

Feature-based survey of model transformation approaches

K. Czarnecki
S. Helsen

Model transformations are touted to play a key role in Model Driven Development™. Although well-established standards for creating metamodels such as the Meta-Object Facility exist, there is currently no mature foundation for specifying transformations among models. We propose a framework for the classification of several existing and proposed model transformation approaches. The classification framework is given as a feature model that makes explicit the different design choices for model transformations. Based on our analysis of model transformation approaches, we propose a few major categories in which most approaches fit.

INTRODUCTION

Model-driven software development is centered on the use of models.¹ Models are system abstractions that allow developers and other stakeholders to effectively address concerns, such as answering a question about the system or effecting a change. Examples of model-driven approaches are Model Driven Architecture** (MDA**),^{2,3} Model-Integrated Computing (MIC),⁴ and Software Factories.⁵ Software Factories, with their focus on automating product development in a product-line context, can also be viewed as an instance of generative software development.⁶

Model transformations are touted to play a key role in Model Driven Development** (MDD**). Their intended applications include the following:

- Generating lower-level models, and eventually code, from higher-level models⁷
- Mapping and synchronizing among models at the same level or different levels of abstraction⁸

- Creating query-based views of a system^{9,10}
- Model evolution tasks such as model refactoring^{11,12}
- Reverse engineering of higher-level models from lower-level models or code.¹³

Considerable interest in model transformations has been generated by the standardization effort of the Object Management Group, Inc. (OMG**). In April 2002, the OMG issued a Request for Proposal (RFP) on Query/Views/Transformations (QVT),¹⁴ which led to the release of the final adopted QVT specification in November 2005.¹⁵ Driven by practical needs and the OMG's request, a large number of approaches to model transformation have been proposed over the last three years. However, as of

©Copyright 2006 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/06/\$5.00 © 2006 IBM

this writing, industrial-strength and mature model-to-model transformation systems are still not available, and the area of model transformation continues to be a subject of intense research. In this paper, we propose a feature model to compare different model transformation approaches and offer a survey and categorization of a number of existing approaches from four sources:

1. *Published in the literature*—VIATRA (Visual Automated model TRAnsformations) framework,^{16,17} Kent Model Transformation language,^{18,19} Tefkat,^{20,21} GReAT (Graph Rewriting and Transformation language²²), ATL (Atlas Transformation Language^{23,24}), UMLX,²⁵ ATOM3 (A Tool for Multi-formalism and Meta-Modeling²⁶), BOTL (Bidirectional Object-oriented Transformation Language^{27,28}), MOLA (Model transformation LAnguage²⁹), AGG (Attributed Graph Grammar system³⁰), AMW (Atlas Model-Weaver³¹), triple-graph grammars,³² MTL (Model Transformation Language³³), YATL (Yet Another Transformation Language³⁴), Kermet,³⁵ C-SAW (Constraint-Specification Aspect Weaver),³⁶ and MT Model Transformation Language.³⁷
2. *Described in the final adopted QVT specification*—The Core, Relations, and Operational languages.¹⁵ Older QVT submissions are also mentioned whenever appropriate.
3. *Implemented within open-source tools*—AndroMDA,³⁸ openArchitectureWare,³⁹ Fujaba (From UML** to Java** And Back Again⁴⁰), Jamda (JAVa Model Driven Architecture⁴¹), JET (Java Emitter Templates⁴²), FUUT-je,⁴³ and MTF (Model Transformation Framework⁴⁴), which is a freely available prototype.
4. *Implemented within commercial tools*—XMF-Mosaic,⁴⁵ OptimalJ**,⁴⁶ MetaEdit+**,^{47,48} ArcStyler,⁴⁹ and Codagen Architect.⁵⁰

The feature model makes explicit the possible design choices for a model transformation approach, which is the main contribution of this paper. We do not give detailed classification data for each individual approach mainly because these details are constantly changing. Instead, we give examples of approaches for each design choice. Furthermore, we propose a clustering of existing approaches into a few major categories that capture their main characteristics and design choices. We conclude with remarks on the practical applicability of the different categories.

WHAT IS MODEL TRANSFORMATION?

Transformation is a fundamental theme in computer science and software engineering. After all, computation can be viewed as data transformation.

Computing with basic data such as numeric values and with data structures such as lists and trees is at the heart of programming. Type systems in programming languages help ensure that operations are applied compatibly to the data. However, when the subject of a transformation approach is metadata, i.e., data representing software artifacts such as data schemas, programs, interfaces, and models, then we enter the realm of *metaprogramming*—writing programs called *metaprograms* that write or manipulate other programs. One of the key challenges in this realm is that metaprograms have to respect the rich semantics of the metadata upon which they operate. Similarly, model transformation is a form of metaprogramming and, thus, must face the same challenge.

Model transformation is closely related to program transformation.⁵¹ In fact, their boundaries are not clear-cut, and both approaches overlap. Their differences occur in the mindsets and traditions of their respective transformation communities, the subjects being transformed, and the sets of requirements being considered. Program transformation is a more mature field with a strong programming language tradition. On the other hand, model transformation is a relatively new field, essentially rooted in software engineering. Consequently, the transformation approaches found in both fields have quite different characteristics. While program transformation systems are typically based on mathematically oriented concepts such as term rewriting, attribute grammars, and functional programming, model transformation systems usually adopt an object-oriented approach for representing and manipulating their subject models.

Because model transformations operate on models, we need to clarify what models are. A model is an abstraction of a system or its environment, or both. In software engineering, the term *model* is often used to refer to abstractions above program code, such as requirements and design specifications. Some authors in model-driven software development consider program code as models too. This view is consistent with the fact that program code is an abstraction of the underlying machine code produced by the compiler. Although being visual is not a defining characteristic of models, requirements

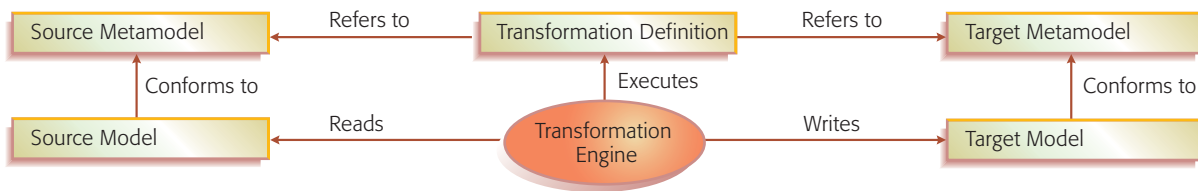


Figure 1
Basic concepts of model transformation

and design models are often more visual than programs. Models are frequently expressed in focused languages specialized for a particular class of software applications or a particular aspect of an application. For example, the Matlab** Simulink**/Stateflow** environment offers notations specialized for modeling control software, whereas interaction diagrams in Unified Modeling Language** (UML**) are focused on representing the interaction aspect of a wide range of systems. Highly specialized modeling languages are increasingly referred to as *domain-specific modeling languages*.

In general, model transformations involve models (in the sense of abstractions above program code) or models and programs. Because the concept of models is more general than the concept of program code, model transformations tend to operate on a more diverse set of artifacts than program transformations. Model transformation literature considers a broad range of software development artifacts as potential transformation subjects. These include UML models, interface specifications, data schemas, component descriptors, and program code. The varied nature of models further invites specialized transformation approaches that are geared to transforming particular kinds of models. For example, as explained later in the discussion section, most model transformation approaches based on graph transformations are better suited for transforming UML models than program code. However, there is no fundamental reason why program transformation systems could not be applied to the same artifacts as model transformations. In fact, transformational software development,⁵² which involves the automated refinement of high-level specifications into implementations, is an old and familiar theme in the area of program transformation.

