Classification of Model Transformation Approaches

Krzysztof Czarnecki and Simon Helsen
University of Waterloo, Canada
czarnecki@acm.org, shelsen@computer.org

1 Introduction

The Model-Driven Architecture (MDA) [Fra03] is an initiative by the Object Management Group (OMG) to define an approach to software development based on modeling and automated mapping of models to implementations. The basic MDA pattern involves defining a platform-independent model (PIM) and its automated mapping to one or more platform-specific models (PSMs).

The MDA approach promises a number of benefits including improved portability due to separating the application knowledge from the mapping to a specific implementation technology, increased productivity due to automating the mapping, improved quality due to reuse of well proven patterns and best practices in the mapping, and improved maintainability due to better separation of concerns.

While the current OMG standards such as the Meta Object Facility (MOF) [MOF] and the UML [UML] provide a well-established foundation for defining PIMs and PSMs, no such well-established foundation exists for transforming PIMs into PSMs [GLR+02]. In 2002, in its effort to change this situation, the OMG initiated a standardization process by issuing a Request for Proposal (RFP) on Query / Views / Transformations (QVT) [QVT]. This process will eventually lead to an OMG standard for defining model transformations, which will be of interest not only for PIM-to-PSM transformations, but also for defining views on models and synchronization between models. Driven by practical needs and the OMG's request, a large number of approaches to model transformation have recently been proposed.

In this paper, we propose a feature model to compare different model transformation approaches and offer a survey and categorization of over 20 existing approaches

- published in the literature (GreAT [AKS03], UMLX [Wil03], ATOM [ATOM], VIATRA [VVP02], BOTL [BM03, MB03], relational approach in [AK02], logic-programming approaches in [GLR+02]),
- submitted in response to the OMG's QVT RFP in the 2nd submission round ([QVTP], [CDI], [AST+], [IOPT]),
- implemented in open-source MDA tools (Jamda [JAM], AndroMDA [AND], FUUT-je and GMT [FUU]), and
- implemented in commercial MDA tools (OptimalJ [OTPJ], ArcStyler [AS], XDE [XDE], Codagen Architect [CA], b+m Generator Framework [B+M])

The feature model makes the different possible design choices for a model transformation approach explicit, which is the main contribution of this paper. We do not give the detailed classification data for each individual approach; this is because it would require a significant space and the details of the individual approaches are a moving target anyway. Instead, we propose a clustering of the existing approaches into a few major categories that capture their different flavours (and main design choices).

The paper is organized as follows. Section 2 presents our feature model of model transformation approaches. Section 3 presents the major categories of existing transformation approaches. Section 4 concludes the paper with some remarks on the practical applicability of the different categories.

2 Design Features of Model Transformation Approaches

This section is the result of applying domain analysis to existing model transformation approaches. Domain analysis is concerned with analyzing and modeling the variabilities and commonalities of systems (or concepts) in a domain [Cza02]. We document our results using feature diagrams [KCH+90, Cza02], which are a common notation in domain analysis. Fig. 1 shows the top-level feature diagram, where each subnode represents a major area of variation. Further explanation of the notation is given in the legend of Fig. 2.

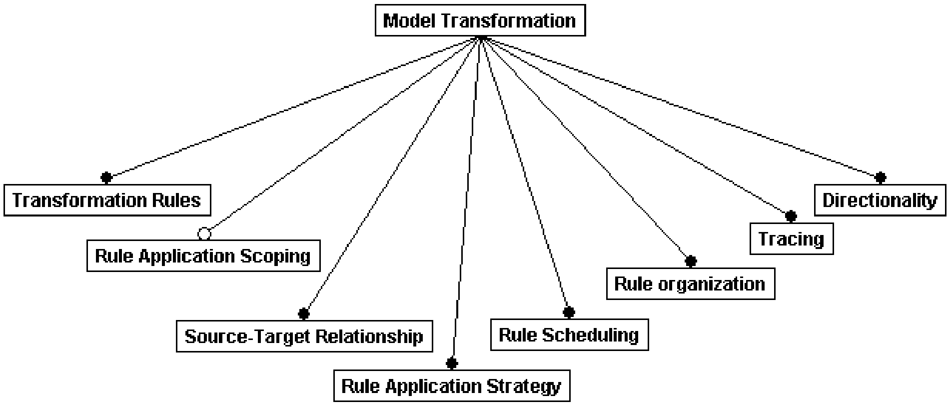

Fig. 1 Feature diagram representing the top-level areas of variation[1]

Essentially, a feature diagram defines a taxonomy. We should note that we do not aim for this taxonomy to be normative. Unfortunately, the relatively new area of model transformations has many overloaded terms, and many of the terms we use in our taxonomy are often used with different meanings in the original descriptions of the different approaches. For example, we use the term pattern to denote a model fragment with zero or more metavariables, while some approaches (e.g., [IOPT]) use this term to denote any abstract specification of some constellation of model elements in a model. However, we provide the definitions of the terms as we use them.

Each of the following elaborates on one major area of variation from Fig. 1 by giving its feature diagram, describing the different choices in the text, and providing examples of approaches supporting a given feature. The combination of feature diagrams and the additional information is referred to as a feature model. Please note that our feature model treats model-to-model and model-to-code approaches uniformly. We will distinguish between these categories later in Section 3.

