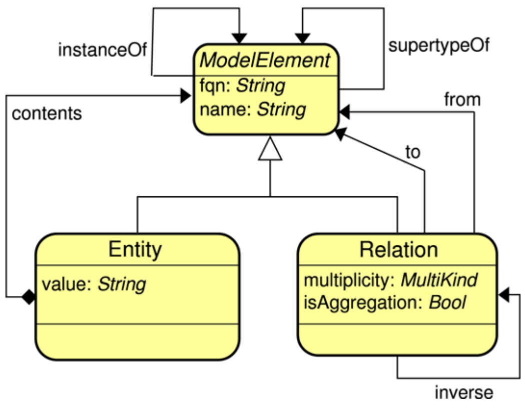


Fig. 1. The metamodel of the VPM model space.

relationships between model elements.[1] Furthermore, entities may also have an associated string value which contains application-specific data.

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 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. 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 modeling tools. For example, the FQN of the entity Association in Fig. 3 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 VIATRA2).

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 supertypeOf relationship between the two relations representing the properties.

2.2. Demonstrating example

The technicalities of VTML are demonstrated in Fig. 2 using the metamodels of a model transformation from UML class diagrams to description logic. This transformation was implemented in the context of the DECOS European IP

[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.

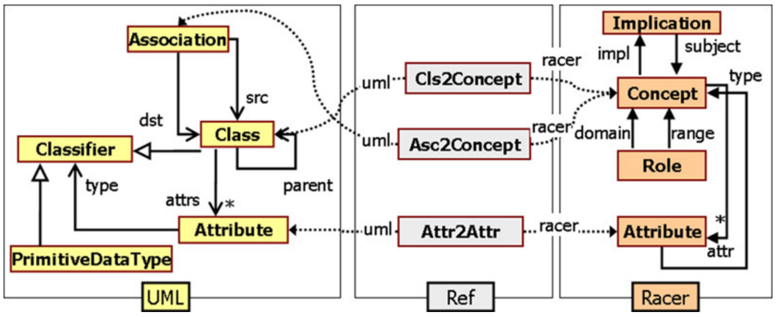


Fig. 2. Sample UML and Racer metamodels.

```
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);

}
```

```
entity(racer)
{
 entity(concept);
  relation(attr,concept,attribute);
  relation(impl,concept,implication);
 entity(role);
  relation(domain,role,concept);
  relation(range,role,concept);
 entity(attribute);
  relation(type,attribute,concept);
 entity(implication);
  relation(subject,implication,concept);
}
entity(Ref) {
 entity(class2concept);
  relation(uml,class2concept,Class);
  relation(racer,class2concept,concept);
 entity(assoc2concept);
  relation(uml,assoc2concept,Association);
  relation(racer,assoc2concept,concept);
 entity(attr2attr);
  relation(uml,attr2attr,Attribute);
  relation(racer,attr2attr,attribute);
}
```

Fig. 3. Sample VTML metamodels: UML and Racer.

[12] to carry out semantic validation of domain-specific modeling languages and models in the dependable embedded domain using the Racer reasoner [20]. Extracts from this transformation (with a simplified UML metamodel and a subset of rules) will be used throughout the paper for demonstration purposes.

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.

The metamodel of the Racer tool consists of Concepts, which can be interrelated through several Roles (as indicated by domain and range). Each concept is allowed to have several attributes (Attribute) denoted by attr. Inheritance between concepts (similar to UML) can be defined using Implication elements. A child concept (denoted by impl) implicates its super concept (subject).

Finally, the source and target metamodels are interrelated by using the constructs of a reference metamodel (denoted as Ref). This reference metamodel declares that a UML class may be related to a Racer concept (as cls2concept), a UML association may also be connected to a Racer concept (as asc2concept), and UML attributes can be related to Racer attributes (see att2attr). The VTML representation of the metamodels is listed in Fig. 3.

3. The pattern language

Graph patterns (GP) are the atomic units in VIATRA2 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

```
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, it is
//not present in 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)!="")
}
```

Fig. 4. Basic patterns and negative patterns.

of the *Viatra Textual Command Language (VTCL)*, which is the main textual language in VIATRA2 to specify model transformations.

3.1. Graph patterns

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) satisfies a graph pattern, if the pattern can be matched to a subgraph of the model using a generalized *graph pattern matching* technique presented in [26]. In this section, we present an informal introduction to graph patterns in VTCL, while their formal semantics will be discussed in Section 5.

3.1.1. 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. 4, a simple pattern can be fulfilled by a 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), i.e. the transformation engine is responsible for the appropriate ordering of predicates by using sophisticated algorithms for search plan generation [28].

The keyword *neg* denotes if a subpattern serves as a negative condition for another pattern. The negative pattern in the right column of Fig. 4 can be satisfied if there is a class (CP) for the class in the parameter (C) that is the parent of C as indicated by relation P of type Class.parent. If this condition can be satisfied, the outer (positive) pattern matching will fail. Thus the pattern matches to top-most classes in the parent hierarchy.

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. Additional Boolean constraints can be expressed by the *check* condition, which also need to be fulfilled for successful pattern matching. Our sample pattern isTopClass can be matched to classes with non-empty names.

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) when the expressiveness of such patterns converges to first order logic [21].

3.1.2. 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

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

Fig. 5. Recursive patterns.

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.

Pattern calls 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, VIATRA2 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 in Fig. 5 illustrates the usage of recursion.

