Applying a Model Transformation Taxonomy to Graph Transformation Technology

Tom Mens [1]

Software Engineering Lab
Université de Mons-Hainaut
Mons, Belgium

Pieter Van Gorp [2]

Formal Techniques in Software Engineering
Universiteit Antwerpen
Antwerpen, Belgium

Dániel Varró [3]

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

Gabor Karsai [4]

Institute for Software-Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Abstract

A taxonomy of model transformations was introduced in [16]. Among others, such a taxonomy can help developers in deciding which language, forma lism, tool or mechanism is best suited to carry out a particular model transformation activity. In this paper we apply the taxonomy to the technique of graph transformation, and we exemplify it by referring to four representative graph transformation tools. As a byproduct of our analysis, we discuss how well each of the considered tools carry out the activity of model transformation.

Key words: model transformation, taxonomy, graph transformation, MDA, MDD, MDE

1 Introduction

A *taxonomy* of model transformation was introduced in [16]. By *taxonomy* we mean "A system for naming and organizing things [...] into groups which share similar qualities" (Cambridge Dictionaries Online). Such a taxonomy can be used for a wide variety of purposes. Among others, it can help a software developer choosing a particular model transformation approach that is best suited for his needs, it can help tool builders to assess the strengths and weaknesses of their tools compared to other tools, and it can help scientists to identify limitations across tools or technology that need to be overcome by improving the underlying techniques and formalisms.

In this paper we use the taxonomy to study the technique of graph transformation, in order to find out what are the merits and drawbacks of graph transformation technology with respect to the activity of model transformation. We exemplify this by applying the taxonomy to a representative subset of tools relying on graph transformation technology. As a byproduct of this analysis, we identify how well these tools carry out the activity of model transformation.

Because graph transformation consists of a large set of different theories, languages, techniques and tools, we need to restrict ourselves to a representative subset somehow. Therefore, we tried to classify existing tools relying on graph transformation technology according to their principal domain of application, and we came up with four important categories:

General-purpose graph transformation tools include *AGG* [5] [26] and PROGRES [24];

Reengineering tools based on graph transformation include *Fujaba* [6] (From UML to Java And Back Again) [2] that supports round-trip engineering between UML and Java, and *VARLET* [7] that performs information system reengineering based on triple graph grammars;

Model transformation tools that make use of graph transformation include *GReAT* [25] and *MOLA* [10];

Model checking and verification tools that rely on graph transformation include *VIATRA* (VIsual Automated model TRAnsformations) [4], *GROOVE* [19] and *CheckVML* [20].

For the sake of this paper, we have selected a representative tool from each category, namely *AGG*, *Fujaba*, *GReAT* and *VIATRA*. In the remainder we

[1] Email: tom.mens@umh.ac.be
[2] Email:pieter.vangorp@ua.ac.be
[3] Email:varro@mit.bme.hu
[4] Email:gabor.karsai@vanderbilt.edu
[5] http://tfs.cs.tu-berlin.de/agg/
[6] http://wwwcs.uni-paderborn.de/cs/fujaba/
[7] http://wwwcs.uni-paderborn.de/cs/varlet/

will use these four tools to illustrate the use of the taxonomy of [16]. The structure of the paper follows the main structure of the taxonomy, which is summarised below:

What needs to be transformed into what? This dimension comprises the following criteria:

- Number of source and target models
- Technical space of the transformation (e.g., MDA, XML)
- Endogenous versus exogenous transformations (i.e., within the same language or between languages)
- Horizontal versus vertical transformations (i.e., on the same level of abstraction or across levels of abstractions)
- Syntactical versus semantical transformations (i.e., simple syntactical rewriting or complex transformations that take semantics into account)

Important characteristics of a transformation :

- Level of automation provided by the tool
- Complexity of the transformation
- Preservation of properties by the transformation

Success criteria for a transformation tool :

- Suggesting when to apply transformations
- Customising or reusing transformations
- Verifying and guaranteeing correctness of transformations
- Testing and validating transformations
- Dealing with incomplete or inconsistent models
- Grouping, composing, and decomposing transformations
- Genericity of transformations
- Bidirectionality of transformations
- Supporting traceability and change propagation

Quality requirements for a transformation tool :

- Usability and usefulness
- Verbosity versus conciseness
- Performance and scalability
- Extensibility
- Interoperability
- Acceptability by user community
- Standardization

2 What needs to be transformed into what?

Graphs seem to be a natural representation of models since many models are intrinsically graph-based in nature (e.g., statecharts, activity diagrams, collaboration diagrams, class diagrams, Petri nets), in contrast to source code where a tree structure is likely to be more appropriate (e.g., parse trees, abstract syntax trees). As a consequence, graph transformations seem to a natural means

to specify and execute model transformations.