2.1 Transformation Rules

---

[1] The feature diagrams in this paper have been created using CaptainFeature, a feature modeling tool available from http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/captainfeature/

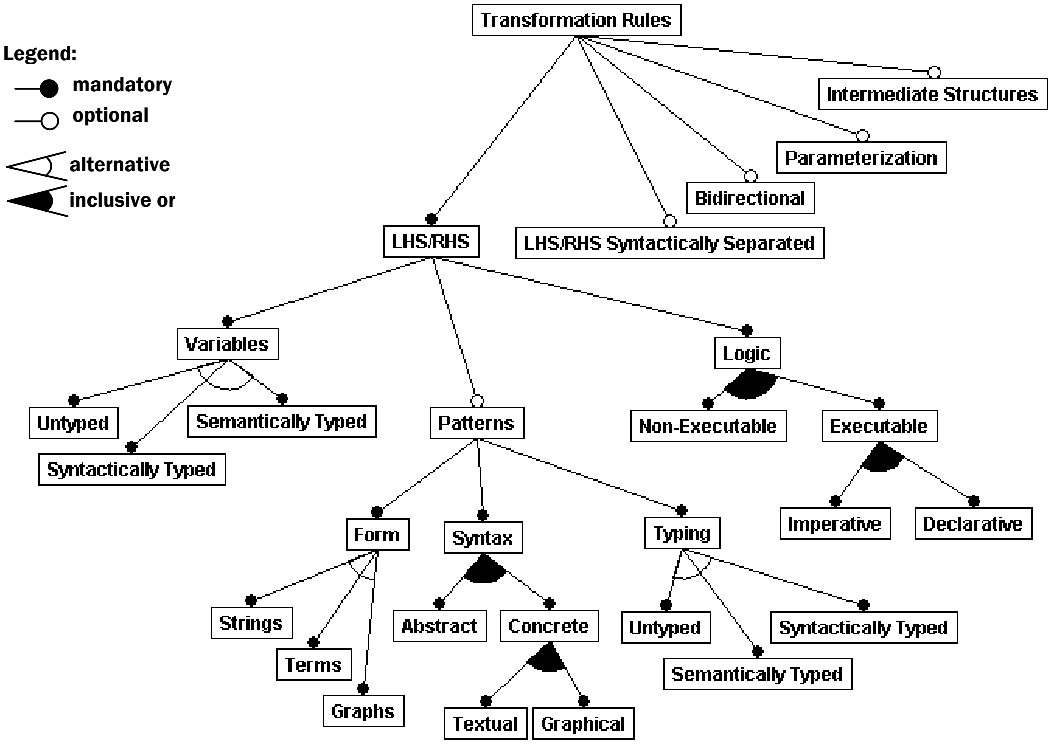

Fig. 2 Features of transformation rules

A transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS).[2] The LHS accesses the source model, whereas the RHS expands in the target model. The RHS and LHS may or may not be syntactically separated. In other words, the rule syntax may specifically mark RHS and LHS as such (as in classical rewrite rules), or there might be no syntactic distinction (as in a transformation rule implemented as a Java program; see Section 3.2.1).

Both LHS and RHS can be represented using any mixture of the following:

- *Variables*: Variables hold elements from the source and/or target models (or some intermediate elements). They are sometimes referred to as metavariables to distinguish them from variables that may be part of the transformed model (e.g., Java variables in transformed Java programs).
- *Patterns*: Patterns are model fragments with zero or more variables. We can have string, term, and graph patterns. String patterns are used in textual templates (see Section 3.1.2). Model-to-model transformations usually use term or graph patterns (see Section 3.2.2 and 3.2.3). Patterns can be represented using abstract or concrete syntax of the corresponding source or target model language, and the syntax can be textual and/or graphical (see Section 3.2.3).
- *Logic*: Logic expresses computations and constraints on model elements. Logic may be non-executable or executable. Non-executable logic is used to specify relationship between models (e.g., [QVTP]). Executable logic can take a declarative or imperative form. Examples of the declarative form include OCL [OCL] queries to retrieve elements

---

[2] We view templates as a special case of transformation rules. The LHS may be as minimal as a parameter list, but may also contain some logic to access the source model. The RHS contains at least a pattern, but may also include additional logic performing pattern composition.

from the source model (e.g., XDE) and the implicit creation of target elements through constraints (e.g., [CDI]). Imperative logic has often the form of Java code calling repository APIs (such as JMI [JMI]) to manipulate models directly.

Both variables and patterns can be untyped, syntactically typed, or sematically typed. In the case of syntactic typing, a variable is associated with a metamodel element whose instances it can hold. Semantic typing allows stronger properties to be asserted. For example, the syntactic type of a variable could be "expression," whereas its semantic type could be "expression evaluating to an integer value." The latter is not available in the current model transformation languages, but is supported in some metaprogramming languages such as MetaML and MetaOcaml [MML, MOML]. We included semantic typing in the feature model to indicate possible future development.

Two other aspects of transformation rules are

- *Bidirectionality*: A rule may be executable in the inverse direction (see Section 2.8).
- *Rule parameterization*: Transformation rules may have additional control parameters allowing configuration and tuning.
- *Intermediate structures*: Some approaches (e.g., VIATRA and GreAT) require the construction of intermediate model structures.