In summation, perhaps the most important distinction between the current approaches to program transformation and model transformation is that the

latter has been targeted for a particular set of requirements that include the representation of models using an object-oriented paradigm, the traceability among models at different levels of abstraction, the transformation mapping among multiple models (i.e., n -way transformations), and the multidirectionality of transformations. Although these requirements could also be the subject of program transformation approaches, they are typically not considered by program transformation systems.

EXAMPLES OF MODEL TRANSFORMATIONS

To make our discussion more concrete, we present two examples of model transformations: one that maps models to models and another that maps models to code.

Figure 1 gives an overview of the main concepts involved in model transformation. The figure shows the simple scenario of a transformation with one input (source) model and one output (target) model. Both models conform to their respective metamodels. A metamodel typically defines the abstract syntax of a modeling notation. A transformation is defined with respect to the metamodels. The definition is executed on concrete models by a transformation engine. In general, a transformation may have multiple source and target models. Furthermore, the source and target metamodels may be the same in some situations.

Sample metamodels and models

Figures 2A and *2B* show sample metamodels expressed as UML class diagrams. *Figure 2A* gives a simplified metamodel for class models that includes the abstract concept of classifiers, which comprises classes and primitive data types. Packages contain classes, and classes contain attributes. All model elements have names, and classes can be marked as persistent. *Figure 2B* shows a simple metamodel for defining relational database schemas for a relational database management system (RDBMS). A schema

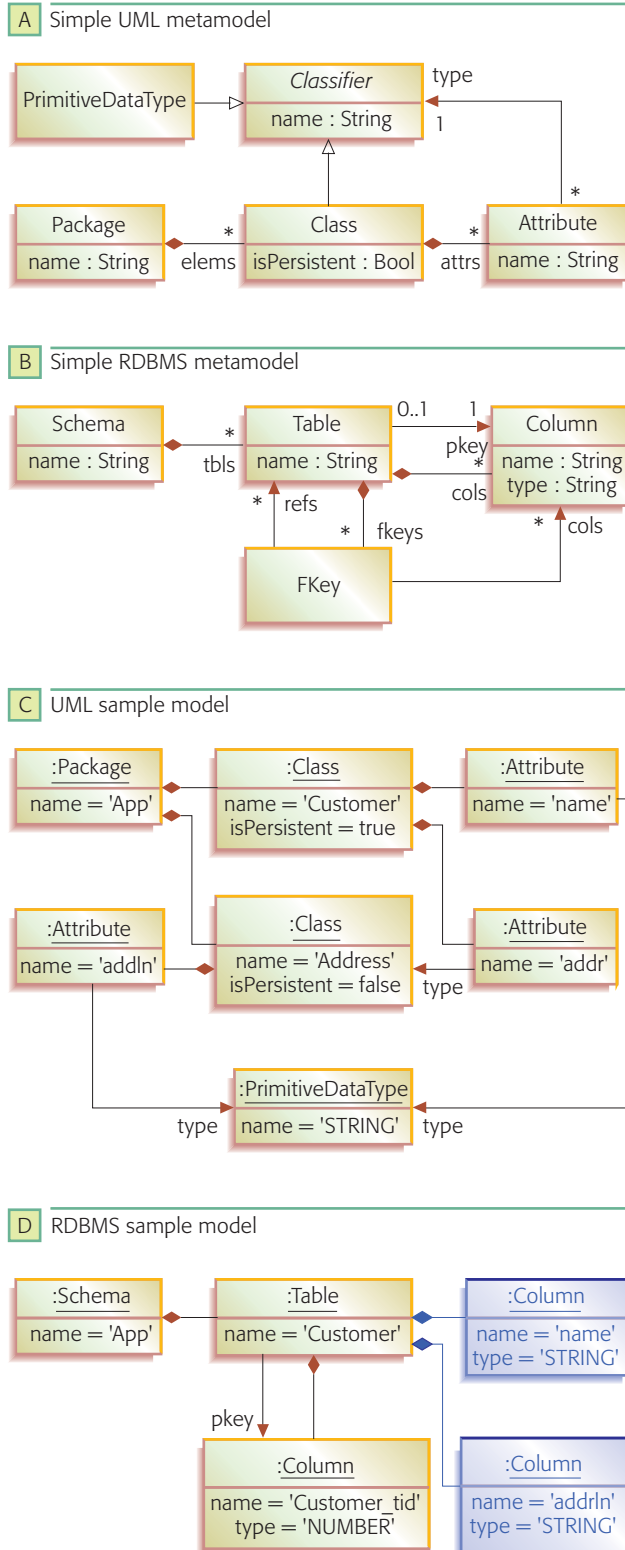


Figure 2
UML-to-RDBMS example and sample models: (A) Simple UML metamodel, (B) Simple RDBMS metamodel, (C) UML sample model, and (D) RDBMS sample model

contains tables, and tables contain columns. The column type is represented as a string. Every table has one primary-key column, which is pointed to by *pkey*. Additionally, the concept of foreign keys is modeled by *FKey*, which relates foreign-key columns to tables.

Sample instances of the metamodels using the UML object diagram notation are shown in **Figures 2C** and **2D**. The instance in Figure 2C represents a class model with one package, *App*, containing two classes, *Customer* and *Address*. *Customer* is persistent, and *Address* is not. Figure 2D shows an instance of the schema metamodel. The instance represents a schema that can be used to make *Customer* objects persistent.

UML-to-schema transformation

As a first example, we consider transforming class models into schema models described in the previous section. Such a transformation needs to realize the following three mappings:

1. *Package-to-schema*: Every package in the class model should be mapped to a schema with the same name as the package.
2. *Class-to-table*: Every persistent class should be mapped to a table with the same name as the class. Furthermore, the table should have a primary-key column with the type *NUMBER* and the name being the class name with *_tid* appended.
3. *Attribute-to-column*—The class attributes have to be appropriately mapped to columns, and some columns may need to be related to other tables by foreign key definitions. For simplicity, the attribute mapping is not further considered in this paper.

The above transformation would map the class model in Figure 2C to the schema model in Figure 2D. The part of the result in Figure 2D shown in green (lefthand and middle boxes) is handled by the first two mappings. The blue part (righthand boxes) corresponds to the result of the attribute-to-column mapping.

Transformations expressed in QVT Relations language

Example 1 shows how this transformation can be expressed using the QVT Relations language, which is a declarative language for model-to-model transformations. The language has both a textual and a graphical representation, but only the textual representation is shown here. The transformation

declaration specifies two parameters for holding the models involved in the transformation. The parameters are typed over the appropriate metamodels. The execution direction is not fixed at transformation definition time, which means that both `uml` and `rdbms` could be source and target models and vice versa. The user specifies the direction in which the transformation has to be executed only upon invoking the transformation.

Example 1

```
transformation umlRdbms {
  uml : SimpleUML, rdbms : SimpleRDBMS {
    key Table (name, schema);
    key Column (name, table);

    top relation PackageToSchema {
      domain uml p:Package {name = pn}
      domain rdbms s:Schema {name = pn}
    }

    top relation ClassToTable {
      domain uml c:Class {
        package = p:Package {},
        isPersistent = true,
        name = cn
      }
      domain rdbms t:Table {
        schema = s:Schema {},
        name = cn,
        cols = cl:Column {
          name = cn + '_tid',
          type = 'NUMBER',
          pkey = cl
        }
      }
      when {
        PackageToSchema (p, s);
      }
      where {
        AttributeToColumn (c, t);
      }
    }
  }

  relation AttributeToColumn {
    ...
  }
  ...
}
```

Each mapping is represented as a *relation*. A relation has as many *domain declarations* as there are models involved in the transformation. A domain is

bound to a model (e.g., `uml`) and declares a *pattern*, which will be bound with elements from the model to which the domain is bound. Such patterns consist of a variable and a type declaration, which itself may specify some of the properties of that type. When the transformation is executed, the relations are verified and, if necessary, enforced by manipulating the target model. If the target model is empty, its content is freshly created; otherwise, the existing content is updated.

