A Taxonomy of Model Transformation and its Application to Graph Transformation

Tom Mens[1] and Pieter Van Gorp[2]

[1] Software Engineering Lab, Université de Mons-Hainaut
Av. du champ de Mars 6, 7000 Mons, Belgium
tom.mens@umh.ac.be
[2] Formal Techniques in Software Engineering, Universiteit Antwerpen
Middelheimlaan 1, 2020 Antwerpen, Belgium
pieter.vangorp@ua.ac.be

Abstract. This article proposes a taxonomy of model transformation, based on the discussions of a working group on model transformation of the Dagstuhl seminar on Language Engineering for Model-Driven Software Development. This taxonomy can be used to help developers in deciding which model transformation approach is best suited to deal with a particular problem. We validate the taxonomy by applying it to graph transformation, a promising technology to deal with model transformation.

1 Introduction

In this paper we propose a *taxonomy* of model transformation. Such a taxonomy is particularly useful to help a software developer choosing a particular model transformation approach that is best suited for his needs.

The taxonomy is based on the discussions of a working group of the Dagstuhl seminar on Language Engineering for Model-Driven Software Development. The working group was composed of: J. Bézivin, A. Cherchago, K. Czarnecki, T. Gardner, T. Girba, M. Gogolla, J.-M. Jézequel, F. Jouault, A. Königs, J. Küster, T. Mens, L. Tratt, P. Van Gorp, D. Varro, H. Wehrheim, and M. Wermelinger. The working group addressed a variety of important issues with *model transformation*, undoubtedly the most profound aspect of model-driven software development [1, 2]. The group started with a discussion on the essential characteristics of model transformations, as well as their supporting languages and tools. The group also discussed the commonalities and variabilities between existing model transformation approaches.

As an initial validation of this taxonomy, we apply it to the technique of graph transformation to find out what are the merits and drawbacks of this technique with respect to model transformation.

2 Definitions and Examples

Before classifying model transformation techniques, one should understand some model driven engineering definitions. We will clarify the definition of a model and a model transformation by means of two running examples.

Several sources acknowledge that *a model is a simplified representation of a part of the world named the system* [3, 4]. A model is useful if it helps to gain a better understanding of the system. In an engineering context, a model is useful if it helps deciding the appropriate actions that need to be taken to reach and maintain the system's goal.

The goal of software is to automate some tasks in the real world. Models of software requirements, structure and behavior at different levels of abstraction help all stakeholders deciding how this goal should be accomplished and maintained. According to this definition, source code is a model too since it is a simplified representation of the lower-level machine structures and operations that are required to automate the tasks in the real world. Moreover, *correct* source code is a very useful model since it tells the machine what actions need to be taken to maintain the system's goal. Design representations of the source code (e.g., UML diagrmas) are useful models if they make the source code more understandable.

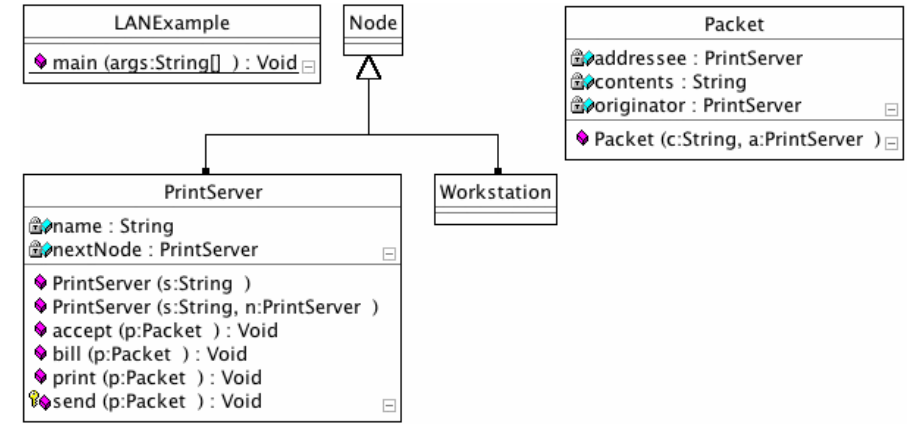
When building modeling tools, one needs to model the structure and well-formedness rules of the language in which the models are expressed. Such models are called metamodels [5]. Having a precise metamodel is a prerequisite for performing automated model transformations.

Consider the UML class diagrams in Fig. 1. The diagrams visualise the static structure of a Local Area Network (LAN) application [6] before and after executing a model transformation. The method `bill` is pulled up from the subclass `PrintServer` (see Fig. 1 (a)) to the superclass `Node` (see Fig. 1 (b)).

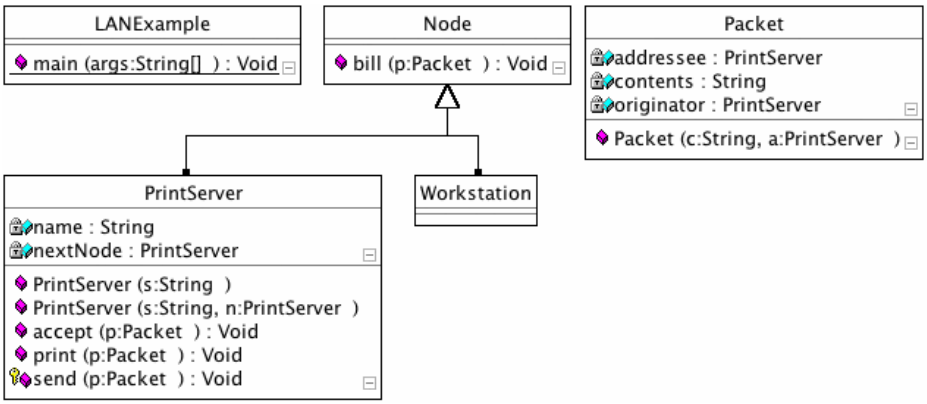
As another example, consider the problem of translating hierarchical statecharts into flat ones [7]. A part of the translation process consists of redirecting transitions starting from composite states to make these transitions start from the contained states. If a contained state already has an outgoing transition with the same label as the outer transition, that contained state will not get an extra outgoing transition. This prevents non-determinism. Fig. 2 visualises one step of the flattening algorithm. To complete the flattening, the transition to `Delivered` needs to be flattened as well and the `Active` state should ultimately be removed.