2.2 Rule Application Scoping

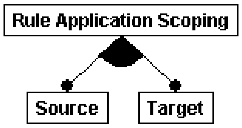

Fig. 3 Features of rule application scoping

Source scope is the scope of the source model that is considered for rule application. Some approaches support flexible scoping (e.g. XDE and GreAT), where a scope smaller than the entire source model can be set. The latter is important particularly for performance reasons. Target scope is the scope of the target model, in which the RHS will be expanded. Target scoping may also be provided (e.g., XDE).

2.3 Relationship Between Source and Target

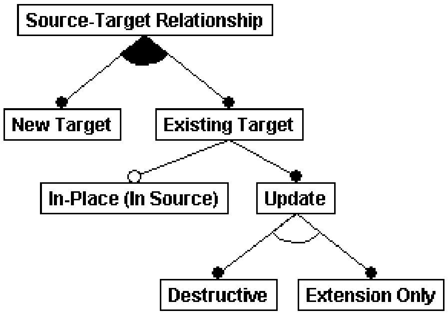

Fig. 4 Features of the relationship between source and target

Some approaches mandate the creation of a new target model that has to be separate from the source (e.g., IBM/DSTC). In some other approaches, source and target is always the same model, i.e., they only support in-place update (e.g., VIATRA). Yet other approaches (e.g., XDE) allow setting the target scope to a new model or (possibly parts of) an existing one, which could be the original source model (i.e., in-place update). Furthermore, an approach could allow a destructive update of the existing target or an update by extension only. Approaches using non-deterministic selection and fixpoint iteration scheduling may restrict in-place update to extension in order to ensure termination (e.g., VIATRA).

2.4 Rule Application Strategy

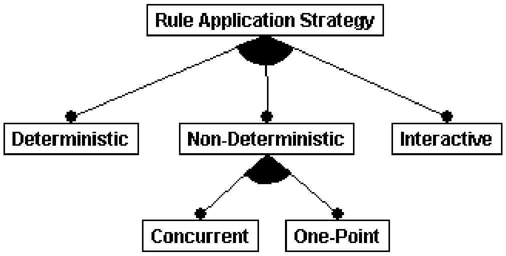

Fig. 5 Features of rule application strategy

A rule needs to be applied to a specific location within its source scope. Since there may be more than one match for a rule within a given source scope, we need an application strategy. The strategy could be deterministic or non-deterministic. For example, a deterministic strategy could exploit some standard traversal strategy (such as depth-first) over the containment hierarchy in the source. Stratego [STR] is an example of a term rewriting language with rich mechanisms to express traversal in tree structures. Examples of non-deterministic strategies include one-point application, where a rule is applied to one non-deterministically selected location, and concurrent application, where one rule is applied concurrently to all matching locations in the source (e.g., VIATRA).

The target location for a rule can be determined automatically or manually. In the case of in-place update, the source location becomes the target location (as in VIATRA). In an approach with separate source and target models, traceability links can be used to determine the target (as in IBM/DSTC): A rule may follow the tractability link to some target element that was created by some other rule and use the element as its own target. Finally, the target could be selected interactively by the user (as offered in XDE).

2.5 Rule Scheduling

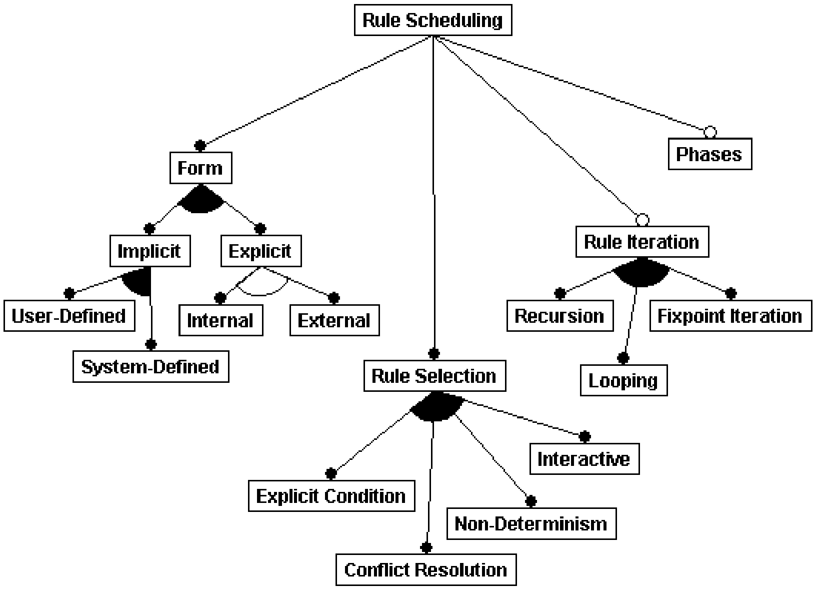

Fig. 6 Features of rule scheduling

Scheduling mechanisms determine the order in which the rules are applied. The scheduling mechanism can vary in four main areas:

- *Form*: The scheduling aspect can be expressed implicitly or explicitly. Implicit scheduling can be system or user defined. Implicit user-defined scheduling involves designing the patterns and logic of the rules to guarantee certain execution orders. For example, a given rule could check for some information that only some other rule would produce. Explicit scheduling has dedicated constructs to explicitly control the execution order. Explicit scheduling could be internal or external. In external scheduling, there is a clear separation between the rules and the scheduling logic (e.g., VIATRA), which is not the case in internal scheduling. An example of internal scheduling would be a mechanism allowing a transformation rule to directly invoke other rules (e.g., [CDI] and most template approaches in Section 3.1.2, which offer a way to call other templates).
- *Rule selection*: Rules can be selected by an explicit condition. Some approaches allow non-deterministic choice and/or conflict resolution mechanisms (e.g., based on priorities). Any combination of these selection mechanisms could be offered.
- *Rule iteration*: Rule iteration mechanisms include recursion, looping, and fixpoint iteration (i.e., repeated application until no changes detected). Any combination of these mechanisms could be offered.
- *Phasing*: The transformation process may be organized into separate phases, where each phase has a specific purpose and only certain rules can be invoked in a given phase. For example, structure-oriented approaches (see Section 3.2.4) have a separate phase to create the containment hierarchy of the target model and a separate phase to set the attributes and references in the target.

2.6 Rule Organization

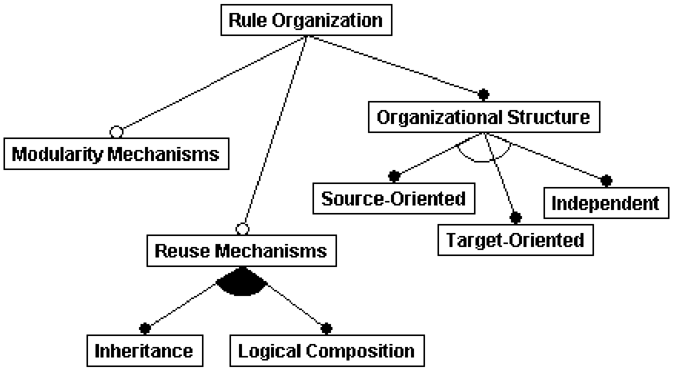

Fig. 7 Features of rule organization

We consider three areas in the context of organizing multiple transformation rules:

- *Modularity mechanisms*: Some approaches allow packaging rules into modules (e.g., [AST+] and VIATRA). A module can import another module to access its content.
- *Reuse mechanisms*: Reuse mechanisms offer a way to define a rule based on one or more other rules. In general, scheduling mechanisms can be used to define composite transformation rules; however, some approaches offer dedicated reuse mechanisms such as inheritance between rules (e.g., [AST+], derivation in [IOPT], extension in [CDI], specialization in [QVTP] or modules [AST+], and composition by logic operators (e.g., [QVTP]).
- *Organizational structure*: Rules may be organized according to the structure of the source language (as in attribute grammars, where actions are attached to the elements of the source language) or the target language, or they may have their own independent organization. An example of the organization according to the structure of the target is [IOPT]. In this approach, there is one rule for each target element type and the rules are nested according to the containment hierarchy in the target metamodel. For example, if the target language has a package construct in which classes can be nested, the rule for creating packages will contain the rule for creating classes (which will contain rules for creating attributes and methods, etc.).

2.7 Traceability Links

Transformations may record links between their source and target elements. These links can be useful in performing impact analysis (i.e., analyzing how changing one model would affect other related models), synchronization between models, model-based debugging (i.e., mapping the stepwise execution of an implementation back to its high-level model), and determining the target of a transformation (see Section 2.4).

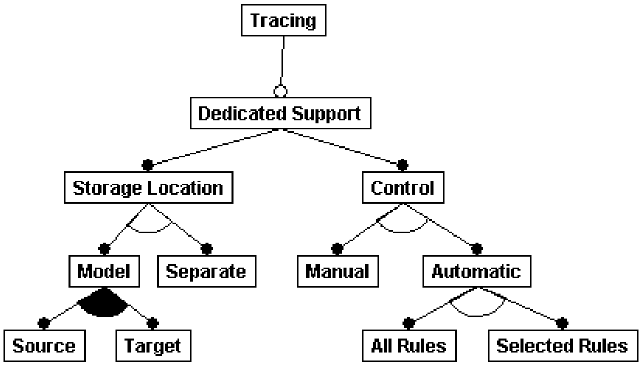

Fig. 8 Features of tracing

Some approaches provide dedicated support for traceability (e.g., [CDI]), while other expect the user to encode traceability using the same mechanisms as for adding any other kinds of links in models (e.g., VIATRA, GreAT). Some approaches in the latter category require developers to manually encode the creation of traceability links in the transformation rules (e.g., [CDI]), while other create traceability links automatically (e.g., [IOPT]). In the case of automated support, the approach may still provide some control over which traceability links get created (in order to limit the amount of traceability data that gets generated). Finally, there is the choice of location where the links are stored, e.g., in the source and/or target, or separately. A preferable approach is to store a GUID in each model element and store the traceability information separate from the source and target.

2.8 Directionality

Most rules are applied in one direction by binding the LHS in the source and expanding the RHS in the target model. In some cases, a declarative rule (i.e., one that only uses declarative logic and/or patterns) can be applied in the inverse direction, too. This property seems attractive in the context of synchronization between models. An alternative approach is to define two separate rules, one for each direction.