A relation may specify a condition under which it applies by using a *when clause*. The *where clause* specifies additional constraints among the involved elements, which may need to be enforced. The *key definitions* are used by the transformation engine to identify objects that need to be updated during a transformation execution. There is much more to say about the execution semantics of QVT Relations, and the interested reader is invited to explore the QVT specification document.¹⁵

UML-to-Java transformation

In this example, we consider the generation of Java code from class models conforming to the meta-model in Figure 2A. In particular, a Java class with the appropriate attribute definitions and getters and setters should be generated for each class in the class model. Example 2 shows the desired output for the input model from Figure 2C.

Example 2

```
public class Customer {
  private String name;
  private Address addr;

  public void setName ( String name ) {
    this.name = name;
  }

  public String getName () {
    return this.name;
  }

  public void setAddr ( String addr ) {
    this.addr = addr;
  }

  public String getAddr () {
    return this.addr;
  }
}
```

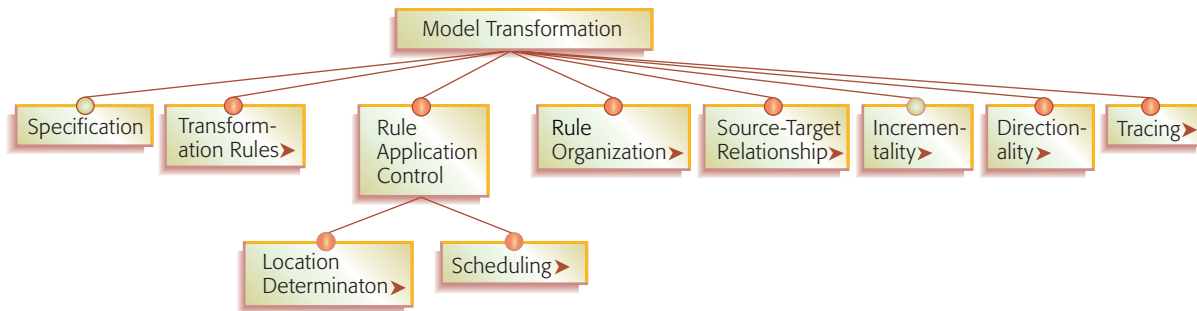



Figure 3
Top-level feature diagram

The code can conveniently be generated using a textual template approach, such as the openArchitectureWare template language demonstrated in Example 3. A template can be thought of as the target text with holes for variable parts. The holes contain metacode which is run at template instantiation time to compute the variable parts. The metacode in Example 3 is underlined. It has facilities to iterate over the elements of the input model (FOREACH), access the properties of the elements, and call other templates (EXPAND).

Example 3

```

<<DEFINE Root FOR Class>>
  public class <<name>> {
    <<FOREACH attrs AS a>>
      private <<a.type.name>> <<a.name>>;
    <<ENDFOREACH>>
    <<EXPAND AccessorMethods FOREACH attribute>>
  }
<<ENDDEFINE>>

<<DEFINE AccessorMethods FOR Attribute>>
  public <<type.name>> get<<name.toFirstUpper>>() {
    return this.<<name>>;
  }
  public void set<<name.toFirstUpper>>(
    <<type.name>> <<name>> ) {
    this.<<name>> = <<name>>
  }
<<ENDDEFINE>>

```

FEATURES OF MODEL TRANSFORMATION APPROACHES

This section presents the results 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 given domain.⁵³ We document our results using feature diagrams.^{54,55} Essentially, a feature diagram is a hierarchy of common and variable features characterizing the set of instances of a concept. In our case, the features provide a terminology and representation of the design choices for model transformation approaches. We do not aim for this terminology to be normative. Unfortunately, the relatively new area of model transformation has many overloaded terms, and many of the terms we use in our terminology are often used with different meanings in the original descriptions of the different approaches. Consequently, we provide the definitions of the terms as we use them. Furthermore, we expect the terminology to evolve as our understanding of model transformation matures. Our main goal is to show the vast range of available choices as represented by the current approaches.

Figure 3 shows the top-level feature diagram, where each subnode represents a major point of variation. The fragment of the cardinality-based feature modeling notation^{56,57} used in this paper is further explained in **Table 1**. Note that our feature diagrams treat model-to-model and model-to-text approaches uniformly. We will distinguish between these categories later in the “Major Categories” section. The description of the top-level features in **Figure 3** follows.

- *Specification*—Some transformation approaches provide a dedicated specification mechanism, such as preconditions and postconditions expressed in Object Constraint Language (OCL).⁵⁸ A particular transformation specification may represent a function between source and target

models and be executable; however, in general, specifications describe relations and are not executable. The QVT-Partners⁵⁹ submission distinguished between relations as potentially non-executable specifications of transformations and their executable implementations. The QVT specification¹⁵ still keeps this distinction, although the Relations language is now meant to be used primarily for expressing executable transformations.

- **Transformation rules**—In this paper, transformation rules are understood as a broad term describing the smallest units of transformation. Rewrite rules with a lefthand side (LHS) and a righthand side (RHS) are obvious examples of transformation rules; however, we also consider a function or a procedure implementing some transformation step as a transformation rule. In fact, the boundary between rules and functions is not so clear-cut; for example, function definitions in modern functional languages such as Haskell resemble rules with patterns on the left and expressions on the right. Templates can be considered as a degenerate form of rules, as discussed later in the “Template-Based Approaches” section.
- **Rule application control**—This has two aspects: *location determination* and *scheduling*. Location determination is the strategy for determining the model locations to which transformation rules are applied. Scheduling determines the order in which transformation rules are executed. Although control mechanisms usually address both aspects at the same time, for presentation purposes, we discuss them separately.
- **Rule organization**—This comprises general structuring issues, such as modularization and reuse mechanisms.
- **Source-target relationship**—This is concerned with issues such as whether source and target are one and the same model or two different models.
- **Incrementality**—This refers to the ability to update existing target models based on changes in the source models.
- **Directionality**—This describes whether a transformation can be executed in only one direction (unidirectional transformation) or multiple directions (multidirectional transformation).
- **Tracing**—This is concerned with the mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements.

Table 1 Symbols used in cardinality-based feature modeling

Symbol	Explanation
	Solitary feature with cardinality [1..1], i.e., <i>mandatory</i> feature
	Solitary feature with cardinality [0..1], i.e., <i>optional</i> feature
	Solitary feature with cardinality [n..m], $n \geq 0 \wedge m \geq n \wedge m > 1$, i.e., <i>mandatory clonable</i> feature
	Grouped feature
	Reference to feature model <i>F</i>
	<i>xor</i> -group
	<i>or</i> -group

Each of the following subsections elaborates on one major area of variation represented as a *reference* in Figure 3 by giving its feature diagram, describing the different choices, and providing examples of approaches supporting a given feature. The diagrams remain at a certain level of detail to fit the available space; however, each feature could be further analyzed uncovering additional subfeatures. Also, the feature groups in the presented diagrams usually express typical rather than all possible feature combinations. For example, different language paradigms (see Figure 5B later) are organized into an *xor*-group rather than an *or*-group (Table 1). Hybrid approaches may always provide any combinations of these features, which would correspond to an *or*-group.