Number of source and target models.

Graph transformation theory allows us to express *one-to-one model transformations* directly as a *graph production* (also known as graph transformation rule) with a left-hand side (LHS) representing the source model, and a right-hand side (RHS) representing the target model. *One-to-many* transformations (e.g., branching and splitting of models), *many-to-one* transformations (e.g., merging of models) and *many-to-many* transformations can also be defined formally as graph transformations, but this requires a more complex solution.

Triple graph grammars [22], for example, allow the specification of many-to-many model transformations that transform pairs of related models simultaneously while maintaining their consistency. Using this idea, multi graph rewriting rules can be defined that transform any number of (related) source models in a consistent way [13].

Technical space.

All of the considered graph transformation tools allow us to deal with models in the XML or MDA technical space directly, or using some translators.

The *AGG* tool uses GXL [8], a standard graph exchange format as meta-metamodel. However, since GXL is based on XML, it is relatively easy to provide a mapping to the *XML technical space*. *AGG* also supports export into GTXL [9], a standard exchange format for graph transformations.

The *VIATRA* model transformation tool uses an XMI input/output format that conforms to the MOF model. It primarily supports the *MDA technical space*, but business process models and XSD models are also targeted.

The *Fujaba* CASE tool suite supports UML development and automatic generation of Java code. The underlying technology to represent UML diagrams is based on graph transformations. *Fujaba* uses a vendor-specific version of UML, so it can be considered as part of the *MDA technical space*.

The *GReAT* model transformation tool uses UML and OCL notation to specify metamodels and transformations. As such, it belongs to the *MDA technical space*.

Endogenous versus exogenous transformations.

In graph transformation, the structure of a metamodel is described by means of a *type graph* [3]. In an *endogenous* graph transformation, the source and target graphs are instances of the same type graph. This is the case in the *AGG* tool, in which only a single type graph can be specified for a given graph transformation system.

[8] `http://www.gupro.de/GXL/`
[9] http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html

Exogenous transformations, where the source and target are expressed in a different domain (i.e., have a different metamodel) can be expressed using graph transformations, provided that a different type graph can be specified for source and target model. Note that (temporary) edges (so called "cross-links") between nodes belonging to different type graphs might also be required. This is supported by *GReAT* [25], where multiple type graphs (as well as a cross-links) are supported.

Horizontal versus vertical transformations.

Graph transformation technology can be, and has been, used for specifying horizontal as well as vertical model transformations.

As an example of *vertical exogenous* model transformation, Karsai *et al.* show how to transform a platform-independent model (PIM) into a more platform-specific model (PSM) using *GReAT* [11]. As another example, Baresi *et al.* illustrate the use of graph transformation for refinement of software architectures [1].

As an example of *horizontal endogenous* model transformation, graph transformations have been used to specify program refactorings [17,27], and model refactorings in *AGG* and *Fujaba* [15]. A typical example is a statechart flattening, where all composite (nested) states are recursively expanded until one obtains a statechart only containing simple states [7].

As an example of *horizontal exogenous* transformation, graph transformations have been used for migration from one domain-specific language to another using *GReAT* [25].

Finally, Große-Rhode *et al.* [18] illustrate *vertical endogenous* transformations by specifying formal refinements as typed graph transformations.

Syntactical versus semantical transformations.

Graphs and graph transformations can be used to specify and modify the syntactic structure of a model, but also to specify the dynamic behaviour of a model. Even syntactic transformations can take semantic information into account, by specifying graph constraints (invariants) that need to be preserved by the transformation. This is especially needed in the case of model refactoring, where the structure of a model is improved while preserving its behaviour [17,15].

3 Important characteristics of a transformation

Level of automation.

Graph transformation theory offers two flavours of automating the application of a series of graph transformations: *graph grammars* and *programmed graph transformation*.

The graph grammar approach is supported by *AGG*. Starting from an initial graph, all possible applicable graph productions are applied repeatedly, and in parallel. This iterative process is repeated as long as possible. The set of result graphs obtained by this process is called the language generated by the graph grammar.