A class Parent is the ancestor of an other class Child, if Child is a direct child of classes Parent, 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.

3.1.3. Constraint checking in VIATRA2

Due to the heterogeneity of the actual modeling tools aimed to be integrated by VIATRA2 , we typically build on the tools themselves for initial well-formedness checks of models, thus we regularly assume that (i) models imported into VIATRA2 model transformation framework have already been pre-checked by the front-end tool itself, and (ii) models exported from VIATRA2 are post-checked by the back-end tool. Naturally, one can easily specify and check well-formedness constraints of a modeling language in VIATRA2 by using graph patterns and logging "transformations", but this is not the main usage scenario for the framework.

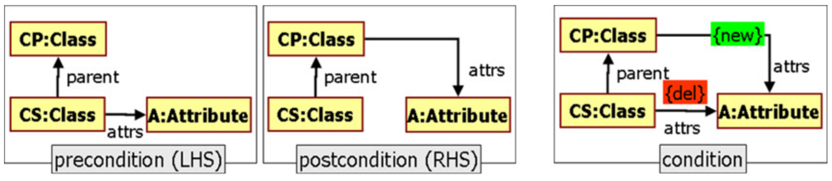
These constraints are typically checked only after the termination of a model transformation by regular graph pattern matching to increase the flexibility of designing transformations. Thus constraints are required to be respected in the end, but such constraints are not enforced during the transformation for performance considerations. Actually, successful matches of the negation of constraint patterns are sought in order to constraint violation.

4. The transformation language

Transformation descriptions in VIATRA2 consist of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph transformation (GT) [13] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [10] rules can be used for the description of control structures.

4.1. Graph transformation rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules [13], 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 notation for defining graph transformation rules using the gtrule keyword.



(a) Traditional notation. (b) Compact notation.

Fig. 6. Sample graph transformation rule.

```
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));
   }
}
```

Fig. 7. Graph transformation rule in traditional notation.

4.1.1. Traditional notation of graph transformation rules

The sample graph transformation rule in Fig. 6 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 the graphical notation in Fig. 6(a) and the textual one in Fig. 7). 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 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.

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. Examples for negative application conditions will be given in the case study of Section 6.

4.1.2. Informal semantics of graph transformation rules

Parameter passing. A main difference with the traditional GT notation is related to the use of parameter passing between 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.

- A parameter of the postcondition is treated as an *input parameter* if (i) it is also a precondition parameter or (ii) it is passed to the entire GT rule as an input parameter. 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.

```
gtrule liftAttrsR(inout CP, inout CS, inout A) =
{
   condition pattern cond(CP,CS,A) =
   {
      Class(CP);
      Class(CS);
      Class.parent(Par,CS,CP);
      Attribute(A);
      del Class.attrs(Attr,CS,A);
      new Class.attrs(Attr2,CP,A);
   }
}
```

Fig. 8. Graph transformation rules in compact notation.

- 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 unchanged by the rule does not need to 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.

This way, rule liftAttrsR can be further compacted by simply omitting the parent relation from the postcondition and the pattern parameter lists.

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 Section 3.1.2.

However, pattern calls in the postcondition (RHS) of GT rules is a unique feature compared to other GT tools. Currently, VIATRA2 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. More examples of pattern calls will be provided in the case study of Section 6.

4.1.3. *Compact notation of graph transformation rules*

The compact format of graph transformation rules directly corresponds to the FUJABA [18] notation as shown in Figs. 6(b) and 8.

The rule contains a simple pattern (marked with keyword condition), that jointly defines the left-hand side (LHS) of the graph transformation rule, and the actions to be carried out. Pattern elements marked with keyword new are created after a matching for the LHS is succeeded (and therefore do not participate in the pattern matching), and elements marked with keyword del are deleted after pattern matching.

4.1.4. *Generic transformations and meta-transformations*

To provide algorithm-level reuse for common transformation algorithms independent of a certain metamodel, VIATRA2 supports generic transformations and meta-transformations, which are built on explicit instance-of relations

```
gtrule liftUp(inout CP, inout CS, inout A, inout ClsE, inout AttE,
              inout ParR, inout AttR) = {
   condition pattern transClose(CP,CS,A, ClsE, AttE, ParR, 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);
      del relation(Attr,CS,A);
      del instanceOf(Attr,AttR);
      new relation(Attr2,CP,A);
      new instanceOf(Attr2,AttR);
   }
}
```

Fig. 9. Generic transformation rules.

of the VPM metamodeling framework. 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 generic example of Fig. 9 generalizes the previous GT rule parameterized by types taken from arbitrary metamodels during execution time.

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 VIATRA2 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.

Note that while other GT tools may also store metamodels and models uniformly in a common graph structure, only PROGRES [23] supports type parameters in rules, while none of them supports manipulation of (existing) type–instance relationship like the dynamic reclassification (retyping) of objects as in VIATRA2 . In our solution, generic algorithms (e.g. transitive closure, graph traversals, fault modeling etc.) can be reused without changes for different metamodels.

4.1.5. *Invoking graph transformation rules*

The basic invocation of a graph transformation rule is initiated using the apply keyword within a choose or a forall construct (further details on these constructs are given in Section 4.2). 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 example in Fig. 10 illustrates some possible invocations of our sample rule liftAttrsR.

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 (one by one) for each of them in a single compound step.