Transformation rules

The features of transformation rules are given in **Figure 4A**. Their descriptions follow.

Domains

A domain is the part of a rule responsible for accessing one of the models on which the rule operates. Rules usually have a source and a target domain, but they may also involve more than two domains. Transformations involving *n* domains are sometimes referred to as *n-way transformations*. Examples are model merging or model weaving,³¹ which are transformations with more than one input domain. In general, a set of domains can also be seen as one large composite domain; however, it is

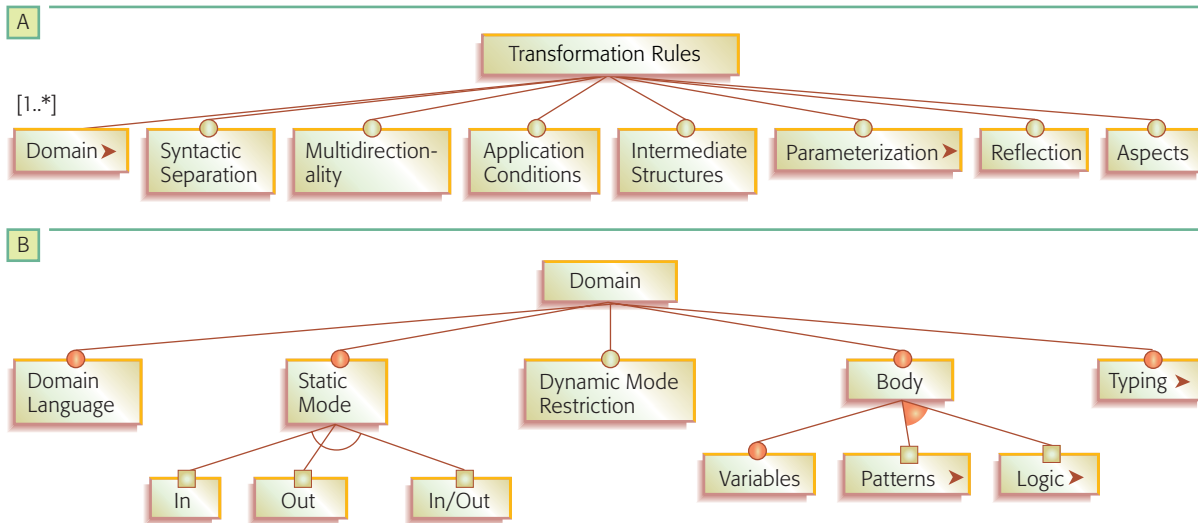


Figure 4
Features of transformation rules: (A) rules and (B) domains

useful to distinguish among individual domains when writing transformations.

Domains can have different forms. In QVT Relations, a domain is a distinguished typed variable with an associated pattern that can be matched in a model of a given model type (Example 1). In a rewrite rule, each side of the rule represents a domain. In an implementation of a rule as an imperative procedure, a domain corresponds to a parameter and the code that navigates or creates model elements by using the parameter as an entry point. Furthermore, a rule may combine domains of different forms. For example, the source domain of the templates in Example 3 is captured by the metacode, whereas the target domain has the form of string patterns.

The features of a domain are shown in **Figure 4B** and described in the following subsections:

Domain languages. A domain has an associated *domain language* specification that describes the possible structures of the models for that domain. In the context of MDA, that specification has the form of a metamodel expressed in the Meta Object Facility (MOF**).⁶⁰ Transformations with source and target domains conforming to a single metamodel are referred to as *endogenous* or *rephrasings*, whereas transformations with different source and target metamodels are referred to as *exogenous* or *translations*.^{61,62}

Static modes. Similar to the parameters of a procedure, domains have explicitly declared or implicitly assumed *static modes*, such as *in*, *out*, or *in/out*. Classical unidirectional rewrite rules with an LHS and RHS can be thought of as having an in-domain (source) and an out-domain (target), or a single in/out-domain for in-place transformations. Multidirectional rules, such as in MTF, assume all domains to be in/out.

Dynamic mode restriction. Some approaches allow *dynamic mode restriction*—restricting the static modes at execution time. For example, MTF allows marking any of the participating *in/out*-domains as read-only, that is, restricting them to *in* for a particular execution of a transformation. Essentially, such restrictions define the execution direction.

Body. There are three subcategories under Body, variables, patterns, and logic:

- *Variables* may 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 models being transformed (e.g., Java variables in transformed Java programs).
- *Patterns* are model fragments with zero or more variables. Sometimes, such as in the case of templates, patterns can have not only variables embedded in their body, but also expressions and statements of the metalanguage. Depending on the

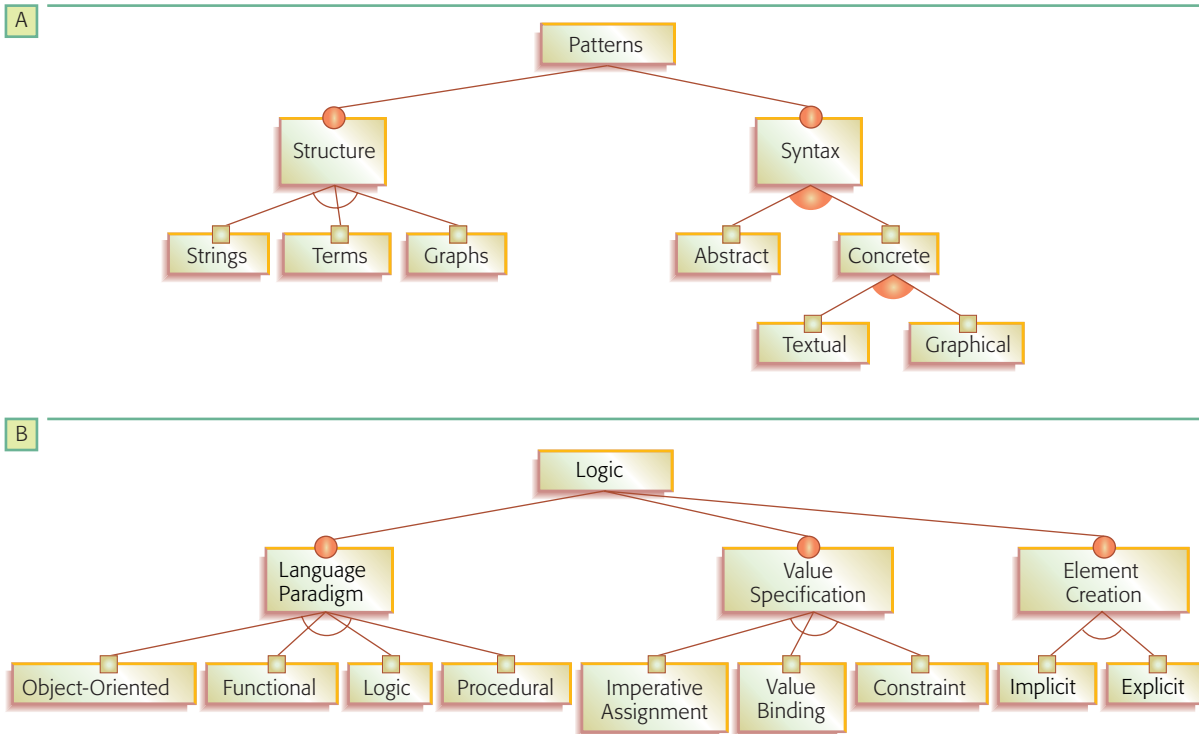


Figure 5
Features of the body of a domain: (A) patterns and (B) logic

internal representation of the models being transformed, we can have string, term, or graph patterns (*Figure 5A*). String patterns are used in textual templates, as discussed later in the “Template-Based Approaches” section. Model-to-model transformations usually apply term or graph patterns. Patterns can be represented using the abstract or concrete syntax of the corresponding source or target model language, and the syntax can be textual or graphical.