Transformation rules are usually designed to have a functional character: given some input in the source model, they produce a concrete result in the target model. However, since different inputs may lead to the same output, the inverse of a rule may not be a function. In this case, the inversion could enumerate a number of possible solutions (which could be infinite), or just establish part of the result in a concrete way (because the part could be the same for all solutions) and use variables, defaults, or values already present in the output for the other parts. However, inversing a set of rules may also fail to produce any result due to non-termination.

3 Major Categories

At the top level, we distinguish between model-to-code and model-to-model transformation approaches. In general, we can view transforming models to code as a special case of model-to-model transformations; we only would need to provide a metamodel for the target programming language. However, for practical reasons of reusing existing compiler technology, code is often generated simply as text, which is then fed into a compiler. For this reason, we distinguish between model-to-code transformation (which would be better described as model-to-text) and model-to-model transformation. Several tools offer both model-to-model and model-to-code transformations (e.g., Jamda, XDE, and OptimalJ).

In the model-to-code category, we distinguish between visitor-based and template based approaches. In the model-to-model category, we distinguish among direct manipulation approaches, relational approaches, graph-transformation-based approaches, structure-driven approaches, and hybrid approaches.

3.1 Model-To-Code Approaches

3.1.1 Visitor-Based Approaches

A very basic code generation approach consists in providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream. Example of this approach is Jamda. Jamda is an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism (so called CodeWriters) to generate code. Jamda does not support the MOF standard to define new metamodels; however, new model element types can be introduced by subclassing the existing Java classes that represent the predefined model element types.

3.1.2 Template-Based Approaches

The majority of currently available MDA tools support template-based model-to-code generation, e.g., b+m Generator Framework, FUUT-je, Codagen Architect, AndroMDA, ArcStyler, OptimalJ and XDE (the latter two also provide model-to-model transformations). AndroMDA reuses existing open-source template-based generation technology, namely Velocity [VELO] and XDoclet [XD].

A template usually consists of the target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion (see [Cle01] for a introduction to template-based code generation). According to our terminology, the LHS uses executable logic to access source; the RHS combines untyped, string patterns with executable logic for code selection and iterative expansion; and there is no syntactic separation between the LHS and RHS. Template approaches usually offer user-defined scheduling in the internal form of calling a template from within another one.

The LHS logic accessing the source model may have different forms. The logic could be simply Java code accessing the API provided by the internal representation of the source model (e.g., JMI), or it could be declarative queries (e.g., in OCL or XPath [XP]). The b+m Generator Framework propagates the idea of separating more complex source access logic (which might need to navigate and gather information from different places of the source model) from templates by moving them into user-defined operations of the source-model elements.

Compared to a visitor-based transformation, the structure of a template resembles more closely the code to be generated. Templates lend themselves to iterative development as they can be easily derived from examples. Since the template approaches discussed in this section operate on text, the patterns they contain are untyped and can represent syntactically or semantically incorrect code fragments. On the other hand, textual templates are independent of the target language and simplify the generation of any textual artefacts, including documentation.

A related technology is frame processing, which extends templates with more sophisticated adaptation and structuring mechanisms (Bassett's frames [Bas97], XVCL [XVCL], FPL [FPL], ANGIE [ANG]). To our knowledge, FPL and ANGIE have been applied to generate code from models.

3.2 Model-To-Model Approaches

Model-to-model transformations translate between source and target models, which can be instances of the same or different metamodels. All of these approaches support syntactic typing of variables and patterns.

Most existing MDA tools provide only model-to-code transformation, which they use for generating PSMs (in this case being just the implementation code) from PIMs. Why are model-to-model needed? When bridging large abstraction gaps between PIMs and PSMs, it is easier to generate intermediate models rather than go straight to the target PSM. For example, when going from a class diagram to an EJB implementation, tools such as OptimalJ would generate an intermediate EJB component model, which contains all the necessary information to produce the actual Java code from it. This makes the transformations more modular and maintainable. Also, intermediate models may be needed for optimization and tuning, or at least for debugging purposes. In addition to PIM-to-PSM transformation, model-to-model transformations are useful for computing different views of a system model and synchronizing between them.

3.2.1 Direct-Manipulation Approaches

These approaches offer an internal model representation plus some API to manipulate it. They are usually implemented as an object-oriented framework, which may also provide some minimal infrastructure to organize the transformations (e.g., abstract class for transformations). However, users have to implement transformation rules and scheduling mostly from scratch using a programming language such as Java. Examples of this approach include Jamda and implementing transformations directly against some MOF-compliant API (e.g., JMI).

3.2.2 Relational Approaches

This category groups declarative approaches where the main concept is mathematical relations (e.g., [AK02], [QVTP], [CDI], declarative approaches in [GLR+02], and mapping rules in [AST+]).