3 Model transformation taxonomy

In order to decide which model transformation approach is most appropriate for addressing a particular problem, a number of crucial questions need to be answered. Each of the following subsections investigate a particular question, and suggest a number of objective criteria to be taken into consideration to provide a concrete answer to the question. Based on the answers, the developer can then select the model transformation approach that is most suited for his needs.



(a)



(b)

Fig. 1. Class diagram before and after executing the *pull up method* transformation.

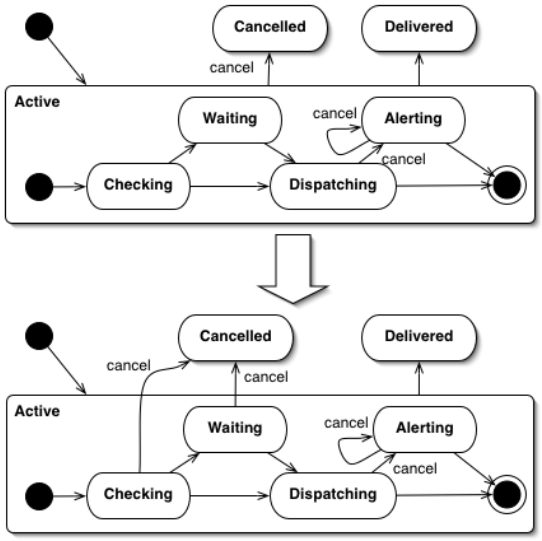


Fig. 2. *A hierarchical statechart being transformed into a more flat one.* The example of a delivery process is inspired by a popular UML book [8]. We have adapted the example such that an order can only be cancelled if it has not been dispatched yet.

3.1 What needs to be transformed into what?

The first important question concerns the source and target artifacts of the model transformation. If these artifacts are programs (i.e., source code, bytecode, or machine code), one uses the term *program transformation*. If the software artifacts are models, we use the term *model transformation*. According to the definitions presented in Section 2, the latter term encompasses the former one since a model can range from abstract analysis representations of the system, over more concrete design models, to very concrete models of source code. Hence, model transformations also include transformations from a more abstract to a more concrete model (e.g., from design to code) and vice versa (e.g., in a reverse engineering context). Model transformations are obviously needed in common tools such as code generators and parsers.

Given that all program transformations can be performed as model transformations, one can classify the source and target models of a transformation in terms of their structure. More specifically, some systems can be represented as a strict tree whereas others require a graph representation. Note that every graph can be encoded as a tree with references from certain nodes to nodes different from their child nodes. However, navigating a graph encoded as a tree requires (potentially tedious) join operations. Encoding a graph in a relational datastructure leads to even more join operations since the relations between tree nodes and their children need to be represented by means of references as well.

Therefore, one should choose the technology that matches the system as closely as possible without sacrificing too much runtime performance.

Number of source and target models. A first distinguishing characteristic of a model transformation is its number of source and target models. Kleppe et al. [9] provided the following definition of model transformation. *A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition. A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.* We suggest that this should be generalised, in that a model transformation should also be applicable to **multiple source models** and/or **multiple target models**. An example of the former is model merging, where we want to combine or merge multiple source models that have been developed in parallel into one resulting target model. An example of the latter is a transformation that takes a platform-independent model (PIM), and transforms it into a number of platform-specific models (PSM). Both examples are schematically represented in Fig. 3.

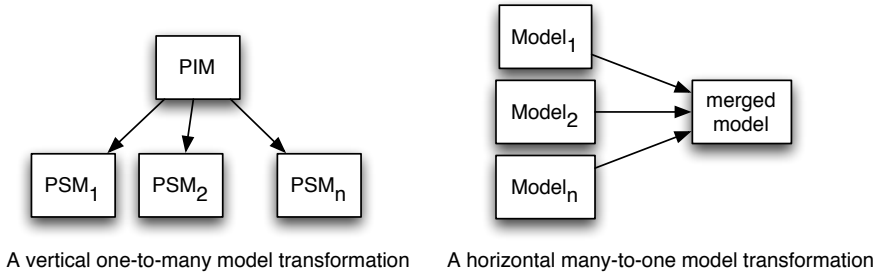


Fig. 3. Examples of model transformations

Technological space. The source and target models of a model transformation may belong to one and the same or to different technological spaces [10]. A technological space is determined by the meta-metamodel that is used (**M3**-level). For example, the world-wide web consortium (W3C) promotes the *XML technological space*, which uses *XML Schema* as meta-metamodel. This space includes support for languages such as HTML, XML, XMI, XSLT, and XQuery. As another example, the Object Management Group (OMG) promotes the *MDA technological space*, which uses the *MOF* as meta-metamodel, and supports languages such as UML.

If we want to have transformations between models in different technological spaces, transformation tools need to provide exporters and importers to bridge the technological spaces while the actual transformation is executed in the technological space of either the source or target model.

For example, when translating XML documents into UML diagrams one can choose to execute the actual transformation in either the XML or the MDA technological space. To perform the transformation in the XML technological space, one would use an XSLT or XQuery program translating the general XML document into an XML document conforming to the syntax of the XMI standard (XML metadata interchange) and conforming to the semantics of the MOF-XMI document for the UML standard. An XMI parser can then be used to import the resulting XMI document in a UML CASE tool, residing in the MDA technological space.

Performing the transformation in the MDA technological space would require a MOF metamodel for XML. After parsing the XML document into instances of this metamodel, the actual transformation could be performed as a MOF transformation. The QVT request for proposals [11] aims to standardise a programming language for implementing this kind of model transformations.

Endogenous versus exogenous transformations. In order to transform models, these models need to be expressed in some modeling language (e.g., UML for design models, and programming languages for source code models). The syntax and semantics of the modeling language itself is expressed by a *metamodel*. For example, the syntax of the UML metamodel is expressed using class diagrams, whereas its semantics is described by a mixture of well-formedness rules (expressed as OCL constraints) and natural language [12].

Based on the language in which the source and target models of a transformation are expressed, a distinction can be made between *endogenous* and *exogenous* transformations. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages.[3]

This distinction is essentially the same as the one that was proposed in the "Taxonomy of Program Transformation" [13], but ported to a model transformation setting. In that taxonomy, the term *rephrasing* was used for an endogenous transformation, whereas the term *translation* was used for an exogenous transformation.
Typical examples of exogenous transformation (i.e., translation) are:

- *Synthesis* of a higher-level, more abstract, specification (e.g., an analysis or design model) into a lower-level, more concrete, one (e.g, a model of a Java program). A typical example of synthesis is *code generation*, where the source code is translated into bytecode (that runs on a virtual machine) or executable code, or where the design models are translated into source code.