Programmed graph transformation is supported by *Fujaba*, *VIATRA*, and *GReAT*. In this approach, rules are specified to control the order of graph transformations. *Fujaba* uses so-called *story-driven modelling*, where an activity diagram is used to specify in which order graph transformations should be applied. In *VIATRA*, the graph transformations are driven by abstract state machines as the specification formalism. *GReAT* uses explicitly sequenced transformation rules with input and output parameters that specify the initial bindings for selected pattern variables, as well as allow passing selected graph nodes to subsequent rules. The connectivity between the parameters of rules determines the flow of execution.

Complexity of the transformation.

Complex transformations require more complex control mechanisms to govern the execution order of rules. Graph transformation languages may vary significantly in the strength of these control structures or the way they are specified.

AGG supports layered graph grammars to impose a certain order on the graph production rules to be applied. *Fujaba* uses story diagrams (a kind of activity diagrams) to control the application of graph transformations. In *GReAT*, the control structure is based on hierarchical dataflow-like diagrams (that represent control-flow as well), but an explicit loop control structure is missing. Loops can be expressed as transitions to previous rules in the flow.

Even with these advanced control mechanisms it remains to be seen whether graph transformation alone suffices to express complex transformations. For practical purposes, a hybrid approach combining the virtues of a graph transformation language and a textual constraint language may be more opportune. Given that many of the graph transformation tools rely on UML notation in one way or another, OCL seems to be a viable alternative to specify textual constraints of graph transformations. For instance, *GReAT* uses OCL constraints to impose restrictions on the result of the matching process. An OCL evaluator is available to check whether the result of the transformation complies with the constraints.

Preservation of properties.

Graph transformation theory seems promising to formally specify model refactorings and to show that these refactorings preserve behavioural properties. An initial feasibility study has shown, however, that current graph transformation formalisms lack expressiveness in order to accomplish this goal [17].

Therefore, Van Eetvelde and Janssens [27] proposed a number of extensions to graph transformations in order to enhance their expressive power. Incorporating these extensions in graph transformation tools remains to be done.

4 Success criteria for a graph transformation tool

Suggesting when to apply transformations.

Most graph transformation languages and tools provide a mechanism and language to express constraints on the graphs that need to be transformed. Graph constraints consist of path expressions that state that particular links and / or node values should be present or not.

In *AGG*, one can use the graphical user interface to define constraints as *negative application conditions*, but one can also specify general graph constraints that can be translated into postconditions of graph transformation rules. In *Fujaba*, control constructs can be used to combine graph constraints and graph transformations in arbitrary ways. In *GReAT*, constraints can be used to restrict the results of the graph matching, and zero cardinality on association ends is used to implement a restricted form of negative application conditions.

Customising or reusing transformations.

A simple yet crucial way to customise graph transformations is by means of *parameterisation*. A parameterised graph production represents an infinite set of possible graph productions, each one obtained by providing concrete values for the parameters. A parameterisation mechanism is available in all considered graph transformation tools.

Some graph transformation tools are integrated into an object-oriented development environment, thus allowing to exploit well-known object-oriented mechanisms such as inheritance to enable reuse. For example, in *Fujaba*, graph productions are used as specifications of methods, and inheritance can be used to reuse these methods in subclasses. Similarly, *GReAT* supports reusability via features provided by the visual modeling environment (GME) it is implemented in: transformation rules (or higher-level transformation sequences called *blocks*) can be reused through the type/instance machinery supported by the environment.

In *VIATRA*, a graph transformation can be built up from predefined patterns, that are reusable across transformations. Another important way to enhance the reusability of transformations is by making use of *rule inheritance* [28]. Unfortunately, this mechanism is not yet supported in the considered tools.

Verifying and guaranteeing correctness of the transformations.

A model is syntactically correct if it conforms to its metamodel structure and well-formedness rules. In graph transformation, structure conformance is enforced by means of type graphs. Type graphs may include cardinality constraints, and may also support inheritance (as in the UML metamodel). The type graph mechanism is supported in all four considered tools.

Another way to enforce syntactical correctness is by defining a dedicated graph grammar that contains rules to be used for syntax-directed editing.