- *Logic* expresses computations and constraints on model elements (*Figure 5B*). Logic may follow different programming paradigms such as object-oriented or functional and be nonexecutable or executable. Nonexecutable logic is used to specify relationships among models. Executable logic can take a declarative or imperative form. Examples of the declarative form include OCL queries to retrieve elements from the source model and the implicit creation of target elements through constraints, as in the QVT Relations and Core languages. Imperative logic often has the form of program code calling repository application programming interfaces (APIs) to manipulate models directly. For instance, the Java Metadata Interface

(JMI)⁶³ provides a Java API to access models in a MOF repository. Imperative code uses imperative assignment, whereas declarative approaches may bind values to variables, as in functional programming, or specify values through constraints.

Typing. The *typing* of variables, logic, and patterns can be untyped, syntactically typed, or semantically typed (*Figure 6*). Textual templates are examples of untyped patterns (see the “Template-Based Approaches” section). 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, such as well-formedness rules (static semantics) and behavioral properties (dynamic semantics). A type system for a transformation language could statically ensure for a transformation that the models produced by the transformation will satisfy a certain set of syntactic and semantic properties, provided the input models satisfy some syntactic and semantic properties.

Syntactic separation

Some approaches offer *syntactic separation* (see *Figure 4A*). They clearly separate the parts of a rule

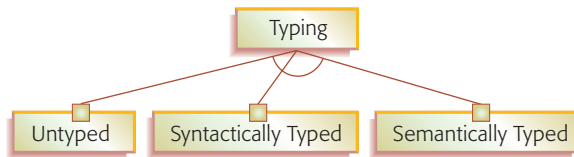


Figure 6
Typing

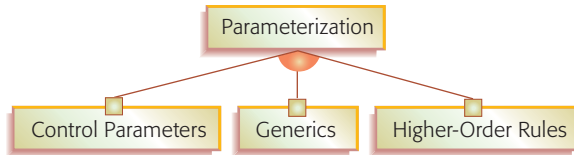


Figure 7
Parameterization

operating on one model from the parts operating on other models. For example, classical rewrite rules have an LHS operating on the source model and an RHS operating on the target model. In other approaches, such as a rule implemented as a Java program, there might not be any such syntactic distinction.

Multidirectionality

Multidirectionality refers to the ability to execute a rule in different directions (see Figure 4A). Rules supporting multidirectionality are usually defined over in/out-domains. Multidirectional rules are available in MTF and QVT Relations.

Application condition

Transformation rules in some approaches may have an *application condition* (see Figure 4A) that must be true in order for the rule to be executed. An example is the *when-clause* in QVT Relations (Example 1).

Intermediate structure

The execution of a rule may require the creation of some additional *intermediate structures* (see Figure 4A) which are not part of the models being transformed. These structures are often temporary and require their own metamodel. A particular example of intermediate structures are traceability links. In contrast to other intermediate structures, traceability links are usually persisted. Even if traceability links are not persisted, some approaches, such as AGG and VIATRA, rely on them to prevent

multiple “firings” of a rule for the same input element.

Parameterization

The simplest kind of *parameterization* is the use of control parameters that allow passing values as control flags (Figure 7). *Control parameters* are useful for implementing policies. For example, a transformation from class models to relational schemas could have a control parameter specifying which of the alternative patterns of object-relational mapping should be used in a given execution.⁷ *Generics* allow passing data types, including model element types, as parameters. Generics can help make transformation rules more reusable. Generic transformations have been described by Varró and Pataricza.¹⁷ Finally, *higher-order rules* take other rules as parameters and may provide even higher levels of reuse and abstraction. Stratego⁶⁴ is an example of a term rewriting language for program transformation supporting higher-order rules. We are currently not aware of any model transformation approaches with a first class support for higher-order rules.

Reflection and aspects

Some authors advocate the support for *reflection* and *aspects* (Figure 4) in transformation languages. Reflection is supported in ATL by allowing reflective access to transformation rules during the execution of transformations. An aspect-oriented extension of MTL was proposed by Silaghi et al.⁶⁵ Reflection and aspects can be used to express concerns that crosscut several rules, such as custom traceability management policies.⁶⁶

Rule application control: Location determination

A rule needs to be applied to a specific location within its source scope. As there may be more than one match for a rule within a given source scope, we need a strategy for determining the application locations (Figure 8A). The strategy could be *deterministic*, *nondeterministic*, or *interactive*. For example, a deterministic strategy could exploit some standard traversal strategy (such as depth first) over the containment hierarchy in the source. Stratego⁶⁴ is an example of a term rewriting language with a rich mechanism for expressing traversal in tree structures. Examples of nondeterministic strategies include *one-point* application, where a rule is applied to one nondeterministically selected location, and *concurrent* application, where one rule is