[3] If we have to deal with transformations with multiple source models and/or multiple target models, there can even be more than 2 different languages involved.

- *Reverse engineering* is the inverse of synthesis and extracts a higher-level specification from a lower-level one.
- *Migration* from a program written in one language to another, but keeping the same level of abstraction.

Typical examples of endogenous transformation (i.e., rephrasing) are:

- *Optimisation*, a transformation aimed to improve certain operational qualities (e.g., performance), while preserving the semantics of the software.
- *Refactoring*, a change to the internal structure of software to improve certain software quality characteristics (such as understandability, modifiability, reusability, modularity, adaptability) without changing its observable behaviour [14]. The *pull up method* transformation of Figure 1 is an example of such a refactoring.
- *Simplification* and *normalisation*, used to decrease the syntactic complexity, e.g., by translating syntactic sugar into more primitive language constructs. The statechart flattening transformation of Figure 2 is an example of such a simplification.

One can further classify endogenous model transformations in terms of the number of models involved. If this number is only one, the source and target model are the same and all changes are made *in-place*. Other endogenous transformations create model elements in one model based on properties of another model (regardless of the fact that both models conform to the same metamodel). Such transformations are called *out-place*. Note that exogenous transformations are always out-place. We do not incorporate this distinction in the proposed taxonomy since for most applications it doesn't matter whether a transformation is implemented in- or out-place. Still, the terms have shown to be useful in technical discussions on model transformation.

Horizontal versus vertical transformations. A *horizontal transformation* is a transformation where the source and target models reside at the same abstraction level. Typical examples are *refactoring* (an endogenous transformation) and *migration* (an exogenous transformation). A *vertical transformation* is a transformation where the source and target models reside at different abstraction levels. A typical example is *refinement*, where a specification is gradually refined into a full-fledged implementation, by means of successive refinement steps that add more concrete details [15, 16].

Table 1 illustrates that the dimensions *horizontal versus vertical* and *endogenous versus exogenous* are truly orthogonal, by giving a concrete example of all possible combinations. As a clarification for the *Formal refinement* mentioned in the table, a specification in first-order predicate logic or set theory can be gradually refined such that the end result uses exactly the same language as the original specification (e.g., by adding more axioms).

Table 1. Orthogonal dimensions of model transformations

	horizontal	vertical
endogenous	*Refactoring*	*Formal refinement*
exogenous	*Language migration*	*Code generation*

3.2 Important characteristics of a model transformation

Level of automation. A distinction can and should be made between model transformations that can be automated and transformations that need to be performed manually (or at least need a certain amount of manual intervention).

An example of the latter is a transformation from a requirements document to an analysis model. For such a transformation, manual intervention is needed to address and resolve ambiguity, incompleteness and inconsistency in the requirements that are (partially) expressed in natural language.

Complexity of the transformation. Some transformations, such as model refactorings, can be considered as small, while others are considerably more heavy-duty. Examples of the latter are parsers, compilers and code generators. The difference in complexity between small transformations and heavy-duty transformations is so big that they require an entirely different set of techniques and tools.

Preservation. Although there is a wide range of different types of transformations that are useful during model-driven development, each transformation preserves certain aspects of the source model in the transformed target model. The properties that are preserved can differ significantly depending on the type of transformation. For example, with *refactorings* or *restructurings*, the (external) behaviour needs to be preserved, while the structure is modified. With *refinements*, the program correctness needs to be preserved [17].

3.3 Success criteria for a transformation language or tool

In the previous discussion, we restricted ourselves to characteristics of the model transformation or of the models being transformed. Equally important, or perhaps even more important, are the characteristics of a *transformation language* or *transformation tool*. Below we enumerate a number of important *functional requirements* that contribute to the success of such a language or tool.

Ability to create/read/update/delete transformations (CRUD). While this is a trivial requirement for a transformation language, it is not that obvious for a transformation tool. For example, if we consider a typical program refactoring tool, it comes with a predefined set of refactoring transformations that can

be applied, but there is often no way to define new refactoring transformations, or to fine-tune existing transformations to specific needs of the user. As such, having the possibility to create new transformations or update existing ones is an important criterion.

Ability to suggest when to apply transformations. For certain application scenarios, dedicated tools can be built that suggest to the user which model transformations might be appropriate in a given context. For example, a refactoring tool might not only apply refactoring transformations, but also suggest in which context a particular refactoring should be applied [18, 19].

Ability to customise or reuse transformations. For example, if we adopt an object-oriented transformation language, we may be able to use the inheritance mechanism to reuse the specifications of model transformations. Other customisation or reuse mechanisms include parameterisation and templates.

Ability to verify and guarantee correctness of the transformations. The simplest notion of correctness is *syntactic correctness*: given a well-formed source model, can we guarantee that the target model produced by the transformation is well-formed? Another notion is *syntactic completeness*: for each element in the source model, there should be a corresponding element in the target model that can be created by a model transformation.
A significantly more complex notion is *semantic correctness*: does the produced target model have the expected semantic properties? This is for example a crucial requirement for refactoring transformations, were we want to be able to ensure that these transformations preserve certain behavioural properties. Other important semantic properties are *termination* and *confluence*: given a set of transformations, they should always lead to a result (i.e., they should terminate) and this result should be unique (confluence).

Ability to test and validate transformations. Since transformations can be considered as a special kind of programs (e.g., the XSLT transformation language is a Turing-equivalent programming language), we need to apply systematic testing and validation techniques to transformations to ensure that they have the desired behaviour.

Ability to deal with incomplete or inconsistent models. It is important to be able to transform models early in the software development life-cycle, when requirements may not yet be fully understood or are described in natural language. This often gives rise to ambiguous, incomplete or inconsistent models, which implies that we need to have mechanisms for inconsistency management. These mechanisms may be used to detect, and possibly resolve, inconsistencies in the transformations themselves, or in the models being transformed.

Ability to group, compose and decompose transformations. The ability to combine existing transformations into new composite ones is useful to increase the readability, modularity and maintainability of a transformation language. Decomposition of a complex transformation into smaller steps may also require a control mechanism to specify how these smaller transformations need to be combined. This control mechanism may be implicit or explicit.