4.2. *Control structure*

To control the execution order and mode of graph transformations, VTCL 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 [10] that correspond to the constructs of conventional programming languages.

```
//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)
```

Fig. 10. Calling GT rules from ASM programs.

```
// 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
```

Fig. 11. Overview of built-in ASM rules in VIATRA2.

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. 11. 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 VIATRA2 dialect of ASMs also includes built-in rules for manipulating the model space. As a result, 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. 12.

```
// 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 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);
```

Fig. 12. Overview of model manipulation rules in VIATRA2.

```
// 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 Cl below M with find isTopClass(Cl) do
      call printTopLevel(Cl);
   //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");
}
```

Fig. 13. A sample ASM program driving a model transformation.

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. VIATRA2 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 are 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 [26] for more details). The following example (in Fig. 13) demonstrates some of the main control structures.

5. Formal semantics of graph patterns in VTCL

As the main conceptual novelties of the VIATRA2 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 omit the formalization of other elements of the language, like ASM or GT rules. These already have a rich theoretical background, and VIATRA2's contribution is less significant. For instance, a formal semantics of ASMs is given in [10], while an ASM formalization of GT rules is listed in [26].

5.1. 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/2, and (ii) traditional *boolean and arithmetic function symbols*. A *state of the model space* (denoted by $\mathcal{A}$) 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 by $[\![\mathsf{entity}(v)]\!]^{\mathcal{A}}$), if an entity exists in the model space in state $\mathcal{A}$ uniquely identified by v. Otherwise the predicate is evaluated to false: $[\![\neg\mathsf{entity}(v)]\!]^{\mathcal{A}}$.
- *Relation.* A predicate $\mathsf{relation}(v, s, t)$ is true (denoted by $[\![\mathsf{relation}(v, s, t)]\!]^{\mathcal{A}}$), 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 by $[\![\mathsf{supertypeOf}(sup, sub)]\!]^{\mathcal{A}}$), 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 by $[\![\mathsf{instanceOf}(ins, typ)]\!]^{\mathcal{A}}$), if ins is an 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 by $[\![\mathsf{in}(chi, par)]\!]^{\mathcal{A}}$), 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 [26].

5.2. 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 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 such as selection (σ), projection (π), inner join ($\bowtie$), and left outer join ($\ltimes$). Below we assume the reader's familiarity with these elementary relational operations.

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 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 σ fulfilling criterion F 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 ($\bowtie$) 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.

Negative pattern calls. Negative pattern calls (Rows 6–7) are syntactically very close to ordinary pattern calls, however, their semantic treatment is essentially different. We assume that there are no recursive calls alternating

Table 1
Deriving predicates for VTCL patterns

	VTCL grammar (simplified)	Derived predicates	Semantics
1	*fact* = entity(V) (or relation(V,S,T))	$fact(V) \leftarrow \mathsf{entity}(V)$ $fact(V,S,T) \leftarrow \mathsf{relation}(V,S,T)$	$\llbracket fact(V)\rrbracket_F^{\mathcal{A}} \overset{def}{=} \sigma_F(\{v \mid \llbracket \mathsf{entity}(v)\rrbracket^{\mathcal{A}}\})$ $\llbracket fact(V,S,T)\rrbracket_F^{\mathcal{A}} \overset{def}{=} \sigma_F(\{(v,s,t) \mid \llbracket \mathsf{relation}(v,s,t)\rrbracket^{\mathcal{A}}\})$
2	*fact* = supertypeOf(A,B) (or instanceOf(A,B))	$fact(A,B) \leftarrow \mathsf{supertypeOf}(A,B)$ (or instanceOf(A,B))	$\llbracket fact(A,B)\rrbracket_F^{\mathcal{A}} \overset{def}{=} \sigma_F(\{(a,b) \mid \llbracket \mathsf{supertypeOf}(a,b)\rrbracket^{\mathcal{A}}\})$ (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))	$\llbracket fact(A,B)\rrbracket_F^{\mathcal{A}} \overset{def}{=} \sigma_F(\{(a,b) \mid \llbracket \mathsf{in}(a,b)\rrbracket^{\mathcal{A}}\})$ (or below(a,b))
4	*simple* = $fact_1$ **;** $\ldots fact_n$ **;**	$simple(\overline{X}) \leftarrow$ $fact_1(\overline{X_1}) \wedge \cdots \wedge fact_1(\overline{X_n})$	$\llbracket simple(\overline{X})\rrbracket_F^{\mathcal{A}} \overset{def}{=} \overset{F_1}{\bowtie}_l (\llbracket fact_l(\overline{Y_l})\rrbracket_{F_1}^{\mathcal{A}})$ where $F_1 = F \wedge \bigwedge_k X_{i,k} = Y_{j,k}$ for each variable renaming.
5	*poscall* = *simple* **find (** $base_1$ **);** … **find (** $base_n$ **);**	$poscall(\overline{X}) \leftarrow$ $simple(X_0) \wedge \bigwedge_j base_j(\overline{X_j})$	$\llbracket poscall(\overline{X})\rrbracket_F^{\mathcal{A}} \overset{def}{=} \llbracket simple(\overline{Y_0})\rrbracket_{F_1}^{\mathcal{A}} \overset{F_1}{\bowtie} \llbracket base_j(\overline{Y_j})\rrbracket_{F_1}^{\mathcal{A}}$ with $F_1 = F \wedge \bigwedge_k (X_k = Y_k)$ for each var renaming.
6	*neg* = ***neg*** **find (** $\{base \mid recur\}$ **);**	$neg(\overline{X}) \leftarrow base(\overline{X})$	$\llbracket neg(\overline{X})\rrbracket_F^{\mathcal{A}} \overset{def}{=} \llbracket base(\overline{X})\rrbracket_F^{\mathcal{A}}$
7	*base* = *poscall* $neg_1, \ldots, neg_n$	$base(\overline{X}) \leftarrow poscall(\overline{X_0}) \wedge \bigwedge_j (\neg neg_j(\overline{X_j}))$	$\llbracket base(\overline{X})\rrbracket_F^{\mathcal{A}} \overset{def}{=} \sigma_{F_2}(\llbracket poscall(\overline{Y_0})\rrbracket_{F_1}^{\mathcal{A}} \overset{F_1}{\ltimes} (\overset{F_1}{\ltimes} \llbracket neg_j(\overline{Y_j})\rrbracket_{F_1}^{\mathcal{A}}))$ 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 (** $recur_n$ **);**	$recur(\overline{X}) \leftarrow base(\overline{X_0}) \wedge recur_1(\overline{X_1}) \wedge \cdots \wedge recur_n(\overline{X_n})$	$\llbracket recur(\overline{X})\rrbracket_F^{\mathcal{A}} \overset{def}{=} lfp(\llbracket base(\overline{Y_0})\rrbracket_{F_1}^{\mathcal{A}} \overset{F_1}{\bowtie} \llbracket recur_1(\overline{Y_1})\rrbracket_{F_1}^{\mathcal{A}} \overset{F_1}{\bowtie} \cdots \overset{F_1}{\bowtie} \llbracket recur_n(\overline{Y_n})\rrbracket_{F_1}^{\mathcal{A}})$ where $F_1 = F \wedge \bigwedge_k X_k = Y_k$ for each variable renaming
9	*body* = *recur* (***check*** *cond*)?	$body(\overline{X}) \leftarrow recur(\overline{X}) \wedge check(\overline{X})$	$\llbracket body(\overline{X})\rrbracket_F^{\mathcal{A}} \overset{def}{=} \llbracket recur(\overline{X})\rrbracket_{F_1}^{\mathcal{A}}$ where $F_1 = F \wedge check(\overline{X})$
10	**pattern** *patt* = { bod_1 } **or** { bod_2 }	$patt(\overline{X}) \leftarrow bod_1(\overline{X}) \vee bod_2(\overline{X})$	$\llbracket patt(\overline{X})\rrbracket_F^{\mathcal{A}} \overset{def}{=} \llbracket bod_1(\overline{X})\rrbracket_F^{\mathcal{A}} \cup \llbracket bod_2(\overline{X})\rrbracket_F^{\mathcal{A}}$