The *AGG* tool can check termination and consistency of a graph grammar based on graph constraints. More specifically, it is the only available tool that implements the mechanism of *critical pair analysis* to check termination and confluence of graph grammars [9]. Two graph productions may form a *critical pair* if they are in conflict, in the sense that they do not preserve the confluence property. This property is needed to guarantee that a rewriting system has a functional behaviour.

In *VIATRA*, graph transformations (of UML statecharts) are verified by projecting model transformation rules into the SAL intermediate language, which provides access to an automated combination of symbolic analysis tools (like model checkers and theorem provers) [4]. One of these is the *Check-VML* framework, which is currently being integrated directly into the *VIATRA* model transformation framework in order to support full-fledged verification.

Testing and validating transformations.

While systematic testing approaches such as unit testing are commonplace in traditional (object-oriented) software development, this is much less the case for graph transformations. Ideally, each graph production specification should come with a suite of tests that verify that the graph production has the desired behaviour. Geiger et al. investigate graph-transformation based testing and debugging in the *Fujaba* environment [8,6].

To a certain extent, *AGG*'s critical pair analysis can also be considered as validation of transformations since it allows the developer to test whether a give set of graph transformations (i.e., a graph grammar) is consistent.

Probably the most advanced graph transformation tool when it comes to verification and validation is *VIATRA*, a transformation-based verification and validation environment for improving the quality of systems designed using the UML by automatically checking consistency, completeness, and dependability requirements [4].

Dealing with incomplete or inconsistent models.

In all the graph transformation tools we studied, the models (i.e., graphs) under consideration have to be well-formed. In other words, none of the approaches allow for inconsistent models. Incompleteness poses less of a problem,

as long as the well-formedness constraints are guaranteed.

Grouping, composing and decomposing transformations.

Composition of graph transformations can be achieved by using *controlled or programmed graph transformation*, i.e., a set of control mechanisms to govern the execution order of rules [23]. Typical control mechanisms are sequencing, branching and looping. They are supported in *Fujaba* by means of so-called *story diagrams*, and in *VIATRA* by means of abstract state machines that drive the transformations. Additionally, in *Fujaba*, transformations are implemented as method bodies, so composition of transformations can be achieved by performing method calls. In *VIATRA*, graph transformations can be composed out of more primitive patterns, though recursive patterns are not supported yet.

Another mechanism that has been proposed to group and compose graph transformation is the structuring mechanism of *graph transformation units* [14,12]. In the graph grammar variant of graph transformation (e.g., *AGG*), one can use *layered graph grammars* as a primitive kind of structuring mechanism. The layers fix the order in which rules are applied. Rules of layer 0 are applied as long as possible, followed by rules of layer 1, and so on.

In *GReAT*, one can form *blocks* from sequenced transformation rules. Blocks encapsulate rules, are hierarchical, and can participate in recursive calls. Rules of a special form, called *tests* are available to implement control flow within a block that is dependent on the input models.

Genericity of transformations.

The only graph transformation tool that supports *higher-order transformations* is *VIATRA* [29]. Higher-order transformations enable a very compact description of certain transformation problems in MDA. A possible disadvantage is a degradation in performance. But this problem is addressed by automatically deriving efficient first-order transformations from generic higher-order ones. To this extent, *meta transformations* are used, i.e., transformations whose source and target models are transformations themselves.

Bidirectionality of transformations.

By definition, a graph transformation rule is unidirectional. This does not mean, however, that it is impossible to support bidirectionality in a graph transformation tool. One obvious way would be to define two graph grammars, one for each direction. Another possibility is to rely on a transaction an rollback mechanism to undo (i.e., reverse) earlier transformations.

Supporting traceability and change propagation.

Most of the considered graph transformation tools have no or poor support for traceability and change propagation and do not provide an incremental update mechanism. Meta transformations can be very helpful here in order to maintain or upgrade existing model transformations.

5 Quality requirements for a graph transformation tool

Usability and usefulness.

That graph transformation technology is *useful* for the purpose of model transformation has been amply illustrated by experiments performed with transformation tools such as *GReAT* and *VIATRA*. Even general-purpose graph transformation tools like *AGG* and *Fujaba* have been shown to support model transformation.

That graph transformations are also *usable* is more difficult to assess, as this depends on several factors, such as the intended target audience. For research purposes, all of the studied tools are already quite usable. To become usable in an industrial setting, most tools still need to mature to make them more performant and user-friendly, but this is just a matter of time.