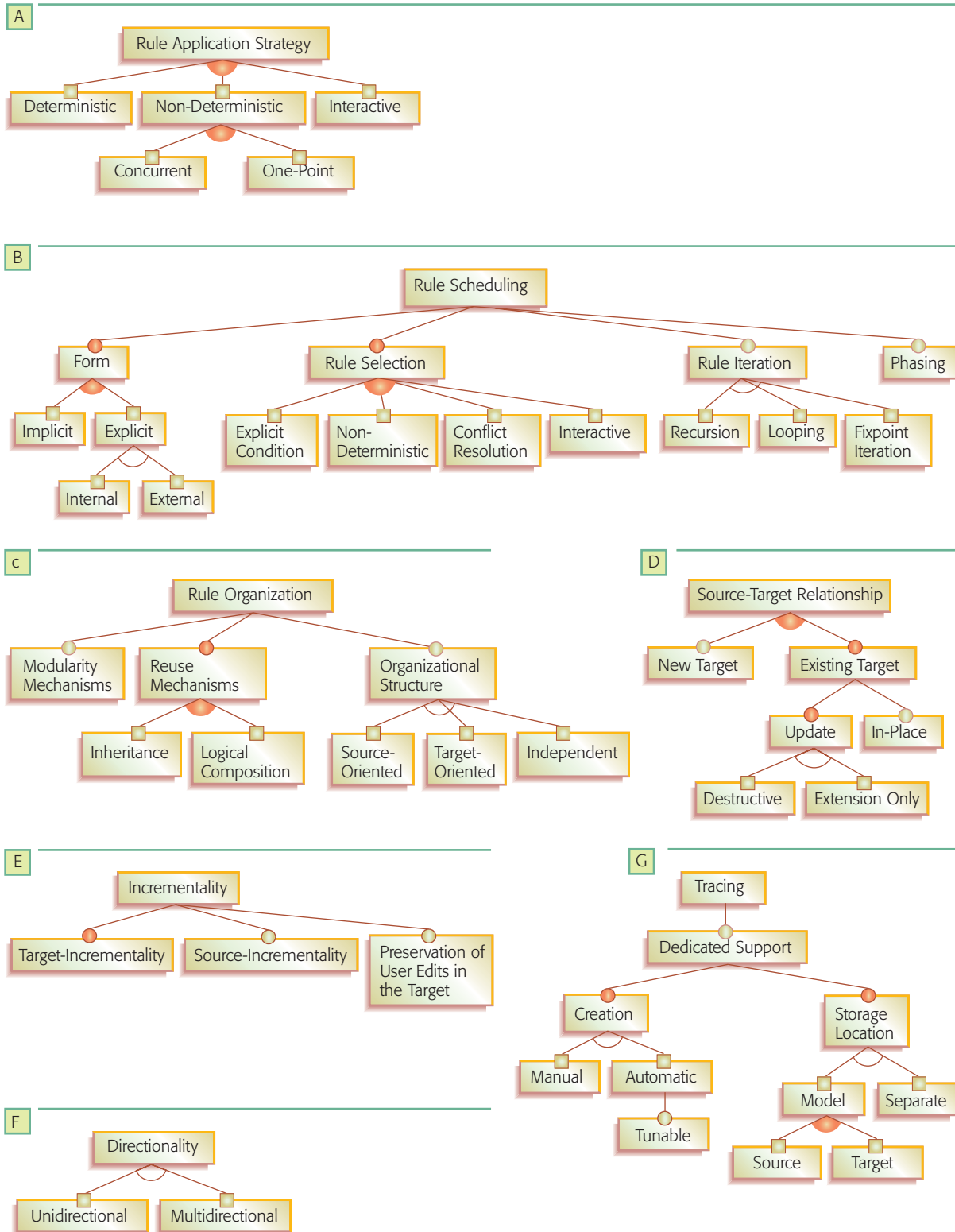


Figure 8

Model transformation approach features: (A) location determination, (B) rule scheduling, (C) rule organization, (D) source-target relationship, (E) incrementality, (F) directionality, and (G) tracing

applied concurrently to all matching locations in the source. Concurrent application is supported in AToM3, AGG, and VIATRA. AGG offers *critical pair analysis* to verify for a set of rules that there will be no rules competing for the same source location. Some tools, such as AToM3, allow the user to determine the location for rule application interactively.

The target location for a rule is usually deterministic. In an approach with separate source and target models, traceability links can be used to determine the target: A rule may follow the traceability link to a target element that was created by another rule and use the element as its own target. In the case of in-place update, the source location becomes the target location, although traceability links can also be used (as later illustrated in Figure 10).

Rule application control: Rule scheduling

Scheduling mechanisms determine the order in which individual rules are applied. Scheduling mechanisms can vary in four main areas (Figure 8B).

1. *Form*—The scheduling aspect can be expressed implicitly or explicitly. *Implicit scheduling* implies that the user has no explicit control over the scheduling algorithm defined by the tool, as in BOTL. The only way a user can influence the system-defined scheduling algorithm is by designing the patterns and logic of the rules to ensure 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 can be internal or external. In *external scheduling*, there is a clear separation between the rules and the scheduling logic. For example, VIATRA offers rule scheduling by an external finite state machine. In contrast, *internal scheduling* is a mechanism allowing a transformation rule to directly invoke other rules, as in ATL or the code template shown in Example 3.
2. *Rule selection*—Rules can be selected by an explicit condition, as in MOLA. Some approaches, such as BOTL, offer a nondeterministic choice. Alternatively, a conflict resolution mechanism based on priorities can be provided. Interactive rule selection is also possible. Both

priorities and interactive selection are supported in AToM3.

3. *Rule iteration*—Rule iteration mechanisms include recursion, looping, and fixpoint iteration (i.e., repeated application until no changes are detected). For example, ATL supports recursion, MOLA has a looping construct, and VIATRA supports fixpoint iteration.
4. *Phasing*—The transformation process may be organized into several phases, with each phase having a specific purpose, and only certain rules can be invoked in a given phase. For example, structure-oriented approaches, such as OptimalJ and the QVT submission by Interactive Objects and partners,⁶⁷ 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 (see the “Structure-driven approaches” section).

Rule organization

Rule organization is concerned with composing and structuring multiple transformation rules. We consider three areas of variation in this context (Figure 8C):

1. *Modularity mechanisms*—Some approaches (e.g., QVT, ATL, MTL, and VIATRA) allow packaging rules into modules. A module can import another module to access its content.
2. *Reuse mechanisms*—Reuse mechanisms offer a way to define a rule based on one or more other rules. In general, scheduling mechanisms, such as calling one rule from another, can be used to define composite transformation rules. However, some approaches offer dedicated reuse mechanisms, such as inheritance between rules (e.g., rule inheritance,⁶⁸ derivation,⁶⁷ extension,⁶⁹ and specialization⁵⁹), inheritance between modules (e.g., unit inheritance⁶⁸), and logical composition.⁵⁹
3. *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 the QVT submission by Interactive Objects and partners.⁶⁷ 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).

Source-target relationship

Some approaches, such as ATL, mandate the creation of a new target model that has to be separate from the source (*Figure 8D*). However, in-place transformation can be simulated in ATL through an automatic copy mechanism. In some other approaches, such as VIATRA and AGG, source and target are always the same model; that is, they only support in-place update. Yet other approaches, for example, QVT Relations and MTF, allow creating a new model or updating an existing one. QVT Relations also support in-place update. Furthermore, an approach could allow a destructive update of the existing target or an update by extension only, that is, where existing model elements cannot be removed. Approaches using nondeterministic selection and fixpoint iteration scheduling (see “Rule Scheduling” section earlier) may restrict in-place update to extension in order to ensure termination. Alternatively, transformation rules may be organized into an expansion phase followed by a contraction phase, which is often done in graph transformation systems such as AGG.

Incrementality

Incrementality involves three different features (*Figure 8E*):

1. *Target incrementality*—The basic feature of all incremental transformations is target-incrementality, that is, the ability to update existing target models based on changes in the source models. This basic feature is also referred to as *change propagation* in the QVT final adopted specification.¹⁵ Obviously, target incrementality corresponds to the feature update in *Figure 8D*, but it is now seen from the change-propagation perspective. A target-incremental transformation creates the target models if they are missing on the first execution. A subsequent execution with the same source models as in the previous execution has to detect that the needed target elements already exist. This detection can be achieved, for example, by using traceability links. When any of the source models are modified and

the transformation is executed again, the necessary changes to the target are determined and applied. At the same time, the target elements that can be preserved are preserved.

2. *Source incrementality*—Source incrementality is about minimizing the amount of source that needs to be reexamined by a transformation when the source is changed. Source incrementality corresponds to incremental compilation: A change impact analysis determines the total set of source modules that need to be recompiled based on the list of source modules that were changed. Source incrementality is useful for working with large source models.
3. *Preservation of user edits in the target*—Practical scenarios in the context of model synchronization require the ability to rerun a transformation on an existing user-modified target to resynchronize the target with a changed source while preserving the user edits in the target. The dimensions of model synchronization, such as the degree of preservation of user-provided input in the target models, the degree of automation, and the frequency of triggering, are discussed elsewhere.⁵⁷

Directionality

Transformations may be *unidirectional* or *multidirectional* (*Figure 8F*). Unidirectional transformations can be executed in one direction only, in which case a target model is computed (or updated) based on a source model. Multidirectional transformations can be executed in multiple directions, which is particularly useful in the context of model synchronization. Multidirectional transformations can be achieved using multidirectional rules or by defining several separate complementary unidirectional rules, one for each direction.

Transformation rules usually have a functional character: Given some input in the source model, they produce a concrete result in the target model. A declarative rule (i.e., one that only uses declarative logic or patterns) can often be applied in the inverse direction, too. However, as 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 (this could theoretically be infinite), or just establish a part of the result in a concrete way (because the part is the same for all solutions) and use variables, defaults, or values already present in the result for the rest of it. The invertibility of a transformation

depends not only on the invertibility of the transformation rules, but also on the invertibility of the scheduling logic. In general, inverting a set of rules may fail to produce any result due to nontermination.

