Advanced Model Transformation Language Constructs in the VIATRA2 Framework

András Balogh
abalogh@mit.bme.hu

Dániel Varró
varro@mit.bme.hu

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

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 and meta transformation rules) in unidirectional model transformations frequently used in formal model analysis to carry out powerful abstractions. In addition, powerful language constructs are provided for multi-level metamodeling to design modeling languages and template-based code generation.

1. INTRODUCTION

The crucial role of model transformation (MT) languages and tools for the overall success of model-driven systems development have 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, Object Management Group (OMG) is soon to issue the QVT standard.

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

However, there is a large set of model transformations, which are unidirectional by nature. 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 [6]) have chosen a more intuitive language for unidirectional transformations.

Typical examples of abstraction transformations can be identified during model validation by formal analysis [8]. The VIATRA2 model transformation framework primarily aims at designing model transformations to support the precise model-based systems development with the help of invisible formal methods. Invisible formal methods are hidden by automated model transformations projecting system models into various mathematical domains (and, preferably, vice versa).

VIATRA2 have chosen to integrate two popular, intuitive, yet mathematically precise rule-based specification formalisms, namely, graph transformation (GT) [11] and abstract state machines (ASM) [9] 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 to carry out the validation of transformations. Then model transformation rules can be compiled into efficient, platform-specific transformer plugins for optimal execution.

In the current paper, we present the textual VIATRA2 transformation language (VTL) which is composed of three sublanguages to provide support for multilevel metamodeling (Sec. 2), pattern and rule-based model transformations (Sec. 3) and template-based code generation (Sec. 4).

While there is a large set of tools based on the paradigm of graph transformation (see Sec. 6) available for model transformation purposes, the specification language of VIATRA2 uniquely provides a combined support for (i) multilevel metamodeling (ii) recursive graph patterns, (iii) generic and meta-transformations, and (iv) template based code generation techniques.

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 formalism called metamodeling language, which is another modeling language to capture metamodels.

Currently, most widely used metamodeling languages (e.g. Eclipse Modeling Framework (EMF) [2]) are derived (with slight variations) from the Meta Object Facility (MOF) [16] metamodeling standard issued by the OMG. However, as stated in [21], the MOF standard fails to support multi-level metamodeling [5], which is typically a critical aspect for integrating different technological spaces [7] where different metamodeling paradigms (e.g. EMF, XML Schemas, EBNF grammars) are used.

2.1 Concepts of VPM

Therefore, for generality reasons, we have chosen to use the VPM (Visual and Precise Metamodeling) [21] metamodeling approach in the VIATRA2 framework, which can support different metamodeling paradigms by supporting multi-level metamodeling with explicit and generalized instance-of relations. Naturally, standard metamodeling paradigms are integrated into VIATRA2 by import plugins.

As only an abstract syntax was defined for VPM in [21], a human readable representation (concrete syntax) of VPM metamodels and models was indispensable to be elaborated. We developed two alternative syntaxes, a textual and a graphical, for the metamodeling environment. Our experience showed that the textual format was more usable in case of large, or automatically generated metamodels, while the graphical notation can easily be understood by engineers who are familiar with the MOF/UML language. In the current paper, we mainly focus on the textual language called VTML (Viatra Textual (Meta)Modeling Language).

The VPM language 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 the target (modeling) domain, while *relations* represent the relationships between other model elements [1]. Furthermore, entities may also have an associated value which is a string property that contain application-specific data.

Model elements are arranged into a strict containment hierarchy, which consititutes the VPM model space. Within a container entity, each model element has a unique local name, and each model element can be uniquely identified globally by a fully qualified name (FQN).

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. (ii) The *inverse* relation points to the inverse of the relation (if any). In a UML analogy, a relation can be considered as an association end, and thus the inverse of a relation denotes the other association end along an association. (iii) Relations also have a *multiplicity*, which imposes a restrictions on the model structure. Allowed multiplicity kinds in VPM are one-to-one, one-to-many, many-to-one, and many-to-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.