between negative and positive patterns. That is, if the positive patterns transitively called by a pattern p are denoted by P while patterns transitively called by a negative pattern of p is denoted by N then $P \cup N = \emptyset$.

After that the left outer join ($\ltimes$) 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 are 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 $[\![base]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} [\![recur_1]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} \cdots \overset{F_1}{\bowtie} [\![recur_n]\!]_F^{\mathcal{A}})$. Initially, $[\![recur_j^{(0)}]\!]_F^{\mathcal{A}} = \emptyset$ for all j. Then in the first iteration, it collects all matchings derived by the base cases of recursive patterns, i.e. $[\![recur_j^{(1)}]\!]_F^{\mathcal{A}} = [\![base_j]\!]_F^{\mathcal{A}}$ (i.e. its own base case). In all upcoming iterations, $[\![recur^{(i+1)}]\!]_F^{\mathcal{A}}$ is defined as $[\![base]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} [\![recur_1^{(i)}]\!]_F^{\mathcal{A}} \overset{F_1}{\bowtie} \cdots \overset{F_1}{\bowtie} [\![recur_n^{(i)}]\!]_F^{\mathcal{A}}$ 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 the selection criteria F, while the result of OR patterns (Row 10) is defined as the union of the results for each pattern body.

5.3. 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 variable binding, e.g., a pattern parameter can be an input parameter at a place, while it can be quantified by forall at a different location.

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 the 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** $patt(\overline{X^{in}} \cup \overline{X^f})]\!]_F^{\mathcal{A}} \overset{def}{=}$ 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** $patt(\overline{X^{in}} \cup \overline{X^f})]\!]_F^{\mathcal{A}} \overset{def}{=} \pi_{\overline{X^{in}} \cup \overline{X^f}}([\![patt(\overline{X})]\!]_F^{\mathcal{A}})$, 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 is 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.

5.4. Notes on algorithmic considerations

Note that this formal semantics, which is close to various deductive databases [25], also highlights the main strategy of pattern matching in VIATRA2 . Unfortunately, the actual algorithms used in VIATRA2 are far too complicated, thus their detailed description is out of scope for the current paper.

Since pattern predicates are not ordered in VIATRA2 (nor in deductive databases), their proper ordering is a crucial step for the overall performance of pattern matching. The VIATRA2 implementation improves performance by carrying out global optimization for predicate ordering by flattening pattern non-recursive pattern calls using a generic search plan representation [15].

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 [28].

In case of recursive calls, we build on *magic sets* [25], 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.