Tracing

Tracing can be understood as the runtime footprint of transformation execution (**Figure 8G**). *Traceability links* are a common form of trace information in model transformation, connecting source and target elements, which are essentially instances of the mapping between the source and target domains. Traceability links can be established by recoding the transformation rule and the source elements that were involved in creating a given target element. Trace information can be useful in performing impact analysis (i.e., analyzing how changing one model would affect other related models), determining the target of a transformation as in model synchronization, model-based debugging (i.e., mapping the stepwise execution of an implementation back to its high-level model), and in debugging model transformations themselves.

Some approaches, such as QVT, ATL, and Tefkat, provide dedicated support for tracing. Even without dedicated support, as in the case of AGG, VIATRA and GReAT, tracing information can always be created just as any other target elements. Some approaches with dedicated support, Tefkat for example, require developers to manually encode the creation of traceability links in the transformation rules, while other approaches, such as QVT and ATL, create traceability links automatically. In the case of automated support, the approach may still provide some control over what gets recorded. In general, we might want to control (1) the kind of information recorded (e.g., the links between source and target elements, the rules that created them, and a time stamp for the creation), (2) the abstraction level of the recorded information (e.g., links for top-level transformations only), and (3) the scope for which the information is recorded (e.g., tracing for particular rules or parts of the source only). Finally, there is the choice of location where the links are stored (e.g., in the source or target, or separately).

MAJOR CATEGORIES

At the top level, we distinguish between *model-to-text* and *model-to-model* transformation approaches. The distinction between the two categories is that, while a model-to-model transformation creates its

target as an instance of the target metamodel, the target of a model-to-text transformation is just strings. For completeness, we mention the concept of text-to-model transformation, but it essentially comprises parsing and reverse-engineering technologies, which are beyond the scope of this paper.

Model-to-text transformation corresponds to the concept of “pretty printing” in program transformation. Model-to-text approaches are useful for generating both code and noncode artifacts such as documents. In general, we can view transforming models to code as a special case of model-to-model transformations; we only need to provide a metamodel for the target programming language. However, for practical reasons of reusing existing compiler technology and for simplicity, code is often generated simply as text, which is then fed into a compiler. OMG issued an RFP for a MOF 2.0 Model-to-Text Transformation Language in April 2004,⁷⁰ which will eventually lead to a standard for mapping MOF-based models to text.

Model-to-text approaches

In the model-to-text category, we distinguish between visitor-based and template-based approaches.

Visitor-based approach

A very basic code-generation approach consists in providing some visitor mechanism to traverse the internal representation of a model and write text to a text stream. An example of this approach is Jamda—an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism (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.

Template-based approach

The majority of currently available MDA tools support template-based model-to-text generation (e.g., openArchitectureWare, JET, FUUT-je, CodaGen Architect, AndroMDA, ArcStyler, MetaEdit+, and OptimalJ). AndroMDA reuses existing open-source template-based generation technology: Velocity⁷¹ and XDoclet.⁷² An example of the template-based approach is shown in Example 3.

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. (For an introduction to template-based code generation, see Cleaveland.⁷³) According to our terminology, the LHS uses executable logic to access source, and the RHS combines untyped string patterns with executable logic for code selection and iterative expansion. Furthermore, there is no clear 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 template.

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 such as JMI, or it could be declarative queries, for example, in OCL or XPath.⁷⁴ The openArchitectureWare 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 the logic into user-defined operations of the source-model elements.

Compared with 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. Because 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 artifacts, including documentation.

A related technology is frame processing, which extends templates with more sophisticated adaptation and structuring mechanisms (Bassett's frames,⁷⁵ XVCL,⁷⁶ XFramer,⁷⁷ ANGIE^{**78}). To our knowledge, XFramer and ANGIE have been applied to generate code from models.

Model-to-model approaches

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

Direct manipulation approach

This category of approach offers an internal model representation and some APIs to manipulate it, such

as JMI. It is 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 usually have to implement transformation rules, scheduling, tracing, and other facilities, mostly from the beginning, in a programming language such as Java.

Structure-driven approach

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 the 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 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 can then be adapted 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 the QVT submission by Interactive Objects and Project Technology.⁶⁷ 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.

Operational approach

Approaches that are similar to direct manipulation but offer more dedicated support for model transformation are grouped in this category. A typical solution in this category is to extend the utilized metamodeling formalism with facilities for expressing computations. An example would be to extend a

query language such as OCL with imperative constructs. The combination of MOF with such extended executable OCL becomes a fully-fledged object-oriented programming system. Examples of systems in this category are QVT Operational mappings, XMF-Mosaic's executable MOF, MTL, C-SAW, and Kermeta. Specialized facilities such as tracing may be offered through dedicated libraries.

Example 4 shows our sample transformation from class models to schemas expressed in the QVT Operational language. In contrast to the QVT Relations solution from Example 1, the transformation declaration specifies the parameter modes; that is, the transformation is executed only in one direction from `uml` to `rdbsms`. The entry point for the execution is the function `main()`, which invokes the `packageToSchema` mapping on all packages and then the `attributeToColumn` mapping on all attributes contained in the input model `uml`. The mappings are defined by using an imperative extension of OCL. A mapping is defined as an operation on a model element. For example, `packageToSchema` is an operation of `Package` with `Schema` as its return type. The body of the mapping populates the properties of the return object, while `self` refers to the object on which the mapping was invoked. QVT Operations is a quite feature-rich language. The interested reader is invited to explore the QVT specification document.¹⁵

Example 4

```

transformation umlRdbms(
  in uml : SimpleUML,
  out rdbsms : SimpleRDBMS
);

main() {
  uml.objectsOfType(Package)->map packageToSchema();
  uml.objectsOfType(Attribute)->map attributeToColumn();
}

mapping Package::packageToSchema() : Schema {
  --population section for the schema
  name := self.name;
  tbls := self.elems->map classToTable();
}

mapping Class::classToTable() : Table
  when { self.isPersistent=true; } {

```

```

  name := self.name;
  key := object Column {
    name := self.name + '_tid';
    type := 'NUMBER';
  };
  cols := key;
}
mapping Attributes::attributeToColumn() : Column {
  ...
}
...

```

Template-based approach

Model templates are models with embedded meta-code that compute the variable parts of the resulting template instances. Model templates are usually expressed in the concrete syntax of the target language, which helps the developer to predict the result of template instantiation. The metacode can have the form of annotations on model elements. Typical annotations are conditions, iterations, and expressions, all being part of the metalanguage. An obvious choice for the expression language to be used in the metalanguage is OCL.

A concrete model-template approach is given by Czarnecki and Antkiewicz.⁷⁹ In that approach, a template of a UML model, such as a class or activity diagram, is created by annotating model elements with conditions or expressions represented as stereotypes. A very simple example is shown in **Figure 9**, which reuses the class model from **Figure 2C**. This time, however, the model is shown in its UML concrete syntax. The class `Address` and the `addr` attribute of `Customer` are annotated with the presence condition `addrFeature`. When the template is instantiated with `addrFeature` being true, the resulting model is the same as the template. If the condition is false, the annotated elements, which are blue in the figure, are removed.

Relational approach

This category groups declarative approaches in which the main concept is mathematical relations. In general, relational approaches can be seen as a form of constraint solving. Examples of relational approaches are QVT Relations, MTF, Kent Model Transformation Language, Tefkat, AMW, and mappings in XMF-Mosaic.

The basic idea is to specify the relations among source and target element types using constraints. In

its pure form, such a specification is nonexecutable (e.g., relations^{18,59} and mapping rules⁶⁸). However, declarative constraints can be given executable semantics, such as in 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. Gerber et al.²⁰ 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. An example of the relational approach is shown in Example 1.