There are two special relationships between model elements: 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.

2.2 The VTML language

In VIATRA2 , the textual metamodeling language supporting VPM is called VTML (Viatra Textual Metamodeling Language). The technicalities of VTML are demonstrated in Fig. 1 on a simplified UML metamodel presented originally in the model transformation benchmark of [3].

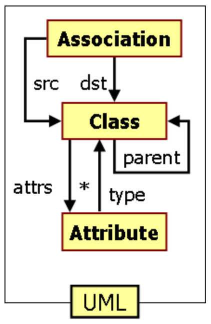

Figure 1: Sample UML metamodel

The VTML equivalent of the metamodel is as follows.

```
entity(UML)
{
   entity(Class);
   entity(Association);
   entity(Attribute);
   relation(src,Association,Class);
   relation(dst,Association,Class);
   relation(parent,Class,Class);
   relation(attrs,Class,Attribute);
   multiplicity(attrs,many-to-many)
   relation(type,Attribute,Class);
}
```

The basic elements of the language are the element declarations defined as Prolog-like facts. An entity can be declared in the form *<type> (<name>)*, where *type* is the type of the given entity and *name* is the name of the new entity. Type declarations are mandatory, because all entities must have a type. If an entity has no definite type, it has to instantiate from the basic VPM entity model element. As entities may contain other model elements, the containment must also be represented. This is done similarly to the C language, where the program blocks are marked with braces ({}). Here, the contained elements are represented in block surrounded by curly brackets after the container entity.

A relation can be defined similarly, but the source and target model elements must also be marked. The syntax of relation definition is the following: <type> (<name> , <source> ,

<target>). A relation is always contained by its source entity.

The containment hierarchy defines namespaces in the model space. This enables the definition of the fully qualified name (FQN) of model elements. The FQN equals to the list of containers of a given model element from the model space root to the element, separated by dots. For example, the FQN of the entity Association in the example is UML.Association, while the FQN of the relation src is UML.Association.src. The local (short) name of a model element must be unique in its container, this also ensures the uniqueness of FQNs.

Special relationships can be represented with keywords supertypeOf, and subtypeOf for specialization relations, and typeOf, and instanceOf for instantiation. The syntax is the following: <relationship> (<supplier> , <client>). For example, typeOf(UML.Class, Dog) defines that the entity Dog to an instance of the metamodel element UML.Class. This way, a model element may have multiple types to support multi-domain modeling.

The VTML language has also a *namespace import* definition that can be used to import namespaces for the given VTML file. Model elements that are in the imported namespaces can be referred to using their local names, instead of the default fully qualified names for user's convenience. Namespace import has the following syntax: import <namespace>; For example, if we want to create an instance model using the simplified UML metamodel above, we can use the import UML; command to import the UML namespace. After that, elements of the metamodel (like UML.Association) can be referred to using their local names (as Association).

3. MODEL TRANSFORMATION LANGUAGE

Transformation descriptions in VIATRA2 consist of several constructs that together form an expressive and comfortable language for developing both model to model transformations and code generators. Graph patterns (GP) defined constraints and conditions on models, graph transformation (GT) [11] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [9] rules can be used for the description of control structures.

The language that is created to implement all these concepts is the Viatra Textual Command Language (VTCL). This language is primary textual, but it will soon be extended by a graphical editor that will support the graphical definition of model transformations.

3.1 Graph patterns

3.1.1 Graph patterns, negative patterns

Graph patterns are the atomic units that are necessary for model transformations. They represent conditions (or constraints) that a part of the model space has to fulfill in order to execute some manipulation steps on the model.

A model (i.e. part of the model space) can satisfy a pattern graph, if the pattern can be matched to a subgraph of the model using a generalized *graph pattern matching* technique first presented in [20].