The basic idea is to state the source and target element type of a relation and specify it using constraints. In its pure form, such specification is non-executable (e.g., [AK02], relations in [QVTP], and mapping rules in [AST+]). However, declarative constraints can be given executable semantic, much like in the case of logic programming. In fact, logic programming with its unification-based matching, search, and backtracking seems a natural choice to implement the relational approach, where predicates can be used to describe the relations. In [GLR+02], Gerber et al explore the application of logic programming (in particular Mercury, a typed dialect of Prolog, and F-logic, an object-oriented logic paradigm) to implement transformations. The QVT proposal in [CDI] was inspired by the F-logic approach. The approach in [QVTP] distinguishes between relations, which in their framework are bi-directional, non-executable specifications of transformations, and mappings, which are executable, unidirectional transformations implementing relations.

All of the relational approaches are side-effect-free. They often support backtracking ([GRL+02] and [QVTP]) and, in contrast to the imperative direct manipulation approaches in Section 3.2.1, create target elements implicitly (e.g., [GRL+02], [CDI]). Relational specifications ([AK02], relations in [QVTP], and mapping rules in [AST+]) can be interpreted bi-directionally. Logic-programming-based approaches also naturally support bi-directionality. But some approaches fix the direction for executable transformations (as [CDI] and mappings in [QVTP]). Logic-programming-based approaches (e.g., [GRL+02] and [CDI]) require strict separation between source and target models (i.e., they do not allow in-place update).

3.2.3 Graph-Transformation-Based Approaches

This category of model transformation approaches draws on the theoretical work on graph transformations. In particular, these approaches operate on typed, attributed, labelled graphs [AEH+96], which is a kind of graphs specifically designed to represent UML-like models. Examples of graph-transformation approaches to model transformation include VIATRA, ATOM, GreAT, UMLX, and BOTL.

Graph transformation rules consist of a LHS graph pattern and a RHS graph pattern. The graph pattern can be rendered in the concrete syntax of its respective (source or target) language (e.g., in VIATRA) or in the MOF abstract syntax (e.g., in BOTL). The former is preferred since for complex syntaxes (like UML) the latter may result in huge patterns even for relatively small transformations. The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place. The LHS often contains conditions in additional to the LHS pattern (e.g., negative conditions). Some additional logic (e.g., in string and numeric domains) is needed in order to compute target attribute values (such as element names). GreAT offers an extended form of patterns with multiplicities on edges and nodes. In most approaches, scheduling has external form and the scheduling mechanisms include nondeterministic selection, explicit condition, and iteration (incl. fixpoint iterations). Fixpoint iterations are particularly useful for computing transitive closures.

3.2.4 Structure-Driven Approaches

Approaches in this category have two distinct phases: the first phase is concerned with creating the hierarchical structure of the target model, whereas the second phase sets the attributes and references in the target. The overall framework determines scheduling and application strategy; users are only concerned with providing the transformation rules.

An example of the structure-driven approach is the model-to-model transformation framework provided by OptimalJ. The framework is implemented in Java and provides so-called incremental copiers that users have to subclass to define their own transformation rules. The basic metaphor is the idea of copying model elements from the source to the target, which then can be varied to achieve the desired transformation effect. The framework uses reflection to provide a declarative interface. A transformation rule is implemented as a method with an input parameter whose type determines the source type of the rule, and the method returns a Java object representing the class of the target model element. Rules are not allowed to have side effects and scheduling is completely determined by the framework.

Another structure-driven approach is [IOPT]. A special property of this approach is the target-oriented rule organization, where there is one rule per target element type and the nesting of the rules corresponds to the containment hierarchy in the target metamodel. The execution of this model can be viewed as a top-down configuration of the target model.

3.2.5 Hybrid Approaches

Hybrid approaches combine different techniques from the previous categories.

The Transformation Rule Language (TRL) [AST+] is a composition of declarative and imperative approaches. It could be also classified in the relational category, but we decided to classify it separately because of its stronger imperative component. Similar to [QVTP], it distinguishes between specification and implementation. A mapping rule in TRL declares a relationship between source and target elements that is constrained by a set of invariants. They are similar to relations in [QVTP] and fit into the relational category (Section 3.2.2). Operational rules in TRL represent executable transformation rules. In contrast to mapping rules, operational rules explicitly state whether a rule creates, update, or deletes elements. Scheduling is explicit in internal form, where a rule explicitly calls other rules in its body. Rule inheritance is supported. Rules can be organized into modules (called *units*). Inheritance between modules (with overriding) is also supported.

XDE is an example of a highly hybrid approach. XDE supports model-to-model transformation through its pattern mechanism. The original motivation for patterns in XDE was to provide automated application of design patterns. Consequently, the basic concept of XDE pattern mechanism is a parameterized collaboration, which is the UML mechanism to model design patterns. With general model-to-model transformations as a subsequent goal, the basic pattern mechanism evolved into a highly hybrid and rather complex approach. A pattern is represented as a package containing the parameterized collaboration and a number of other models that can be automatically customized and copied and/or merged into the target using imperative Java callouts. Upon pattern application, parameters can be bound interactively through a wizard, or they also can be bound automatically. The automatic selection of source elements may be achieved declaratively through OCL queries or trough imperative Java callouts. Repeated pattern application is supported through collection-typed parameters. Each pattern application gets recorded together with all its parameter bindings, and the record can be used to later reapply the pattern with the original parameter bindings. XDE does not put any constraints on the relationship between source and target, i.e., creation of a new target, in-place update, and update of another existing target model are possible. Scheduling is supported through pattern nesting (more sophisticated scheduling has to be programmed in Java). Patterns can be associated with JSP-like code templates (so-called scriptlets) in order to perform model-to-code transformation.