6. Case study: A UML-to-Racer mapping

We present a case study in this section to illustrate the usage of language constructs introduced earlier. We selected a real-life transformation that is used in the European IP DECOS [12]. The transformation maps UML structure models (class diagrams) into ontologies for the Racer reasoner, which can verify the static completeness and consistency of the input model. This transformation is used in the project to validate embedded system models designed using domain-specific metamodels.

The goal of the transformation is the creation of an ontology for a given class structure. The transformation can be described by the following informal rules:

- Each class is mapped to a concept in the Racer model. If the class has a superclass, an implication (representing the inheritance) is created between the two. If the class is a top-level class, an implication is created for the common root class (*TOPC*). The latter is needed to ensure a single inheritance tree.
- Each (binary) association is mapped into the following structure: each association end is mapped to a concept that has two roles; one for each end of the association. This enables a more complex analysis of associations. The concepts created from association ends are inherited from the common root concept *TOPA*.
- Each attribute from the classes is mapped to an attribute for the corresponding concept. An attribute of basic type is mapped to a Racer datatype. This part of the transformation is omitted from the current paper due to its technical nature.

6.1. Graph patterns

We defined several generic graph patterns in Fig. 14 that can be reused by graph transformation rules. The UML-related ones are grouped into a separate UML pattern library, that can be reused in several transformations, and can be exchanged with libraries designed for other UML versions. The separation of generic, reusable graph patterns for a given modeling language (UML, Racer, etc.) results in an improved maintainability of the transformations.

- Pattern *isUmlClassWithSuperClass* matches UML classes that have superclasses, and *isUmlClassWithoutSuperClass* matches top-level classes.
- Pattern *isUmlAssocWithEndsBetweenClasses* selects associations with their association ends and connecting classes that can be transformed into Racer.
- Pattern *isUmlClassWithAttr* matches class attributes, and *isUmlAttrOfType* matches attribute data types; both are used for attribute transformation.
- Transformation-specific patterns are *isClassWithConcept* that matches transformed classes together with their associated Racer concepts, and *conceptImplication* which is used for the concept hierarchy.

```
pattern isUmlAssocEndAtClass (AssocEnd, Class) =
{
  Association(Assoc);
  Association.connection(CAA1,Assoc,AssocEndA);
  AssociationEnd(AssocEnd);
  AssociationEnd.type(TAC1,AssocEndA,ClassA);
  Class(Class);
}
pattern isUmlClassWithSuperClass(Class)=
{
  Class(SuperClass);
  Generalization.supertype(SpGG,UmlGen,SuperClass);
  Generalization(UmlGen);
  Generalization.subtype(SbGG,UmlGen,UmlClass);
  Class(UmlClass);
}
pattern isUmlClassWithoutSuperClass(UmlClass) =
{
  Class(UmlClass);
  neg find isUmlClassWithSuperClass(UmlClass);
}
pattern isUmlClassWithAttr(Class, Attr) =
{
  Class(Class);
  Attribute(Attr);
  Classifier.feature(FE,Class,Attr);
}
pattern isUmlAttrOfType(Attr, Type) =
{
 Attribute(Attr);
 DataType(Type);
 StructuralFeature.type(TSC,Attr,Type);
}
pattern isClassWithConcept(Cls,Concept,Ref)=
{
 Class(Cls);
 class2concept.uml(X1,CC,Cls);
 class2concept(CC) in Ref;
 class2concept.racer(X2,CC,Concept);
 concept(Concept);
}
pattern isAssocWithConcept(AscEnd,Conc,Ref)=
{
   AssociationEnd(AscEnd);
   assoc2concept.uml(X1,A2R,AscEnd);
   assoc2concept(A2R) in Ref;
   assoc2concept.racer(X2,A2R,Conc);
   concept(Conc);
}
pattern conceptRole(Role,Domain,Range,Trg) =
{
   concept(Domain);
   role.domain(X1,Role,Domain);
   role(Role) in Trg;
   role.range(X2,Role,Range);
   concept(Domain);
}
pattern conceptImplication(Concept,Subject,Trg) =
{
 concept(Concept);
 concept.impl(IM1,Concept,Impl1);
 implication(Impl1) in Trg;
 implication.subject(IM2,Impl1,Subject);
 concept(Subject);
}
```

Fig. 14. Graph patterns of the case study.

6.2. *Graph transformation rules*

The rules of the UML-to-Racer mapping are listed in Figs. 15 and 16.

Rule *class2concept* (Fig. 15) maps classes to Racer concepts. Before mapping it ensures that the class is not already mapped (negative condition). It also creates the reference model elements together with the concept objects. The find

```
// Mapping each class to a concept
gtrule class2concept(in Cls, in Trg, in Ref) = {
 precondition pattern pre(Cls, Ref) =  {
   Class(Cls);
   find isUmlClass(Cls);
   neg find isClassWithConcept(Cls,CC, Ref);
 }
 postcondition pattern post(Cls,Trg,Ref,Concept) =  {
   concept(Concept) in Trg;
   find isClassWithConcept(Cls,Concept,Ref);
   Class(Cls);
 }
 action {
  rename(Concept,name(Cls));
 }
}
// Mapping top-level classes to an implication
gtrule topClassImplication (in UmlClass, in TopConcept, in Trg, in Ref) = {
 precondition pattern pre(UmlClass,Concept,Ref) =
 {
   Class(UmlClass);
   find isUmlClassWithoutSuperClass(UmlClass);
   find isClassWithConcept(UmlClass,Concept,Ref);
   concept(Concept);
 }
 postcondition find conceptImplication(Concept,TopConcept,Trg);
}
// Matching classes with supertypes to an implication
gtrule classImplication (in UmlClass, in UmlSuperClass, in Trg, in Ref) =
{
 precondition pattern pre(UmlClass, UmlSuperClass,Concept,SuperConcept, Ref) =
 {
   Class(UmlClass);
   find isUmlClassAndSuperClass(UmlClass,UmlSuperClass);
   Class(UmlSuperClass);
   find isClassWithConcept(UmlSuperClass,SuperConcept,Ref);
   concept(SuperConcept);
   find isClassWithConcept(UmlClass,Concept,Ref);
   concept(Concept);
   neg find conceptImplication(Concept,SuperConcept);
 }
 postcondition find conceptImplication(Concept,TopConcept,Trg);
}
```