In the following example, a simple pattern will be introduced that defines a condition that can be fulfilled by class instances that do not have parent classes.

```
/* C is a class without parents and
   with non-empty name */
pattern isTopClass(C) =
{
   UML.Class(C);
   neg pattern negCondition(C) =
   {
         UML.Class(C);
         UML.Class.parent(P,C,CP);
         UML.Class(CP);
   }

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

Patterns are defined using the *pattern* keyword. Patterns may have parameters, that are listed after the pattern name. The basic pattern body contains model element and relationship definitions, which are identical to the VTML language constructs.

The keyword *neg* marks a new pattern that is embedded to the current one, and represents a negative condition to the pattern. The negative pattern in the example 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.

There are also *check* conditions that are Boolean formulae which must be satisfied in order to make the pattern true. In our example, we check whether the name of the class is empty. Only classes with valid names can match the pattern.

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 [17].

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 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. The two features (pattern call and alternate (OR) bodies) can be used together for the definition of *recursive pattern.* In a typical recursive pattern, one of the bodies contains a recursive call to itself, and the other defines the stop condition for the recursion. The following example illustrates the usage of recursion.

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

A class Parent is the parent of an other class Child, if it is a direct parent of the child class (first body), or it has a direct child (C), which is the parent of the child class (second body). The pattern uses recursion for discovering multi-level parent-child relationships, and multiple bodies to create a stop condition for the recursion.

3.1.3 On the semantics of graph patterns

When a predefined graph pattern is called (using the *find* keyword) this means that a substitution for the free (unbound) parameters of the specified graph pattern has to be found that satisfies the pattern. If there are defined variables in the parameter list, they will be treated as additionally constraints, and remain substituted (bound) throughout the pattern matching process. By default, the free variables will be substituted by *existential quantification*, which means that only one (non-deterministically selected) matching will be generated. If a variable is universally quantified by the external forall construct (see Sec. 3.3), the matching will be done (in parallel) for all possible values of the given variable.

3.2 Graph transformation rules

While graph patterns define logical conditions (formulas) on models, the manipulation of models is defined by graph transformation rules [11], 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.

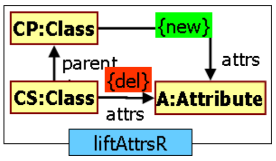

Figure 2: Sample graph transformation rule

The sample graph transformation rule in Figure 2 defines a refactoring step of lifting an attribute from child to parent classes. This means that if the child class has an attribute that is not present in the parent class, it will be lifted to the parent.

The VTCL language allows both popular notation for defining graph transformation rules. The first syntax of a GT rule specification corresponds to the traditional notation: it contains two a *precondition* pattern for the LHS, and a *postcondition* pattern that defines the RHS of the rule. Elements that are present only in the LHS are deleted, elements that are present only in RHS are created, and other model elements remain unchanged.

```
gtrule liftAttrsR(in CP, in CS, in A) =
{
   precondition pattern cond(CP,CS,A,Attr) =
   {
      UML.Class(CP);
      UML.Class(CS);
      UML.Class.parent(Par,CS,CP);
      UML.Attribute(A);
      UML.Class.attrs(Attr,CS,A);
   }
   postcondition pattern rhs(CP,CS,A,Attr) =
   {
      UML.Class(CP);
      UML.Class(CS);
      UML.Class.parent(Par,CS,CP);
      UML.Attribute(A);
      UML.Class.attrs(Attr2,CP,A);
   }
}
```

The graph transformations rules are defined using the *gtrule* keyword, and they are allowed to have directed (in/out/inout) parameters. The LHS and RHS patterns share information by parameter passing (instead of traditional shared matchings).

The second format directly corresponds to the graphical (FUJABA [15]) notation as shown in the following example.

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

The rule contains a simple pattern (marked with keyword *condition*), that 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 done (and therefore do not participate in the pattern matching), and elements marked with keyword *del* are deleted after pattern matching.

In both cases, further actions can be initiated by calling any ASM instructions within the action part of a GT rule, e.g. to report debug information or to generate code.

3.2.1 Generic and meta-transformations

To provide algorithm-level reuse for common transformation algorithms independent of a certain metamodel, VIATRA2 supports generic and meta-transformations, which is 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(in CP, in CS, in A,
              in ClsE, in AttE, in ParR, in 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);

   }
}
```

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 corresponding for edge types.