3.2.6 Other Model-To-Model Approaches

At least two more approaches should be mentioned for completeness: the transformation framework defined in the OMG's Common Warehouse Metamodel (CWM) Specification [CWM] and transformation implemented using XSLT [XSLT].

The CWM transformation framework provides a mechanism for linking source and target elements, but the derivation of the target elements has to be implemented in some concrete language, which is not prescribed by CWM. Effectively, CWM gives a general model, but no actual mechanism to implement model transformations.

Since models can be serialized as XML using the XML Metadata Interchange (XMI) [XMI], implementing model transformations using XSLT, which is a standard technology for transforming XML, seems very attractive. Unfortunately, this approach has severe scalability limitations. Manual implementation of model transformations in XSLT quickly leads to non-maintainable implementations because of the verbosity and poor readability of XMI and XSLT. A solution to overcome this problem is to generate the XSLT rules from some more declarative rule descriptions, as demonstrated in [PBG01, PZB00]. However, even this approach suffers from poor efficiency because the copying required by the pass-by-value semantics of XSLT and the poor compactness of XMI.

4 Discussion

Model transformation is a relatively young area. Although it is related to and builds upon the more established fields of program transformation and metaprogramming, the use of graphical modeling languages and the application of object-oriented metamodeling to language definition set a new context.

While there are satisfactory solutions for transforming models to text (such as template-based approaches), this is not the case for transforming models to models. Many new approaches to model-to-model transformation have been proposed over the last two years, but little experience is available to judge their effectiveness in practical applications. In this respect, we are still at the stage of exploring possibilities and eliciting requirements. Modeling tools available on the market are just starting to offer some model-to-model transformation capabilities, but these are still very limited and often ad hoc, i.e., without proper theoretical foundation. Most of these tools target the generation of EJB applications and the model transformations they offer were specifically developed to support that goal.

In this paper, we classified the existing model-to-model transformation approaches into direct manipulation approaches, relational approaches, graph-transformation-based approaches, structure-driven approaches, and hybrid approaches. In the remainder of this section, we offer some comments on the practical applicability of the different flavours of model transformation. These comments are based on our intuition and the application examples published together with the approaches. Because of the lack of experiments and practical experience, these comments are somewhat speculative, but we hope that they are still valuable as they may stimulate discussion and further evaluation.

- Direct manipulation is obviously the most low-level approach. It offers the user little or no support or guidance in implementing transformations. Basically all work has to be done by the user.
- The structure-driven category groups pragmatic approaches that were developed in the context of (and seem particularly well applicable to) certain kinds of applications such as generating EJB implementations and database schemas from UML models. These applications require a strong support for transforming models with a 1-to-1 and 1-to-n (and sometimes n-to-1) correspondence between source and target elements. Also, in this application context, there is typically no need for iteration (and in particular fixpointing) in scheduling, and the scheduling can be system-defined. It is unclear how well these approaches can support other kinds of applications.
- Graph-transformation-based approaches are inspired by heavily theoretical work in graph transformations. These approaches are powerful and declarative, but also the most complex ones. The complexity stems from the nondeterminism in scheduling and application strategy, which require careful consideration of termination of the transformation process and the rule application ordering (including the property of confluence). There is a large amount of theoretical work and some experience with research prototypes. However, experience with practical applications of these approaches is still limited. It remains to be seen how well the complexities of these approaches will be received in practice.
- Relational approaches seem to strike a well balance between flexibility and declarative expression. They provide flexible scheduling and good control of nondeterminism. Three of the five current QVT submission fit into this category ([CDI], [QVTP], and (partly) [AST+]).
- Hybrid approaches allow the user to mix and match different concepts and paradigms depending on the application. Practical approaches are very likely to have the hybrid character.

Evaluation of the different design options for a model transformation approach will require more experiments and practical experience. Establishing of a comprehensive collection of benchmark problems would be a valuable next step in that direction.

References

[AEH+96] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph Transformation for Specification and Programming. Technical Report 7/96, Universität Bremen, 1996, see http://citeseer.nj.nec.com/article/andries96graph.html

[AK02] D. H. Akehurst, S.Kent. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.): UML 2002 - The Unified Modeling Language 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings, LNCS 2460, 243-258, 2002.

[AKS03] A. Agrawal, G. Karsai and F. Shi. Graph Transformations on Domain-Specific Models. Under consideration for publication in the Journal on Software and Systems Modeling, 2003

[AND] AndroMDA 2.0.3, July 2003, http://www.andromda.org

[ANG] Frame Processor ANGIE, ??, http://www.d-s-t-g.com/neu/pages/pageseng/et/common/techn_angie_frmset.htm

[AS] ArcStyler 4.0, September 2004, http://www.arcstyler.com/

[AST+] Alcatel, Softeam, Thales, TNI-Valiosys and Codagen Corporation, Revised Submission for RFP-QVT, OMG Document ??, August 2003

[ATOM] ATOM^3 A Tool for Multi-Formalism and Meta-Modeling, ?? http://atom3.cs.mcgill.ca/