Fig. 15. Rules for classes in the UML-to-Racer transformation.

construct in the postcondition of the rule denotes that the contents of the called pattern are simply copied to the postcondition pattern (and merged appropriately with existing objects like Class(C)).

Rules *topClassImplication* and *classImplication* (Fig. 15) create the implication (inheritance) relations between the concepts. The first connects the top classes (without superclass) to the special root concept *TOPC*, the second one connects the concepts related to child classes to the concepts of their parents.

Rule *assoc2roles* (Fig. 16) maps associations ends to Racer model elements. The association ends are mapped to concepts, and they are connected to the associated classes by roles. The created concepts are children of the *TOPA* root concept. The action part of the rule sets the names of the newly created model elements for better readability.

The *main* ASM rule is the entry point of the UML-to-Racer transformation (Fig. 17). It takes the source, reference and target models as input, creates the top-level concepts for classes and associations in the Racer model, and calls each graph transformation rule in forall in the given sequence.

7. Related work

Since VIATRA2 can also be regarded as a graph transformation tool, we now compare the advanced language constructs to similar constructs available in the most popular graph transformation tools, namely, AGG [14], ATOM3 [11], FUJABA [18], GReAT [16], PROGRES [23], VMTS [17] and VIATRA2 [3]. In addition, we also include ATL [5] in our comparison as an advanced model transformation language that is not built on graph transformation principles.

More specifically, we compare in Fig. 18 the advanced constructs for specifying *patterns* (such as negative application conditions, multi-objects, path expressions, constraints), *control structures* (such as iterative and

```
gtrule assoc2role(in UmlAssocEnd, in Trg, in Ref, in TopConcept) = {
 precondition pattern pre(UmlAssocEnd,UmlClass,ConceptClass,Ref) =  {
  AssociationEnd(UmlAssocEnd);
  neg find isAssocWithConcept(UMLAssocEnd,Conc,Ref);
  find isUmlAssocEndAtClass (AssocEnd, UmlClass);
  Class(UmlClass);
  find isClassWithConcept(UmlClass,ConceptClass);
  concept(ConceptClass);
 }
 postcondition pattern post(UmlAssocEnd,ConceptClass,
   Conc,RoleA1,RoleA2,Ref,Trg,TopConcept) =
 {
// Precondition elements to be preserved
  AssociationEnd(UmlAssocEnd);
  concept(TopConcept);
  concept(ConceptClass);
// Reference model: created
  find isAssocWithConcept(UMLAssocEnd,Conc,Ref);
// Target model
 // Concepts and implications: created
  concept(Conc) in Trg;
  find conceptImplication(Conc,TopConcept,Trg);
 // Roles
  role(RoleA1) in Trg;
  find conceptRole(RoleA1,ConceptClass,Conc,Trg);
  role(RoleA2) in Trg;
  find conceptRole(RoleA2,Conc,ConceptClass,Trg);
  role.inv(I1,RoleA1,RoleA2);
 }
 action
 {
  rename(Conc,"Assoc_"+name(UmlClass)+"_"+name(UmlAssocEnd));
  rename(RoleA1,name(UmlClass)+"_"+name(UmlAssocEnd)+"_to");
  rename(RoleA2,name(UmlClass)+"_"+name(UmlAssocEnd)+"_from");
 }
}
```

Fig. 16. Rules for associations in the UML-to-Racer transformation.

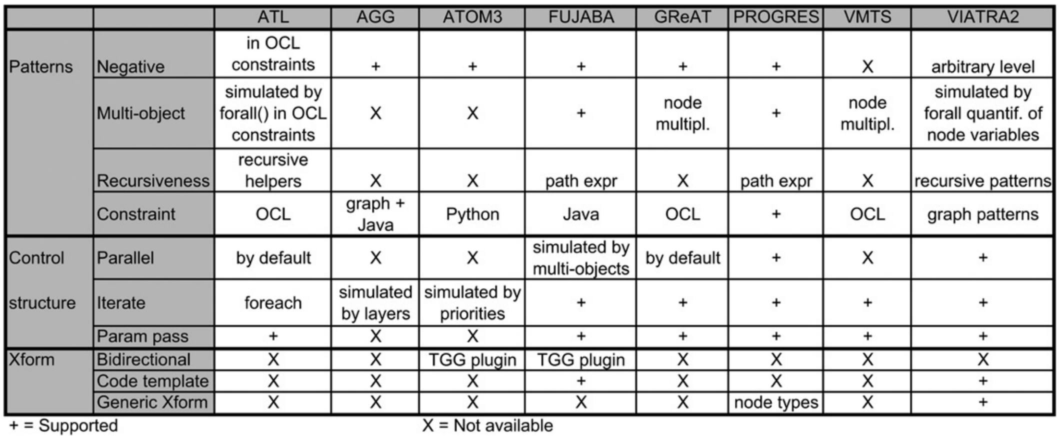
```
// Main rule: Uml, Ref and Racer are model containers
rule main(in UmlM, in RefM, in RacerM) = seq {
  //create top-most concepts
  new(concept(Topc) in RacerM);
  rename(Topc,"TOPC");
  new(concept(Topa) in RacerM);
  rename(Topa,"TOPA");
  //creating concepts representing UML classes
  forall Class below UmlM with apply class2concept(Class,Racer,Ref);
  //creating concepts and roles representing UML association ends
  forall AssocEnd below UmlM with apply assoc2roles(AssocEnd,Racer,Ref,Topa);
  //creating class hierarchy: top-level classes
  forall Class below UmlM with apply topClassImplication(Class,Topc,Racer,Ref);
  //creating class hierarchy: other classes
  forall Class below UmlM, SuperClass below UmlM with
         apply classImplication(Class,SuperClass,Racer,Ref);
 }
}
```