When interpreting this generic pattern, the VIATRA engine first instantiates the type parameters (e.g. ClsE, and ParR) and then queries all the instances of these types. As a result, the same set of rules can be applied in various modeling languages.

3.2.2 *Invoking graph transformation rules*

To execute graph transformation rules they have to be invoked from a transformation program. The basic invocation is done using the *apply* keyword. In this 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. If a rule can be executed for all possible matches (in parallel) by quantifying some of the input parameters using the forall construct. The following example illustrates some possible invocations of our sample rule.

```
//simple execution of a GT rule
//all variables must have a bound value
apply liftAttrsR(Class1,Class2,Attrib);
```

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

//calling the rule for all possible matches
forall C1, C2, A do apply liftAttrsR(C1,C2,A);
```

3.3 Control Structure

To control the execution order and mode of graph transformation the VTCL language includes some concepts that support the definition of complex control flow. As one of the main goals of the development of VTCL was to create a precise formal language, we included the basic set of Abstract State Machine (ASM) language constructs [9] that have formal semantics and are very similar 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 arrays. These values can be updated from the ASM program. These functions are called *dynamic*. There are also function that are *static*, that means they cannot change their values. For example, the basic mathematical functions (+,-,*,/) are static.

We have also defined a special class of functions that are called *native functions*. 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 that can help the execution of model transformations.

ASMs provide complex model transformations with all the necessary 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).

These basic instructions, combined with graph patterns and graph transformation rules, form an expressive, easy-to-use, yet mathematically precise language where the semantics of graph transformation rules are also given as ASM programs (see [20]for more details). The following example demonstrate the main control structures.

```
pattern isClass(C) =
{
   //simple pattern that recognizes classes
   UML.Class(C);
}
rule main() = seq
{
   //print out some text
   print("The transformation begins...");
   //call a GT rule for all matches
   forall C1, C2, A do apply liftAttrsR(C1,C2,A);
   //Calling other rule
   call printFormatted(123);
   //Iterating through all classes
   forall Cl with find isClass(Cl) do seq
   {
      print("Found a class: "+name(Cl));
   }
   //Writing to log
   log(info,"transformation done");
}
rule printFormatted(in C) =
{
   //printing out the value
   print("Value is  : "+C);
}
```

4. CODE TEMPLATES

An important type of model transformations are model-to-code transformations. These programs generate textual output (for example source code for a programming language, or XML-based descriptors) from models. To support these transformations VTCL includes the *print* instruction that prints out text to the output. This floating text can be formatted by the VIATRA code output formatter modules. For example, Java requires the separation of classes into files, and building a directory structure that reflects the package structure. A special code output module can render the source code to the required format.

While using print is easy, it may be inconvient in cases where the code generation includes large blocks of static text (that is unchanged), and small amount of dynamic data, which is typical to many code generation problems. To ease the job of the code generator designer, VIATRA2 includes the concepts of *code template*s. This language extension is called Viatra Textual Template Language (VTTL). A template is a text block that can include references to ASM variables and instructions. The pattern concept is very similar to the one introduced in the Apache Velocity [1] language, but uses the formal ASM and GT paradigms as the control language. The next example gives a bried insight into the VTTL language.