Ability to specify generic and higher-order transformations. Ideally, transformations should be first class entities in a transformation language. If we can represent transformations as models too, we can apply transformations to these models, thus achieving a notion of higher-order transformations. A concrete example of this would be to refactor a given set of transformations (e.g., a family of code generators), to reduce the amount of code duplication in these transformations. In order to achieve this, we need to transform the transformations themselves.

Ability to specify bidirectional transformations. Languages or tools that have the property of bidirectionality require fewer transformation rules, since each transformation can be used in two different directions: to transform the source model(s) into target model(s), and the inverse transformation to transform the target model(s) into source model(s).

Support for traceability and change propagation. To support *traceability*, the transformation language or tool needs to provide mechanisms to maintain an explicit link between the source and target models of a model transformation. To support *change propagation*, the transformation language or tool may have a consistency checking mechanism and an incremental update mechanism [20]. Note that some transformation approaches require to translate the source model first into some standardised format (e.g., XML), then apply the transformation, and then do another translation to obtain the target model. A clear disadvantage of such an approach is that it is difficult to synchronise source and target models when changes are made to them.

3.4 Quality requirements for a transformation language or tool

Besides all the functional requirements enumerated above, a transformation language or tool should also satisfy a number of *non-functional* or *quality requirements*.

Usability and usefulness. The language or tool should be *useful*, which means that it has to serve a practical purpose. On the other hand, it has to be *usable* too, which means that it should be intuitive and efficient to use. Obviously, this issue is directly related to developer training and experience. Instead of using a full-fledged transformation language, developers may prefer a more direct model

manipulation approach, where the internal model is directly accessed by means of an API. The advantage is that developers can keep on using their preferred language and require virtually no extra training. The disadvantage is that the API may restrict the kinds of transformations that can be performed [2].

Verbosity versus conciseness. *Conciseness* means that the transformation language should have as few syntactic constructs as possible. From a practical point of view, however, this often requires more work to specify complex transformations. Hence, the language should be more *verbose* by introducing extra syntactic sugar for frequently used syntactic constructs. It is always a difficult task to find the right balance between these two conflicting goals. Referring to the previous example of direct model manipulation via an API, the developers will typically use a general purpose programming language to specify the transformations. This leads to considerably more verbose code than with dedicated model transformation languages [2].

Performance and scalability. The language or tool should be able to cope with large and complex transformations or transformations of large and complex software models without sacrificing performance.

Extensibility. The flexibility of a tool depends on how easy it is to extend it with new functionality. For example, the tool could offer a built-in plug-in mechanism.

Mathematical properties. If the transformation language or tool has a mathematical underpinning, it may be possible, under certain circumstances, to prove theoretical properties of the transformation such as termination, soundness, completeness, (syntactic and semantic) correctness, etc.

Acceptability by user community. The best transformation language from a theoretical point of view may not necessarily be the best from a pragmatic point of view. For example, it the target community is an object-oriented audience, a transformation language based on a logic or functional paradigm may not be acceptable.

Standardisation. The transformation tool should be compliant to all relevant standards (such as XML, MOF, UML). For example, the tool may need to support XMI for importing or exporting the source or target models of a transformation.

3.5 Which mechanisms can be used for model transformation?

Mechanisms should be interpreted here in a broad sense. They include techniques, languages, methods, and so on. To specify and apply a transformation,

ideas from any of the major programming paradigms can be used. One can decide to use a procedural, an object-oriented, a functional or a logic approach, or a hybrid approach combining any of the former ones.

The major distinction between transformation mechanisms is whether they rely on a *declarative* or an *operational* (or *imperative*) approach. Declarative approaches focus on the *what* aspect, i.e., they focus on what needs to be transformed into what by defining a relation between the source and target models. Operational approaches focus on the *how* aspect, i.e., they focus on how the transformation itself needs to be performed by specifying the steps that are required to derive the target models from the source models.

Declarative approaches (e.g., [21]) are attractive because particular services such as source model traversal, traceability management and automatic bidirectionality can be offered by an underlying reasoning engine. There are several aspects that can be made implicit in a transformation language: (1) navigation of a source model, (2) creation of target model and (3) order of rule execution. As such, declarative transformations tend to be easier to write and understand by software engineers.

Operational (or *constructive*) approaches (e.g., [22]) may be required to implement transformations for which declarative approaches fail to guarantee their services. Especially when the application order of a set of transformations needs to be controlled explicitly, an imperative approach is more appropriate thanks to its built-in notions of sequence, selection and iteration. Such explicit control may be required to implement transformations that reconcile source and target models after they were both heavily manipulated outside that transformation tool.

One particular flavour of a declarative approach is *functional programming*. Such an approach towards model transformation is appealing, since any transformation can be regarded as a function that transforms some input (the source model) into some output (the target model). In most functional languages, functions are first class, implying that transformations can be manipulated as models too. An important disadvantage of the functional approach is that it becomes awkward to maintain state during transformation.

Another flavour of a declarative approach is *logic programming*. A logic language (e.g., Prolog or Mercury) has many features that are of direct interest for model transformation: backtracking, constraint propagation (in the case of constraint logic programming languages), and unification. Unification may either be *partial* (which is easier to use and understand) or *full* (which is more powerful). Additionally, logic languages always offer a query mechanism, which means that no separate query language needs to be provided.

4 Case study: Graph transformation

In this section we will apply the model transformation taxonomy to the technique of graph transformation [23]. In fact, graph transformation is more a programming paradigm than a technique, since it consists of a large set of different

theories, languages and tools. Some of these, such as Fujaba [24] and AGG [25], are general-purpose. Tools such as GReAT [22] and MOLA [26] are specifically tuned for the activity of model transformation. Tools such as VIATRA [27] and GROOVE [28] are used for the activity of model checking.

4.1 What needs to be transformed into what?

In traditional compiler technology, programs are typically represented as an abstract syntax tree. In program analysis research, on the other hand, programs are frequently represented as graphs, e.g., to specify the control-flow or data-flow dependence relationships of programs. Compared to a graph structure, a tree structure is less intuitive, since it deals with cross-references (such as variable assignments and method invocations) in a suboptimal way. On the other hand, tree structures outperform graph structures when it comes to performance and scalability.

Number of source and target models. Graph transformation theory allows us to express *one-to-one model transformations* directly as a *graph production* (a.k.a. graph transformation rule) with a left-hand side (LHS) representing the source model, and a right-hand side (RHS) representing the target model. In other words, a graph transformation $T : G \to H$ is defined as the application of a graph production $P : L \to R$ to an initial graph G by finding an occurrence (or more formally, a *match* $m : L \to G$) of the production's LHS in the initial graph G, and applying the production P in the context of G, which leads to a result graph H. This process is schematically depicted in Fig. 4.