Verbosity versus conciseness.

Compared to XML-based transformation technology, graph transformation seems to give rise to more concise and better readable code. Whether this code is also easier to produce and maintain is not clear and should be investigated further.

Within the realm of graph transformation tools, a distinction should be made between general purpose tools and dedicated model transformation tools. Due to their dedicated nature, the latter tend to produce more concise code. This goes at the expense of verbosity, since it requires the introduction of extra syntactic constructs that are specifically tuned to model transformation.

Performance and scalability.

Graph transformations are sometimes accused of generating inefficient programs or having inefficient algorithms. However, this poor performance is not an inherent limitation of the technique per se. For example, Varró *et al.* show how to transform higher-order model transformations automatically in efficient first-order transformations [29]. They also suggest to use database technology as an underlying engine of graph transformation in order to increase its efficiency. As another example, Vizhanyo *et al.* have illustrated significant performance gains by optimising traditional graph matching algo-

rithms on the one hand and bypassing the generic transformation engine of *GReAT* by native transformation code on the other hand [30].

When applying graph transformation technology for the purpose of model checking, there are certainly scalability problems. But this has nothing to do with restrictions of the graph transformation approach, but with inherent limitations in model checking.

Extensibility.

The *AGG* tool is extensible in the sense that its internal graph transformation engine, which is implemented in Java, can be extended freely to cover a wide variety of different applications. The tools *Fujaba* and *VIATRA*, offer a powerful plug-in mechanism for extending the tool with new functionality. In the case of *VIATRA* this plug-in mechanism can be used to write importers and exporters for models from other technical spaces than MOF/UML. For example, there are already importers for business process models and XSD. In *GReAT* extensibility is achieved by using a procedural language for implementing the code for *attribute mapping* that gets executed after all the graph operations done in a transformation rule. While this makes the formal analysis of transformation programs (at least) very difficult, it was found to be very practical. This procedural source code is compiled into executable code that is dynamically linked into the execution engine (or statically linked with the compiled transformation code).

Acceptability by user community.

In order to get accepted by an existing user community, a language should not diverge too much from what people are accustomed to. For example, for people trained in procedural programming, a procedural style is probably more readable than the declarative grammar approach of *AGG*. For people accustomed to UML notation, *Fujaba* story diagrams provide a very natural notation to express graph transformations without the user even being aware of it. *GReAT* is based on the meta-programmable Generic Modeling Environment (GME), and is tailored for constructing software that performs transformation on models (typically, but not exclusively constructed in GME). It is equipped with an interpretive engine and debugger (for development) and with a compiler (for generating "production" code from the transformation models). It relies on standard C++ development tools.

Standardization.

Another way to make graph transformation technology accepted is by supporting existing standards such as UML and XML. This is already the case for the considered graph transformation tools. They either support UML or XML directly, or provide some translators (e.g., XMI export facilities) to

bridge between technical spaces. For example, *VIATRA* provides a very flexible import/export mechanism for models from other technical spaces. On the other hand, the metamodelling notation used by *VIATRA* is different from the MOF standard.

Graph transformation tools can also be applied using MDA standards like UML, MOF and XMI. Type graphs can be defined as class diagrams in UML editors with proper XMI export facilities. The resulting model can be transformed into a MOF metamodel. Existing MDA frameworks can be used to monitor the OCL well-formedness rules of this metamodel on models residing in a MOF repository. Model transformations can be developed as graph transformations expressed in UML statechart and class diagram editors that export the transformation models to XMI. The transformation models can be transformed into executable MOF transformation code that can transform any model that is an instance of the original type graph [21,5].

For graph transformation languages in particular, two standards are available. GXL is an exchange format for graphs, while GTXL is an exchange format for graph transformations. Both standards are supported by *AGG*.

6 Discussion

In this paper, we used a *taxonomy* of model transformation to evaluate the technique of graph transformation as a way to support the activity of model transformation. The taxonomy also assisted in identifying and evaluating the appropriateness of a representative subset of four tools based on graph transformation technology. A summary of our initial analysis is presented in Table 6. Not all criteria of the model transformation taxonomy appear in this table, since we only displayed those criteria where a difference could be discerned. In the future, we intend to carry out a more detailed analysis, and we will consider other tools based on graph transformation as well.