```
template printClass(in C) =
{
public class $C {
#(forall At,Typ with attrib(C,At,Typ) do seq{()
 private $Typ $At;
#()})
}
}
```

The sample template receives an input parameter (that points to a class instance). It creates the skeleton of the class in Java. ASM variables can be referenced in templates using the $ character. ASM instructions (as the forall that iterated through all attributes of the class) can be referenced using the #() marking. All possible ASM constructs can be used in templates.

5. VIATRA2 AS A QVT TOOL

As VIATRA2 can be considered as a general model transformation tool, first we discuss to what an extent the VIATRA2 transformation language fulfills the main (mandatory and optional) requirements of the QVT RFP.

Man1 *Proposals shall define a query language.* VIATRA2 uses recursive *graph patterns* as its query language, which provides the same expressive power as using OCL as a query languages in QVT.

Man2 *Proposals shall define a language for transformation definitions between a source metamodel S and target metamodel T.* VIATRA2 uses the combination of two rule-based paradigm (graph transformation and abstract state) as a transformation language between two metamodels.

Man3 *The abstract syntax of transformations shall be defined by a MOF metamodel.* Transformations have a metamodel and they are stored as ordinary models in the VIATRA2 model space.

Man4 *The transformation description language shall be capable of generating the target model from a source model automatically.* VIATRA2 provides an interpreter for the previous transformation descriptions, moreover, these descriptions can be compiled into efficient platfrom-specific transfomer plugins.

Man5 *The transformation definition language shall enable the creation of a view of a metamodel.* This feature is not supported.

Man6 *Incremental changes in the source model may be transformed into changes in the target model immediately (declarative transformations).* Incremental model transformations can be achieved by using transformation rules with negative conditions which can be matched only new model elements. However, incremental transformations are not typical in model validation (which is the main application field of VIATRA2), since model validation is usually a very complex and time consuming task. Thus it is impractical to transform the source model into a mathematical domain and reexecute the validation process each time the source model is changed.

Man7 *Proposals shall operate on model instances defined using MOF.* Due to its support for multilevel metamodeling, VIATRA2 can accept models defined using the MOF metamodeling framework, but it is also possible to use other approaches of language design typical to another technological space (e.g. XML Schemas).

Opt1 *Proposals may support transformations that can be executed in two directions.* VIATRA2 supports only unidirectional transformations (which are typical in abstraction transformations used for model validation. The support for truly declarative (bidirectional) transformations is an ongoing activity.

Opt2 *Transformations may support the traceability of transformation executions.* VIATRA2 supports traceability by a log rule, and the explicit storage of reference structures in the model space.

Opt3 *Proposals may support mechanisms for reusing and extending generic transformation definitions.* VIATRA2 facilitates reuse by arranging models and transformations into Java-like libraries, moreover, it supports generic and meta-transformations [22] for algorithm-level reuse.

Opt4 *Proposals may support transactional transformation definitions.* Transactions are supported in VIATRA2 on the tool-level by undo support and on the execution-level by generating EJB3-based transformation plugins where the underlying database is responsible for transactional support.

Opt6 *Proposals may support the definition of transformations where the target model is the same as the source model.* Any number of modeling languages may be involved in VIATRA2 transformations, which includes intra-model transformations as well.

QVT1 *QVT provides support for black-box implementations.* Native Java methods can be embedded into transformations as ordinary ASM functions.

As a summary, VIATRA2 does not currently meet only two QVT RFP requirements: (i) support for declarative, and bidirectional transformations and (ii) support for view generation (this is not supported by QVT either. On the other hand, we strongly believe that VIATRA2 provides better support than the QVT standard for (i) reusable, generic transformations, (ii) unidirectional, and intra-model transformations.

6. 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 [12], ATOM3 [10], FUJABA [15], GReAT [13], PROGRES [19], VMTS [14] and VIATRA2 [4]. In the future we plan to extend this detailed comparison with other model transformation approaches not based on graph transformation, which was out of scope for the current paper due to space limitations.