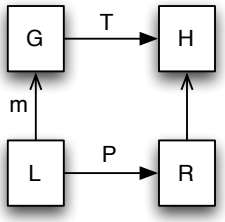


Fig. 4. Schematic representation of a graph transformation

Whether a graph-transformation tool can be used to express *one-to-many*, *many-to-one* or *many-to-many* transformations depends on whether it is possible to manipulate models having different metamodels. In graph-transformation theory, the metamodel is expressed as a so-called *type graph*. The AGG tool provides support for a single type graph, so it can only support *one-to-one* model

transformations. Other tools, such as GREAT and VIATRA, or more flexible, since they can deal with multiple metamodels at once.

Technological space. AGG is a general-purpose graph transformation tool that uses GXL, a standard graph exchange format [29] as meta-metamodel. However, since GXL is based on XML, it is relatively easy to provide a mapping to the *XML technological space*. VIATRA [27], a model transformation tool that is based on graph-transformation technology, belongs to the *MDA technological space*. The VIATRA tool uses an XMI input/output format that conforms to the MOF model. The Fujaba tool suite [24] is a CASE tool for 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 technological space*. Agrawal *et al.* [22] introduced *GReAT*, a graph-transformation tool and associated language to specify model transformations using UML notation. As such, it belongs to the *MDA technological space*.

To summarise, most graph transformation tools allow us to deal with models in the XML or MDA technological space directly, or using some translators.

Endogenous versus exogenous transformations. In graph transformation, the structure of a metamodel is described by means of a *type graph*. In an endogenous graph transformation, the source and target graphs are instances of the same type graph. Exogenous transformations, where the source and target are expressed in a different domain (i.e., have a different metamodel) can also be expressed using graph transformations, provided that one specifies a different type graph for source and target model. This is precisely what has been done in GReAT [22], where a specific graph transformation language is used for this purpose.

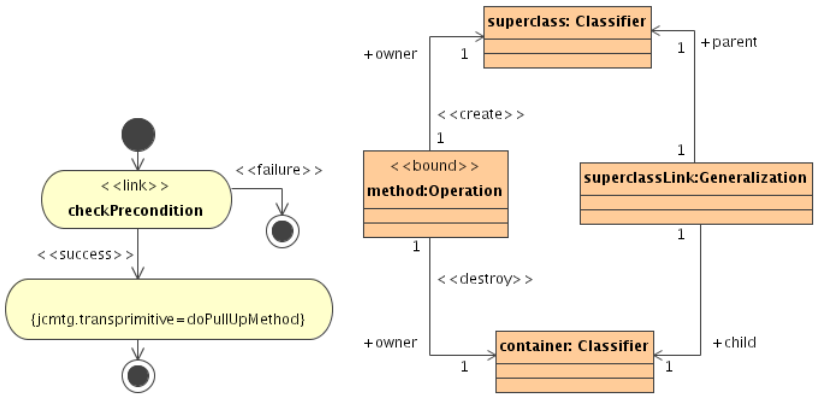
Horizontal versus vertical transformations. Graph transformation technology can be, and has been, used for all activities mentioned in Table 1. As an example of *vertical exogenous* model transformation, Karsai *et al.* show how to transform a platform-independent model (PIM) into a more platform-specific model (PSM) using GReAT [30]. As another example, Baresi *et al.* illustrate the use of graph transformation for refinement of software architectures [31]. As an example of *horizontal endogenous* model transformation, graph transformations have been used to specify refactorings in [32, 33]. As an example of *horizontal exogenous* transformation, graph transformations have been used for migration from one domain-specific language to another using GReAT [22]. Finally, Große-Rhode *et al.* [34] illustrates *vertical endogenous* transformations by specifying formal refinements as typed graph transformations.

As an example of a horizontal endogenous transformation, Fig. 5 shows how the *pull up method* refactoring that was introduced in Figure 1 can be expressed as a graph transformation in the UML profile for Fujaba's Story Driven Modeling (SDM) language [35, 36]. In this profile, graph transformations are modeled

as methods of special classes. This structure is modeled in a class diagram with stereotypes on the classes representing transformations and the operations representing rules. Because of their correspondence to operations, the visual transformation specifications can access their formal parameters. Moreover, Java programs can call Java methods generated from the SDM specifications without knowing about the graph transformation concepts.

Fig. 5 **(a)** specifies the control flow of the actions that need to be executed to perform the transformation. First the preconditions have to be checked. In case of success, the *transformation primitive* `doPullUpMethod` is applied to actually transform the source graph. The specification of `doPullUpMethod` is shown in Fig. 5 **(b)**. Links and objects without a stereotype are to be matched in the host graph. Those with a ≪create≫ stereotype have to be created while those with a ≪destroy≫ stereotype need to be destroyed.

In the example of the *pull up method* transformation for UML models, the ≪bound≫ node `method` represents the argument of the transformation. The type of node *method* is the UML metaclass *Operation*. The edges `owner`, `child` and `parent` between `method`, `container` and `superclass` specify a pattern that checks if the containing class has a superclass. If so, the link to this class is destroyed and a link to the superclass is created.



(a) (b)

Fig. 5. Parts of the specification of the *pull up method* refactoring as a graph transformation in Fujaba's Story Driven Modeling language.

Note that the `Generalisation` node is counter intuitive since it is not directly mapped to a concept in the UML class diagram syntax. Instead, one would like to reason in terms of the inheritance relationship by means of UML's arrow

with a hollow end between `container` and `superclass`. The following example illustrates that nothing prevents tool builders to automate the mapping from concrete UML syntax to a typed graph instance.

Fig. 6 represents a graph transformation on UML statecharts within the UML statecharts syntax. The example rule is part of an endogenous vertical transformation for flattening a nested statechart. It takes those `source` states nested in a `composite` state that do no have an outgoing transition whose label `aLabel` is equal to the label of a transition originating from the composite state. For those states, the transformation will create a transition from the `source` state to the `outer target` state and remove the transition originating from the `composite` state. Nodes that are denoted by a double rounded rectangle (e.g., `source`) can match to multiple nodes in the host graph. Nodes denoted by a crossed-out rounded rectangle (e.g., `inner target`) should not occur in the host graph. In graph transformation terminology this is called a *negative application condition*.