Fig. 17. ASM control structures.

parallel rule applications, or parameter passing between rules), and general *transformation features* (bidirectional transformations, template-based code generation or generic transformations). Obviously, the corresponding ATL constructs are only approximately "equivalent" due to the differences between the graph-centric and object-centric paradigms of these transformation languages.

We would like to point out that VIATRA2 provides a more general support than any of these graph transformation tools in the following areas:

- Pattern calls in VIATRA2 facilitate the *reusability of elementary patterns* to construct complex patterns and rules. Fully declarative pattern calls in both LHS and RHS of GT rules are a novel feature compared to other GT tools. In



		ATL	AGG	ATOM3	FUJABA	GReAT	PROGRES	VMTS	VIATRA2
Patterns	Negative	in OCL constraints	+	+	+	+	+	X	arbitrary level
	Multi-object	simulated by forall() in OCL constraints	X	X	+	node multipl.	+	node multipl.	simulated by forall quantif. of node variables
	Recursiveness	recursive helpers	X	X	path expr	X	path expr	X	recursive patterns
	Constraint	OCL	graph + Java	Python	Java	OCL	+	OCL	graph patterns
Control structure	Parallel	by default	X	X	simulated by multi-objects	by default	+	X	+
	Iterate	foreach	simulated by layers	simulated by priorities	+	+	+	+	+
	Param pass	+	X	X	+	+	+	+	+
Xform	Bidirectional	X	X	TGG plugin	TGG plugin	X	X	X	X
	Code template	X	X	X	+	X	X	X	+
	Generic Xform	X	X	X	X	X	node types	X	+

+ = Supported X = Not available

Fig. 18. Comparison of advanced language constructs in model transformation tools.

practice, it turns out to be a very powerful technique to decompose complex model transformations into reusable pieces.

- VIATRA2 allows *negative conditions with arbitrary depth of negation*. This way, negative patterns are not allowed to have negative patterns in turn. Other GT tools offer negative conditions with a single depth of negation.
- Graph patterns in VTCL allow for *full recursion*. The best existing solution for recursion (used in PROGRES and FUJABA) is called path expressions, which are basically regular expressions over edges. This case structural recursion appears only on a single path of edges.
- Only the transformation language of VIATRA2 supports *generic transformations* (transformation rules having metamodel-level type parameters and dynamic retyping of model elements) and *meta-transformations* (rules manipulating other rules). This allows to create highly reusable libraries for generic algorithms which are applicable to different metamodels.

On the other hand, bidirectional transformations (provided by triple graph grammars [22] in FUJABA and ATOM3) are not supported in VIATRA2 . For all other language features, the VIATRA2 solution is (at least) comparable to some existing approaches available in graph transformation tools. Finally, the approach of integrating graph transformation and ASMs [26] is quite novel.

For other related model transformation tools, it is worth mentioning MT [24], which uses a powerful pattern language with multiplicities for pattern objects. On the one hand, such multiplicities in MT provide a finer control over matching of a single object, i.e. to have exactly five matches for an element. On the other hand, recursive patterns in VTCL provide more general recursion.

Similar structuring concepts (like rules, patterns templates) are proposed in Tefkat [2]. On the one hand, only simple recursion is supported, and generic patterns are out of scope for Tefkat. On the other hand, the extend and supersede constructs in Tefkat are powerful mechanisms for reusability.

8. Conclusions

We presented the 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 paradigm. In addition, powerful language constructs are provided for multi-level metamodeling to design modeling languages.

After a comparison with the transformation language of leading graph transformation tools, we can conclude that the transformation language of VIATRA2 offers advanced constructs for querying (e.g. recursive graph patterns) and manipulating models (e.g. generic transformation and meta-transformation rules) in unidirectional transformations which are frequently used in formal model analysis to carry out abstractions.

The VIATRA2 transformation language (and the framework) has been successfully applied for the transformation-based dependability analysis of UML and BPM models. Furthermore, it constitutes the core transformation technology of the DECOS and SENSORIA European IPs, and an extensive use of the framework is expected in other European projects started recently.

In the future, we plan further extensions to the VTCL language, based on existing feedback from VIATRA2 users. For instance, we plan to (i) add counters to patterns so that the number of matches could be used in constraints, and (ii) extend recursive rules to allow modifications deep down in the recursion.