More specifically, we compare in Table 1 the advanced constructs for specifying *patterns* (such as negative application conditions, multi-objects, path expressions, constraints), *control structures* (such as iterative and parallel rule applications, or parameter passing between rules), and general *transformation features* (bidirectional transformations, template based code generation or generic transformations).

As a conclusion, we would like to point out that VIATRA2 provides a more general support than any of these graph transformation tools in the following sense:

- Only VIATRA2 allows negative conditions with arbitrary levels of negation.
- Recursive graph patterns are true generalizations of path expressions (of PROGRES and FUJABA) where 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 meta transformations (rules manipulating other rules).
- VIATRA2 uniquely provides a template-based code generation method for model-to-code transformations.

On the other hand, bidirectional transformations (provided by triple graph grammars [18] in FUJABA and ATOM3) are not supported in VIATRA2 . For all other language features, the VIATRA2 solution is comparable to some existing approaches available in graph transformation tools, although the approach of integrating graph transformation and ASMs [20] is quite novel.

Obviously, the selection of relevant features for a comparison can never be complete, our main purpose with this comparison is to highlight the fact that the transformation language of VIATRA2 is as powerful as the language of the leading graph transformation tools.

7. 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 specification paradigm. In addition, powerful language constructs are provided for multi-level metamodeling to design modeling languages and template-based code generation.

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

While the paper probably lacks a convincing benchmark example, it is worth pointing out that 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 main transformation technology of the DECOS European IP, and an extensive use of the framework is foreseen in three other European projects starting with the next four months.

8. REFERENCES

[1] Apache Velocity Project. `http://jakarta.apache.org/velocity`.

[2] Eclipse Modeling Framework. `http://www.eclipse.org/emf`.

[3] Model transformations in practice workshop. `http://sosym.dcs.kcl.ac.uk/events/mtip/`.

[4] *VIATRA2 Framework.* An Eclipse GMT Subproject (`http://www.eclipse.org/gmt/`).

[5] C. Atkinson and T. Kühne. The essence of multilevel metamodelling. In M. Gogolla and C. Kobryn (eds.), *Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools*, vol. 2185 of *LNCS*, pp. 19–33. Springer, 2001.

[6] ATLAS Group. *The ATLAS Transformation Language.*

		AGG	ATOM3	FUJABA	GReAT	PROGRES	VMTS	VIATRA2
Patterns	Negative	+	+	+	+	+	X	arbitrary level
	Multi-object	X	X	+	node multipl.	+	node multipl.	simulated by forall quantif. of node variables
	Path expr	X	X	+	X	+	X	recursive patterns
	Constraint	graph + Java	Python	Java	OCL	+	OCL	graph + ASM
Control structure	Parallel	X	X	simulated by multi-objects	by default	+	X	+
	Iterate	simulated by layers	simulated by priorities	+	+	+	+	+
	Param pass	X	X	+	+	+	+	+
Xform	Bidirectional	X	+	+	X	X	X	X
	Code template	X	X	X	X	X	X	+
	Generic Xform	X	X	X	X	X	X	+

+ = Supported X = Not available

Table 1: Comparison of advanced language constructs in graph transformation tools

[7] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. In P. Stevens, J. Whittle, and G. Booch (eds.), *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, vol. 2863 of *LNCS*, pp. 175–189. Springer, San Francisco, CA, USA, 2003.

[8] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, vol. 16(5):pp. 265–275, 2001.

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

[10] J. de Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber (eds.), *5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, vol. 2306 of *LNCS*, pp. 174–188. Springer, 2002.

[11] 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.

[12] C. Ermel, M. Rudolf, and G. Taentzer. *In [11]*, chap. The AGG-Approach: Language and Tool Environment, pp. 551–603. World Scientific, 1999.

[13] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 2003.

[14] T. Levendovszky, L. Lengyel, G. Mezei, and H. Charaf. A systematic approach to metamodeling environments and model transformation systems in vmts. In *Proc. GraBaTs 2004: International Workshop on Graph Based Tools*. Elsevier, 2004.

[15] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*. ACM Press, Limerick, Ireland, 2000.

[16] Object Management Group. *Meta Object Facility Version 2.0*, 2003. `http://www.omg.org`.

[17] 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.

[18] 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*, no. 903 in LNCS, pp. 151–163. Springer, 1994.

[19] A. Schürr, A. J. Winter, and A. Zündorf. *In [11]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.

[20] 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.

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

[22] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor (eds.), *Proc. UML 2004: 7th International Conference on the Unified Modeling Language*, vol. 3273 of *LNCS*, pp. 290–304. Springer, Lisbon, Portugal, 2004.