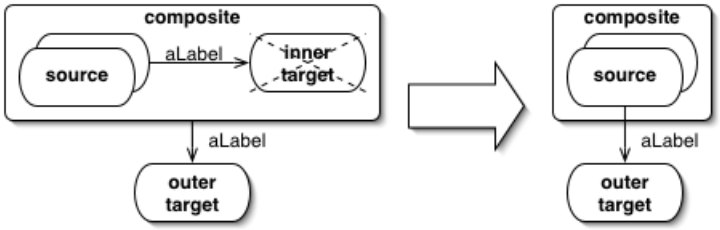


Fig. 6. Part of the specification of statechart flattening as a graph transformation using concrete syntax to improve understandability.

Before the transformation can be executed by a general purpose graph transformation engine, such rules need to be translated into graph transformations on instances of a type graph (i.e., a metamodel structure). In the example rule, the edge from `source` to `inner target` needs to be translated to a node which is instance of `Transition` and has `source` (instance of `State`) and `inner target` (also instance of `State`) as values for its `src` and `trg` attributes. Similarly, the nesting of the `source` and `inner target` states into `composite` needs to be translated into two instances of the `State` metaclass which are connected to an instance of the `CompositeState` metaclass. Fig. 7 shows what the resulting rule would look like.

4.2 Important characteristics of a model transformation

Level of automation. In most graph transformation tools, individual graph productions can be selected by the user and applied to an initial graph, possibly after selecting a match of the LHS in the initial graph (in the case where there

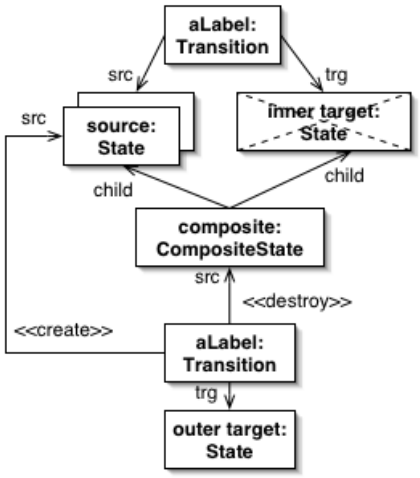


Fig. 7. Part of the specification of statechart flattening as a graph transformation translated to abstract syntax.

are multiple matches of the graph production). Alternatively, one may resort to a fully automatic graph grammar approach. 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. Such a grammar can be seen as a one-to-many transformation.

A graph grammar may also be executed non-deterministically. In this case, for each iteration an applicable production and match is selected at random.

Complexity of the transformation. Complex transformations require more complex mechanisms such as control structures 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. In *PROGRES*, sequence, branch and loop constructs can be used to control the application of graph transformations [37]. The same is true for the *Fujaba* tool suite. In *GReAT*, the control structure is based on hierarchical dataflow-like diagrams, but an explicit loop control structure is missing. Loops can be expressed as transitions to previous states in a dataflow.

Even with these advanced control structure mechanisms it remains to be seen whether graph transformation alone suffices to express complex transformations. For example, experience with *PROGRES* suggests that, for practical purposes, a hybrid approach combining the virtues of a graph transformation language and a textual constraint language is required. 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.

Preservation. Graph transformation theory seems promising to formally specify refactorings and to show that these refactorings preserve behavioural properties. An initial feasibility study has shown, however, that current graph transformation formalisms still have an number of limitations in order to accomplish this goal [32]. Therefore, Van Eetvelde and Janssens [33] 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.

4.3 Success criteria for a transformation language or tool

Ability to create/read/update/delete transformations (CRUD). This ability is present in all of the graph transformation tools we studied, as it forms the essence of any graph transformation approach.

Ability to suggest 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. The latter constraints are called *negative application conditions* [38]. In AGG, one can use the graphical user interface to map constraints to the pre- and postconditions of graph transformation rules [39]. In Fujaba, control constructs can be used to combine graph constraints and graph transformations in arbitrary ways.

Ability to customise or reuse transformations. A simple yet crucial way to customise graph transformations is by means of *parameterisation*. In fact, 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 most available 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.

VPM, the metamodelling framework used by the VIATRA tool, also supports the reusability of transformations by means of rule inheritance [40].

Ability to verify and guarantee correctness of the transformations. A model is syntactically correct if it conforms to its metamodel structure and well-formednes rules. In graph transformation, structure conformance is enforced by means of type graphs, and well-formedness rules can be expressed and checked by means of graph constraints. Heckel *et al.* have explored how such well-formedness rules can be combined. They used *critical pair analysis* to check the termination

and confluence of graph grammars [41]. 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. *AGG* is the only graph transformation tool that implements a critical pair analysis algorithm.

VIATRA [27] and GROOVE [28] are two graph transformation tools that check models for semantic correctness. 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) [27].

In order to guarantee syntactic correctness of a graph transformation, one should first ensure that the transformation itself is type correct. This can be done by considering a transformation itself as a model and checking whether it conforms to the metamodel of the graph transformation language. Next, by ensuring that the postcondition of a graph transformation rule does not conflict with the well-formedness rules of the target graph, one can remove the necessity of checking the type correctness of the output graph on each rule application.

Ability to test and validate 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 testing and debugging in the context of graph transformation [42, 43].

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 (VIsual Automated model TRAnsformations), 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 [27].

Ability to deal with incomplete or inconsistent models. In all the graph transformation approaches 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.

Ability to group, compose and decompose 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 [44]. Typical control mechanisms are sequencing, branching and looping. In Fujaba, transformations are implemented as method bodies, so composition of transformations can be achieved by performing method calls. Another

mechanism that has been proposed to group and compose graph transformation is the structuring mechanism of *graph transformation units* [45, 46]. 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.

Ability to specify generic and higher-order transformations. The only graph transformation tool that supports *higher-order transformations* is VIATRA [47]. 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!

Ability to specify bidirectional transformations. All the graph transformation mechanisms we studied were unidirectional. This means that a graph transformation can be applied in only one direction.

Support for 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.

4.4 Quality requirements for a transformation language or 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 languages such as *GReAT* and *VIATRA*. That graph transformations are also *usable* is more difficult to assess, as this depends on several factors. Abstracting away from factors such as developer training and experience, most graph transformation tools still need to mature to make them more performant and intuitive in use, but this is probably 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. While this may be the case for some tools, this is not an inherent limitation of the paradigm per se. For example, Varró *et al.* show how to transform higher-order model transformations automatically in efficient first-order transformations [47]. As another example, Vizhanyo et al. have illustrated significant performance gains by optimising traditional graph matching algorithms on the one hand and bypassing the generic transformation engine of GReAT by native transformation code on the other hand [48].