[B+M] b+m ArchitectureWare, Generator Framework, http://www.architectureware.de

[Bas97] P.G. Bassett, Framing Software Reuse: Lessons from the Real World. Prentice Hall, Inc., 1997

[BM03] P. Braun and F. Marschall. The Bi-directional Object-Oriented Transformation Language. Technical Report, Technische Universität München, TUM-I0307, May 2003

[CA] Codagen Architect 3.0, http://www.codagen.com/products/architect/default.htm

[CDI] CBOP, DSTC, and IBM. MOF Query/Views/Transformations, Revised Submission. OMG Document ad/03-08-03, http://www.dstc.edu.au/Research/Projects/Pegamento/publications/ad-03-08-03.pdf

[Cle01] C. Cleaveland. Program Generators with XML and Java. Prentice-Hall, 2001

[CWM] OMG, The Common Warehouse Model 1.1. OMG Document ??, March 2003, http://www.omg.org/cgi-bin/doc?formal/03-03-02

[Cza02] K. Czarnecki. Domain Engineering. Chapter in the Wiley Software Engineering Encyclopedia, Second Edition, John Marciniak, (Eds.), Wiley and Sons, Inc., February 2002, pp. 433-444

[FPL] Frame-Processing-Language, ??, http://sourceforge.net/projects/fpl

[Fra03] D. S. Frankel. Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons, 2003

[FUU] FUUT-je, hosted at the Eclipse Generative Model Transformer (GMT) project website, http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/download/index.html

[GLR+02] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood. Transformation: The Missing Link of MDA, In A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.): Graph Transformation: First International Conference (ICGT 2002), Barcelona, Spain, October 7-12, 2002. Proceedings. LNCS vol. 2505, Springer-Verlag, 2002, pp. 90 - 105

[IOPT] Interactive Objects and Project Technology, Revised Submission for RFP-QVT, OMG Document ??, August 2003

[JAM] Java Model Driven Architecture 0.2, May 2003, http://sourceforge.net/projects/jamda/

[JMI] Java Metadata Interface 1.0, July 2002, http://java.sun.com/products/jmi

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990

[MB03] F. Marschall and P. Braun. Model Transformations for the MDA with BOTL. In [Ren03], pp. 25-36

[MML] Meta ML, http://www.cse.ogi.edu/PacSoft/projects/metaml/

[MOF] OMG, Meta Object Facility (MOF), 1.4, OMG Document formal/02-04-03, April 2002, http://www.omg.org/cgi-bin/doc?formal/2002-04-03

[MOML] Meta Objective Caml, http://www.cs.rice.edu/~taha/MetaOCaml/

[OCL] OMG, The Object Constraint Language Specification 1.1, OMG Document ??, September 1997, http://www-itg.lbl.gov/DSDMeetings/obje_11.pdf

[OPTJ] OptimalJ 3.0, User's Guide, 2003, http://www.compuware.com/products/optimalj/default.htm

[PBG01] M. Peltier, J. Bézivin, and G. Guillaume. MTRANS: A general framework based on XSLT for model transformations. In WTUML'01, Proceedings of the Workshop on Transformations in UML, Genova, Italy, April 2001

[PZB00] M. Peltier, F. Ziserman, and J. Bézivin. On levels of model transformation. In XML Europe 2000, Paris, France, June 2000, Graphic Communications Association, pp. 1–17

[QVT] Object Management Group, MOF 2.0 Query / Views / Transformations RFP, OMG Document: ad/2002-04-10, revised on April 24, 2002

[QVTP] QVT-Partners, Revised Submission for MOF 2.0 Query / Views / Transformations RFP, OMG Document ad/2003-08-08, August 2003, http://www.qvtp.org

[Ren03] A. Rensink (Ed.) Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, University of Twente, Enschede, The Netherlands, June 26-27, 2003, CTIT Technical Report TR–CTIT–03–27, University of Twente, 2003, http://trese.cs.utwente.nl/mdafa2003

[STR] Strategies for Program Transformation, http://www.stratego-language.org

[UML] The Unified Modeling Language 1.5, March 2003, http://www.omg.org/cgi-bin/doc?formal/03-03-01

[VELO] Velocity 1.3.1, The Apache Jakarta Project, March 2003, http://jakarta.apache.org/velocity/

[VVP02] D. Varro, G. Varro and A. Pataricza. Designing the automatic transformation of visual languages. In Journal of Science of Computer Programming, vol. ??, no. ??, Elsevier, 2002, pp. ??-??

[Wil03] E. D. Willink. UMLX: A graphical transformation language for MDA. In [Ren03], pp. 13-24

[XD] XDoclet - Attribute Oriented Programming, http://xdoclet.sourceforge.net/

[XDE] Rational XDE, http://www.rational.com/products/xde

[XMI] OMG, OMG-XML Metadata Interchange (XMI) Specification, v1.2, OMG Document formal/02-01-01, http://www.omg.org/cgi-bin/doc?formal/2002-01-01

[XP] XML Path Language Version 1.0, November 1999, http://www.w3.org/TR/xpath

[XSLT] W3C, XSL Transformations (XSLT) Version 1.0, November 1999, http://www.w3.org/TR/xslt

[XVCL] XML-based Variant Configuration Language, ??, http://fxvcl.sourceforge.net/