Nevertheless, based on the analysis we performed, we can already conclude that graph transformation is a promising approach to deal with model transformation. First of all, for many types of models, that are intrinsically graph-based in nature, graph transformations offer a natural and direct way to specify model transformations. Secondly, the graph transformation approach is formally founded: one can resort to many theorems to prove certain properties of a transformation system. Finally, graph transformation technology offers mechanisms to reuse transformations, and to compose smaller transformations into more complex ones.

A disadvantage is that the various graph transformation approaches are not al1ways compatible. With respect to standardization, there is a tendency to combine graph transformation technology with XML and UML notation. This tendency favours acceptability by the user community because of their familiarity with these languages. Compared to *AGG* and *Fujaba*, the *VIATRA* tool is more tuned to the activity of model transformation since it was specifically

built for this purpose. *VIATRA* seems to be one of the most advanced tools, since it offers very advanced features such as higher-order transformations, meta transformations, and verification and validation of transformations. On the other hand, *GReAT* and *Fujaba* provide a full environment for developing transformation programs: a (visual) modeling tool, interpreter with debugger, and a code generator.

References

[1] Baresi, L., R. Heckel, S. Thöne and D. Varró, *Style-based refinement of dynamic software architectures*, in: *Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA4)* (2004), pp. 155–164.

[2] Burmester, S., H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals and A. Zuendorf, *Tool integration at the meta-model level: The fujaba approach*, Int'l Journal on Software Tools for Technology Transfer **6** (2004), pp. 303–318.

[3] Corradini, A., U. Montanari and F. Rossi, *Graph processes*, Fundamenta Informaticae **26** (1996), pp. 241–265.

[4] Csertán, G., G. Huszerl, I. Majzik, Z. Pap, A. Pataricza and D. Varró, *VIATRA - visual automated transformations for formal verification and validation of UML models*, in: *Proc. 17th Int'l Conf. Automated Software Engineering* (2002), pp. 267–270.

[5] Formal Techniques in Software Engineering, *Model driven, Template based, Model Transformer (MoTMoT)*, http://sourceforge.net/projects/motmot/ (2004).

[6] Geiger, L. and A. Zündorf, *Graph based debugging with Fujaba*, in: *Proc. Int'l Workshop on Graph Based Tools*, Electronic Notes in Theoretical Computer Science **72** (2002).

[7] Geiger, L. and A. Zündorf, *Statechart modeling with fujaba*, in: *Proc. Int'l Workshop Graph-Based Tools (GraBaTs)*, Electronic Notes in Theoretical Computer Science (2004).

[8] Geiger, L. and A. Zündorf, *Transforming graph based scenarios into graph transformation based JUnit tests*, in: *Proc. AGTIVE*, Lecture Notes in Computer Science **3062**, Springer, 2004 pp. 61–74.

[9] Heckel, R., J. Küster and G. Taentzer, *Confluence of typed attributed graph transformation systems*, in: *Proc. 1st Int'l Conf. Graph Transformation*, Lecture Notes in Computer Science **2505** (2002), pp. 161–176.

[10] Kalnins, A., J. Barzdins and E. Celms, *Model transformation language MOLA*, in: *Proc. Model-Driven Architecture: Foundations and Applications*, 2004, pp. 14–28.

[11] Karsai, G., A. Agrawal and F. Shi, *On the use of graph transformations for the formal specification of model interpreters*, Journal of Universal Computer Science **9** (2003), pp. 1296–1321.

[12] Klempien-Hinrichs, R., H.-J. Kreowski and S. Kuske, *Typed graph transformation units*, in: *Proc. 2nd Int'l Conf. Graph Transformation*, Lecture Notes in Computer Science **3526** (2004), pp. 112–127.

[13] Königs, A. and A. Schürr, *Mdi - a rule-based multi-document and tool integration approach*, Software and Systems Modelling (2005).

[14] Kreowski, H.-J. and S. Kuske, *Graph transformation units and modules*, Handbook of Graph Grammars and Computing by Graph Transformation **2** (1999), pp. 607–638.

[15] Mens, T., *On the use of graph transformations for model refactoring*, in: *Proc. Int'l Summer School on Generative and Transformational Techniques in Software Engineering*, 2005.

[16] Mens, T. and P. V. Gorp, *A taxonomy of model transformation*, in: *Submitted to International Workshop on Graph and Model Transformation (GraMoT)*, 2005.