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. In fact, a comparison of GROOVE with SPIN – a well-known model checking tool that does not rely on graph transformation – showed that for a certain type of applications GROOVE even outperforms SPIN since it was able to deal with significantly larger state spaces [49].

Extensibility. Some graph transformation tools, such as *Fujaba*, offer a built-in plug-in mechanism for extending the tool with new functionality.

Mathematical properties. The formal foundation of graph transformation technology is one of its main advantages. Research literature on theoretical issues of graph transformation is abundant.

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, the "structured procedural" style in MOLA is probably more readable than the declarative grammar approach of AGG. Similarly, for people accustomed to UML notation, Fujaba provides a very natural notation to express graph transformations without the user even being aware of it.

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

Graph transformation tools can also be applied using MDA standards like UML, MOF and XMI. Type graphs can defined as class diagrams in UML editors with proper XMI export facilities. The resulting model can be transformed into a MOF metamodel [50]. Existing MDA frameworks can be used to monitor the OCL well-formedness rules of this metamodel on models residing in a MOF repository [51]. 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 [35, 36].

Finally, although there is no formal concensus yet on a standard graph transformation language or exchange format, there is an ongoing initiative in this direction. Based on the relative success of GXL, a standard graph exchange format [29] that is used in the AGG tool, work is in progress to come to GTXL, a standard exchange format for graph transformations [52].

4.5 Comparison

In this section we apply the above discussion in practice to compare three state-of-the-art graph transformation tools. To cover as wide a spectrum as possible, we have chosen three different kinds of graph transformation tools:

- *AGG* is a general-purpose graph transformation tool
- *Fujaba* is a tool for round-trip engineering (between UML and Java)
- *VIATRA* is a tool dedicated to model transformation

Table 4.5 summarises the results of our comparison. Note that not all criteria in our taxonomy appear in the table, since we only display those criteria where a difference could be discerned between the considered tools.

5 Conclusion

In this paper, we provided a *taxonomy* of model transformation. To goal is to help the developer choosing a particular transformation language, tool, method or technology for his specific needs. This is important, since the application domains of model transformation technology can be very diverse. Therefore, there is no unique answer to the question which approach to model transformation is the best. Nevertheless, we have shown that it is possible to come up with a set of concrete criteria that need to be taken into consideration when dealing with the following crucial questions:

- What needs to be transformed into what?
- What are the important characteristics of a model transformation?
- What are the success criteria for a transformation language or tool?
- Which mechanisms can be used for model transformations?

Based on the answers to these questions, a particular model transformation approach may be selected and adopted to address a particular problem.

We validated our taxonomy by applying it to graph transformation technology, which comprises a set of techniques and associated formalisms that are directly applicable to model transformation. Based on this, we were able to conclude that graph transformation is a promising approach to deal with model transformation. It is a visual notation: the source, target and the transformation itself can be expressed in a visual way. It is formally founded: one can resort

Table 2. Comparison of graph transformation tools for model transformation

criterion	AGG	Fujaba	VIATRA
number of source and target models	one-to-one	many-to-many	many-to-many
kind of transformation	endogenous	endogenous	endogenous + exogenous
technological space	GXL	MDA[4]	MDA
level of automation	graph grammars	graph transformations	high
complexity	layered grammars	controlled graph transformations	higher-order and meta transformations
reusability	parameterised transformations	parameterised transfos / inheritance of transfos	rule inheritance and higher-order transformations
verification and validation	critical pair analysis	graph-transformation-based JUnit tests	full-fledged verification and validation mechanism[49]
composition	layered graph grammar	method calls in story driven modelling	?
usability	low	high	high
extensibility	None	plug-in mechanism	None
acceptability	low	medium	high
standardisation	GXL	UML, Java, XMI, MOF	UML, XMI, MOF

to many theorems to prove certain properties of the transformation. It offers a mechanism to compose smaller transformations into more complex ones.

A possible disadvantage is that the various techniques for graph transformations are not necessarily compatible with one another. With respect to standardisation, 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 usable and acceptable since it was specifically built for the purpose of model transformation. *VIATRA* seems to be the most advanced tool, even compared with other model transformation tools that are not based on graph transformation, since it offers very advanced features such as rule inheritance, higher-order transformations, meta transformations, and verification and validation of transformations.

References

1. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of MDA. In: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer-Verlag (2002) 90–105 Proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.
2. Sendall, S., Kozaczynski, W.: Model Transformation - The Heart and Soul of Model-Driven Software Development. IEEE Software, Special Issue on Model Driven Software Development (2003) 42–45
3. Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: Proc. 16th Int'l Conf. Automated Software Engineering, IEEE Computer Society (2001) 273
4. Seidewitz, E.: What models mean. IEEE Software **20** (2003)
5. Favre, J.M.: Towards a basic theory to model model driven engineering. In: Proc. 3rd Workshop in Software Model Engineering (Satellite workshop at the 7th International Conference on the UML). (2004)
6. Janssens, D., Demeyer, S., Mens, T.: Case study: Simulation of a LAN. Electronic Notes in Theoretical Computer Science **72** (2003)
7. Wasowski, A.: Flattening statecharts without explosions. In: Proc. ACM SIGPLAN/SIGBED Conf. Languages, compilers, and tools. (2004) 257–266
8. Fowler, M., Scott, K.: UML Distilled – Second Edition – A Brief Guide to the Standard Object Modeling Language. Object technology series. Addison-Wesley (1999)
9. Kleppe, A., Warmer, J., Bast., W.: MDA Explained, The Model-Driven Architecture: Practice and Promise. Addison Wesley (2003)
10. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. (2003)
11. Object Management Group: MOF 2.0 Query / Views / Transformations RFP ad/2002-04-10 (2002) URL: http://www.omg.org/cgi-bin/apps/doc?ad/02-04-10.pdf.
12. Object Management Group: Unified Modeling Language specification version 1.5. formal/2003-03-01 (2003)