Acknowledgements

The paper was partially supported by the DECOS (IST-511764) and SENSORIA (IST-2005-016004) European IST integrated projects, and a Hungarian National Grant (GVOP-3.3.1-05/1-2005-05-0143/3.0). The first author was also supported by the János Bolyai Scholarship. The authors are grateful to Ákos Horváth, Gergely Nyilas, Péter Pásztor, András Pataricza, István Ráth, András Schmidt, Dániel Tóth, Dávid Vágó and Gergely Varró for their help in designing and implementing the VIATRA2 framework to support the presented transformation language. We also thank Balázs Polgár for developing the UML-to-Racer transformation. Finally, we would like to acknowledge the valuable comments of our anonymous reviewers.

References

[1] Eclipse Modeling Framework. http://www.eclipse.org/emf.
[2] Tefkat: The EMF Transformation Engine. http://tefkat.sourceforge.net.
[3] The VIATRA2 Transformation Framework. http://www.eclipse.org/gmt/.
[4] C. Atkinson, T. Kühne, The essence of multilevel metamodelling, in: M. Gogolla, C. Kobryn (Eds.), Proc. UML 2001—The Unified Modeling Language. Modeling Languages, Concepts and Tools, in: LNCS, vol. 2185, Springer, 2001, pp. 19–33.
[5] ATLAS Group. The ATLAS Transformation Language. Available from: http://www.eclipse.org/gmt.
[6] A. Balogh, D. Varró, Advanced model transformation language constructs in the VIATRA2 framework, in: ACM Symposium on Applied Computing — Model Transformation Track, SAC 2006, ACM Press, Dijon, France, 2006, pp. 1280–1287.
[7] A. Balogh, G. Varró, D. Varró, A. Pataricza, Compiling model transformations to EJB3-specific transformer plugins, in: ACM Symposium on Applied Computing — Model Transformation Track, SAC 2006, ACM Press, Dijon, France, 2006, pp. 1288–1295.
[8] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, D. Pollet, Reflective model driven engineering, in: P. Stevens, J. Whittle, G. Booch (Eds.), Proc. UML 2003: 6th International Conference on the Unified Modeling Language, in: LNCS, vol. 2863, Springer, San Francisco, CA, USA, 2003, pp. 175–189.
[9] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, G. Savoia, Dependability analysis in the early phases of UML based system design, International Journal of Computer Systems—Science & Engineering 16 (5) (2001) 265–275.
[10] E. Börger, R. Stärk, Abstract State Machines. A Method for High-Level System Design and Analysis, Springer-Verlag, 2003.
[11] J. de Lara, H. Vangheluwe, AToM3: A tool for multi-formalism and meta-modelling, in: R.-D. Kutsche, H. Weber (Eds.), 5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, 8–12 April, 2002, Proceedings, in: LNCS, vol. 2306, Springer, 2002, pp. 174–188.
[12] DECOS. Dependable Components and Systems. An FP6 Integrated Project. http://www.decos.at.
[13] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook on Graph Grammars and Computing by Graph Transformation, in: Applications, Languages and Tools, vol. 2, World Scientific, 1999.
[14] C. Ermel, M. Rudolf, G. Taentzer, The AGG-Approach: Language and Tool Environment, in: [13], World Scientific, 1999, pp. 551–603.
[15] Á. Horváth, D. Varró, G. Varró, Generic search plans for matching advanced graph patterns, in: Proc. Graph Transformation and Visual Modelling Techniques, GT-VMT 2007, Electronic Communications of the EASST, Volume 6, 2007, URL: http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/viewFile/49/54.
[16] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, On the use of graph transformation in the formal specification of model interpreters, Journal of Universal Computer Science (2003).
[17] T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf, A systematic approach to metamodeling environments and model transformation systems in VMTS, in: Proc. GraBaTs 2004: Workshop on Graph Based Tools, Elsevier, 2004.
[18] U. Nickel, J. Niere, A. Zündorf, Tool demonstration: The FUJABA environment, in: The 22nd International Conference on Software Engineering, ICSE, ACM Press, Limerick, Ireland, 2000.
[19] Object Management Group. Meta Object Facility Version 2.0. http://www.omg.org, 2003.
[20] Racer Systems Gmbh. Racerpro. http://www.racer-systems.com.
[21] A. Rensink, Representing first-order logic using graphs, in: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (Eds.), Proc. 2nd International Conference on Graph Transformation, ICGT 2004, Rome, Italy, in: LNCS, vol. 3256, Springer, 2004, pp. 319–335.
[22] A. Schürr, Specification of graph translators with triple graph grammars, in: B. Tinhofer (Ed.), Proc. WG94: International Workshop on Graph-Theoretic Concepts in Computer Science, in: LNCS, vol. 903, Springer, 1994, pp. 151–163.
[23] A. Schürr, A.J. Winter, A. Zündorf, The PROGRES Approach: Language and Environment, in: [13], World Scientific, 1999, pp. 487–550.
[24] L. Tratt, The MT model transformation language, in: Proceedings of the 2006 ACM Symposium on Applied Computing, SAC, Dijon, France, ACM, 2006, pp. 1296–1303.

[25] D. Ullman, Principles of Database and Knowledge Base Systems, in: Deductive Databases, Computer Science Press, 1988.

[26] D. Varró, Automated model transformations for the analysis of IT systems, Ph.D. Thesis, Budapest University of Technology and Economics, 2004.

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

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