[17] Mens, T., N. Van Eetvelde, S. Demeyer and D. Janssens, *Formalizing refactorings with graph transformations*, Int'l Journal on Software Tools for Technology Transfer **17** (2005), pp. 247–276.

[18] M.Große-Rhode, F. P. Presicce and M. Simeoni, *Formal software specification with refinements and modules of typed graph transformation systems*, Journal of Computer and System Sciences **64** (2002), pp. 171–218.

[19] Rensink, A., *The GROOVE simulator a tool for state space generation*, in: *Proc. AGTIVE 2003*, Lecture Notes in Computer Science **3062** (2004), pp. 479–485.

[20] Rensink, A., A. Schmidt and D. Varró, *Model checking graph transformations: A comparison of two approaches*, in: *Proc. 2nd Int'l Conf. Graph Transformation*, Lecture Notes in Computer Science **3526** (2004), pp. 226–241.

[21] Schippers, H., P. Van Gorp and D. Janssens, *Leveraging UML profiles to generate plugins from visual model transformations*, in: *Proc. Int'l Workshop Software Evolution through Transformations (SETra)*, Electronic Notes in Theoretical Computer Science **127** (2005), pp. 5–16.

[22] Schürr, A., *Specification of graph translators with triple graph grammars*, in: *Proc. WG'P4 Workshop on Graph-Theoretic Concepts in Computer Science*, 1994, pp. 151–163.

[23] Schürr, A., *Logic based programmed structure rewriting systems*, Fundamenta Informaticae **26** (1996), pp. 363–385.

[24] Schürr, A., A. Winter and A. Zündorf, "Handbook of Graph Grammars and Graph Transformation," World Scientific, 1999 pp. 487–550.

[25] Sprinkle, J., A. Agrawal, T. Levendovszky, F. Shi and G. Karsai, *Domain model translation using graph transformations*, in: *Proc. Int'l Conf. Engineering of Computer-Based Systems* (2003), pp. 159–168.

[26] Taentzer, G., *AGG: A graph transformation environment for modeling and validation of software*, in: *Proc. AGTIVE 2003*, Lecture Notes in Computer Science **3062** (2004), pp. 446–453.

[27] Van Eetvelde, N. and D. Janssens, *Extending graph rewriting for refactoring*, in: *Proc. 2nd Int'l Conf. Graph Transformation*, Lecture Notes in Computer Science **3526** (2004), pp. 399–415.

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

[29] Varró, D. and A. Pataricza, *Generic and meta-transformations for model transformation engineering*, in: A. M. Thomas Baar, Alfred Strohmeier, editor, *UML 2004 - The Unified Modeling Language*, Lecture Notes in Computer Science **3273** (2004), pp. 290–304.

[30] Vizhanyo, A., A. Agrawal and F. Shi, *Towards generation of efficient transformations*, in: *Proc. Generative Programming and Component Engineering*, Lecture Notes in Computer Science **3286** (2004), pp. 298–316.

Table 1

criterion	*AGG*	*Fujaba*	*VIATRA*	*GReAT*
number of source and target models	one-to-one	many-to-many	many-to-many	many-to-many
kind of transformation	endogenous	endogenous	endogenous + exogenous	endogenous + exogenous
technical space	XML, GXL, GTXL	MDA, UML, Java	MDA, XSD, business process models	MDA, UML
level of automation	graph grammars	story-driven modelling	transformations driven by abstract state machines	explicitly sequenced transformation steps with context parameters
complexity	layered grammars	controlled graph transformations	higher-order and meta transformations	controlled graph transformations
customisability / reusability	parameterised transformations	parameterised transformations / inheritance of transformations	reuse of predefined patterns	reuse of transformation rules and blocks
verification / validation	termination checking, consistency checking, critical pair analysis	graph-transformation-based JUnit tests	full-fledged verification/validation by *Check-VML* [20]	well-formedness constraints on transformation results
composition	layered graph grammar	method calls in story driven modelling	non-recursive composition of patterns into rules	hierarchical blocks of sequences, recursion
usability	low (not very performant or scalable)	high	high	high
extensibility	by extending AGG's internal engine	plug-in mechanism	powerful plug-in mechanism for writing importers and exporters	procedural code
acceptability	for research purposes	for software development	for model transformation	for model transformation
standardisation	GXL, GTXL	UML, Java, XMI, MOF	UML, XMI, MOF	UML, XMI