13. Visser, E., et al.: A taxonomy of program transformation (2004) URL: http://www.program-transformation.org/Transform/ProgramTransformation.
14. Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley (1999)
15. Wirth, N.: Program development by stepwise refinement. Comm. ACM **14** (1971) 221–227
16. Back, R.J., von Wright, J.: Refinement Calculus. Springer Verlag (1998)
17. Back, R.: On correct refinement of programs. Journal of Computer and Systems Sciences **23** (1981) 49–68
18. van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Proc. 9th Working Conf. Reverse Engineering, IEEE Computer Society (2002) 97–107
19. Tourwé, T., Mens, T.: Identifying refactoring opportunities using logic meta programming. In: Proc. 7th European Conf. Software Maintenance and Re-engineering (CSMR 2003), IEEE Computer Society (2003) 91–100
20. Van Gorp, P., Janssens, D., Gardner, T.: Write Once, Deploy N: a performance oriented mda case study. In: Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference, IEEE (2004)
21. Akehurst, D., Kent, S.: A relational approach to defining transformations in a metamodel. In: Proc. 5th Int'l Conf. UML. Volume 2460 of Lecture Notes in Computer Science., Springer-Verlag (2002) 243–258
22. Sprinkle, J., Agrawal, A., Levendovszky, T., Shi, F., Karsai, G.: Domain model translation using graph transformations. In: Proc. Int'l Conf. Engineering of Computer-Based Systems, IEEE Computer Society (2003) 159–168
23. Courcelle, B.: Graph rewriting: an algebraic and logic approach. In Leeuwen, J.V., ed.: Handbook of Theoretical Computer Science. Volume B., Elsevier (1990) 193–242
24. Niere, J., Zündorf, A.: Testing and simulating production control systems using the Fujaba environment. In: Proc. AGTIVE 99. Volume 1779 of Lecture Notes in Computer Science., Springer-Verlag (1999) 449–456
25. Taentzer, G.: AGG: A tool environment for algebraic graph transformation. In: Proc. AGTIVE 99. Volume 1779 of Lecture Notes in Computer Science., Springer-Verlag (1999) 481–488
26. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. In: Proc. Model-Driven Architecture: Foundations and Applications. (2004) 14–28
27. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - visual automated transformations for formal verification and validation of UML models. In: Proc. 17th Int'l Conf. Automated Software Engineering, IEEE Computer Society (2002) 267–270
28. Rensink, A.: The GROOVE simulator a tool for state space generation. In: Proc. AGTIVE 2003. Volume 3062 of Lecture Notes in Computer Science., Springer-Verlag (2004) 479–485
29. Holt, R., Schürr, A., Sim, S.E., Winter, A.: Graph eXchange Language. http://www.gupro.de/GXL/ (2004)
30. Karsai, G., Agrawal, A., Shi, F.: On the use of graph transformations for the formal specification of model interpreters. Journal of Universal Computer Science **9** (2003) 1296–1321
31. Baresi, L., Heckel, R., Thöne, S., Varró, D.: Style-based refinement of dynamic software architectures. In: Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA4), IEEE Computer Society (2004) 155–164

32. Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour preserving program transformations. In: Proc. 1st Int'l Conf. Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer-Verlag (2002) 286–301
33. Van Eetvelde, N., Janssens, D.: Extending graph rewriting for refactoring. In: Proc. 2nd Int'l Conf. Graph Transformation. Volume 3526 of Lecture Notes in Computer Science., Springer-Verlag (2004) 399–415
34. M.Große-Rhode, F. Parisi Presicce, M.S.: Formal software specification with refinements and modules of typed graph transformation systems. Journal of Computer and System Sciences **64** (2002) 171–218
35. Schippers, H., Van Gorp, P., Janssens, D.: Leveraging UML profiles to generate plugins from visual model transformations. In: Proc. Int'l Workshop Software Evolution through Transformations (SETra). (2004) To appear in ENTCS.
36. Formal Techniques in Software Engineering: Model driven, Template based, Model Transformer (MoTMoT). http://sourceforge.net/projects/motmot/ (2004)
37. Schürr, A., Winter, A., Zündorf, A.: PROGRES: Language and Environment. In: Handbook of Graph Grammars and Graph Transformation. World scientific (1999) 487–550
38. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
39. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars: A constructive approach. Lecture Notes in Theoretical Computer Science **1** (1995)
40. Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Software and Systems Modeling **2** (2003) 187–210
41. Heckel, R., Jochen Malte Küster, Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Proc. 1st Int'l Conf. Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer-Verlag (2002) 161–176
42. Geiger, L., Zündorf, A.: Transforming graph based scenarios into graph transformation based JUnit tests. In: AGTIVE. Volume 3062 of Lecture Notes in Computer Science., Springer (2004) 61–74
43. Geiger, L., Zündorf, A.: Graph based debugging with Fujaba. In: Proc. Int'l Workshop on Graph Based Tools. Volume 72 of Electronic Notes in Theoretical Computer Science., Elsevier (2002)
44. Schürr, A.: Logic based programmed structure rewriting systems. Fundamenta Informaticae **26** (1996) 363–385
45. Hans-Jörg Kreowski, Kuske, S.: Graph transformation units and modules. Handbook of Graph Grammars and Computing by Graph Transformation **2** (1999) 607–638
46. Klempien-Hinrichs, R., Kreowski, H.J., Kuske, S.: Typed graph transformation units. In: Proc. 2nd Int'l Conf. Graph Transformation. Volume 3526 of Lecture Notes in Computer Science., Springer-Verlag (2004) 112–127
47. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In Thomas Baar, Alfred Strohmeier, A.M., ed.: UML 2004 - The Unified Modeling Language. Volume 3273 of Lecture Notes in Computer Science., Springer-Verlag (2004) 290–304
48. Vizhanyo, A., Agrawal, A., Shi, F.: Towards generation of high-performance transformations. In: Proc. Generative Programming and Component Engineering. Lecture Notes in Computer Science, Springer-Verlag (2004)

49. Rensink, A., Schmidt, A., Varró, D.: Model checking graph transformations: A comparison of two approaches. In: Proc. 2nd Int'l Conf. Graph Transformation. Volume 3526 of Lecture Notes in Computer Science., Springer-Verlag (2004) 226–241
50. Matula, M.: UML2MOF tool – netbeans / sun microsystems. http://mdr.netbeans.org/uml2mof/ (2004)
51. Akehurst, D., Patrascoiu, O.: Object Constraint Language library. http://www.cs.kent.ac.uk/projects/ocl/ (2004)
52. Täntzer, G.: XML-based exchange formats for graphs and graph transformation systems. http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html (2004)