All of the relational approaches are side-effect-free and, in contrast to the imperative direct manipulation approaches, create target elements implicitly. Relational approaches can naturally support multi-directional rules. They sometimes also provide backtracking. Most relational approaches require strict separation between source and target models; that is, they do not allow in-place update.

Graph-transformation-based approach

This category of model transformation approaches draws on the theoretical work on graph transformations. In particular, this category operates on typed, attributed, labeled graphs,⁸⁰ which can be thought of as formal representations of simplified class models. Examples include AGG, AToM3, VIATRA, GReAT, UMLX, BOTL, MOLA, and Fujaba.

Graph transformation rules have an LHS and an RHS graph pattern. The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place. The LHS often contains conditions in addition to the LHS pattern, for example, negative conditions. Some additional logic, for example, in string and numeric domains, is needed to compute target attribute values such as element names. GReAT offers an extended form of patterns with multiplicities on edges and nodes.

Graph patterns can be rendered in the concrete syntax of their respective source or target language (e.g., in VIATRA) or in the MOF abstract syntax (e.g., in BOTL and AGG). The advantage of the concrete syntax is that it is more familiar to developers working with a given modeling language than the abstract syntax. Also, for complex languages like UML, patterns in a concrete syntax tend

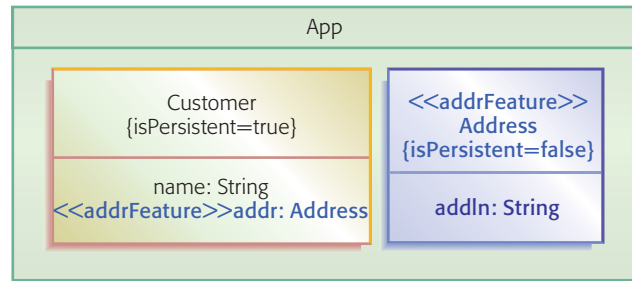


Figure 9
Example of a model template

to be much more concise than patterns in the corresponding abstract syntax (compare Figures 3C and 9 and also see the work by Marschall and Braun²⁸ for examples). On the other hand, it is easy to provide a default rendering for abstract syntax that will work for any metamodel, which is useful when no specialized concrete syntax is available.

AGG and AToM3 are systems directly implementing the theoretical approach to attributed graphs and transformations on such graphs. They have built-in fixpoint scheduling with nondeterministic rule selection and concurrent application to all matching locations, and they rely on implicit scheduling by the user. The transformation rules are unidirectional and in-place.

Figure 10 illustrates how the transformation from class models to schemas can be expressed in AGG. Only two rules are shown. The rule in Figure 10A maps packages to schemas. The mapping from classes to tables is given in Figure 10B. The mapping of attributes to columns is not shown. The RHS of an AGG rule contains a mixture of the new elements and elements from the LHS, as indicated by the indexes prefixing their names. When the LHS is matched, new elements are created. The implicit scheduling is achieved through correspondence objects connecting source and target elements (which are an example of intermediate structures) and negative conditions. For example, the package-to-schema rule matches packages and creates the corresponding schemas and the package-to-schema correspondence objects (i.e., instances of P2S). Each rule has a negative application condition, which is implicitly assumed to be its RHS. Because of the negative application condition, no additional schema objects will be created for a package that is already connected to a schema by a P2S object.

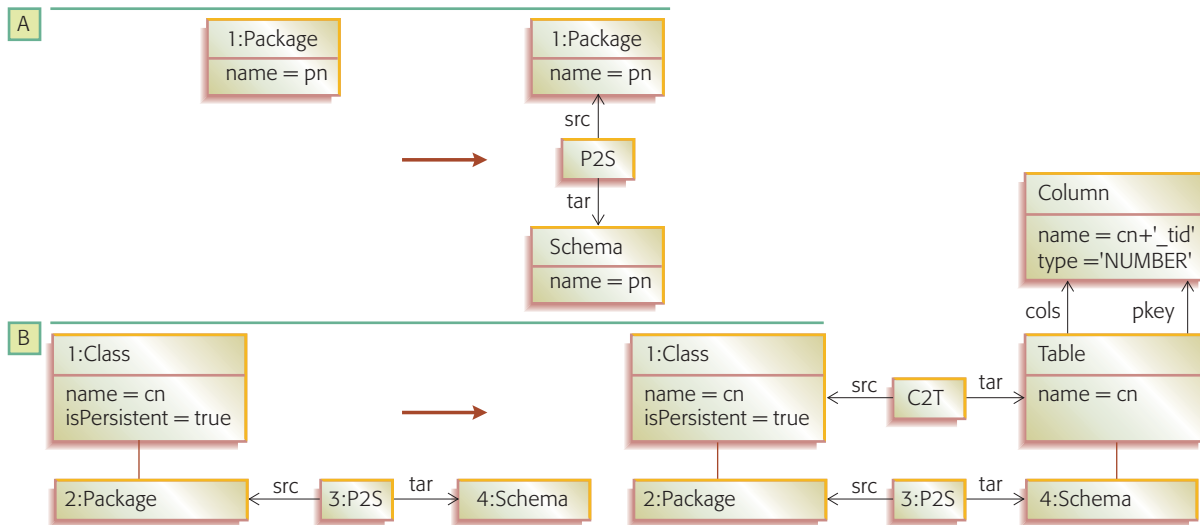


Figure 10
Graph transformation in AGG: (A) package-to-schema rule and (B) class-to-table rule

Systems such as VIATRA, GReAT, MOLA, and Fujaba extend the basic functionality of AGG and AToM3 by adding explicit scheduling. For example, VIATRA users can build state machines to schedule transformation rules. The explicit representation of scheduling in GReAT is a data-flow graph. MOLA and Fujaba use control-flow graphs for that purpose. The class-model-to-schema transformation expressed in MOLA is shown in **Figure 11**. Each enclosing rectangular box represents a looping construct. Boxes with rounded corners represent looping conditions. The elements to be matched are drawn using solid lines; dashed lines are used for the elements to be created. The top condition matches package objects. When a package object is matched, the corresponding schema is created and the body of the loop, which is another loop, is executed. The latter loop iterates over all classes in the package that was matched in the current iteration of the outer loop and creates the corresponding classes and primary-key columns. The final step is a call to `ProcessClassAttributes`, which is a subprogram mapping attributes to columns.

Relational-style, multidirectional approaches based on graph transformations are also possible. For example, Königs³² discusses using a transformation approach based on triple-graph grammars to simulate QVT Relations.

Hybrid approach

Hybrid approaches combine different techniques from the previous categories. The different approaches can be combined as separate components or, in a more fine-grained fashion, at the level of individual rules. QVT is an example of a hybrid approach with three separate components, namely Relations, Operational mappings, and Core. Examples of the fine-grained combination are ATL and YATL.

A transformation rule in ATL may be fully declarative, hybrid, or fully imperative. The LHS of a fully declarative rule (so-called source pattern) consists of a set of syntactically typed variables with an optional OCL constraint as a filter or navigation logic. The RHS of a fully declarative rule (so-called target pattern) contains a set of variables and some declarative logic to bind the values of the attributes in the target elements. In a hybrid rule, the source or target patterns are complemented with a block of imperative logic, which is run after the application of the target pattern. A fully imperative rule (so-called procedure) has a name, a set of formal parameters, and an imperative block, but no patterns. Rules are unidirectional and support rule inheritance.

Other approaches

Two more approaches are mentioned for completeness: transformation implemented using Extensible Stylesheet Language Transformation (XSLT⁸¹) and

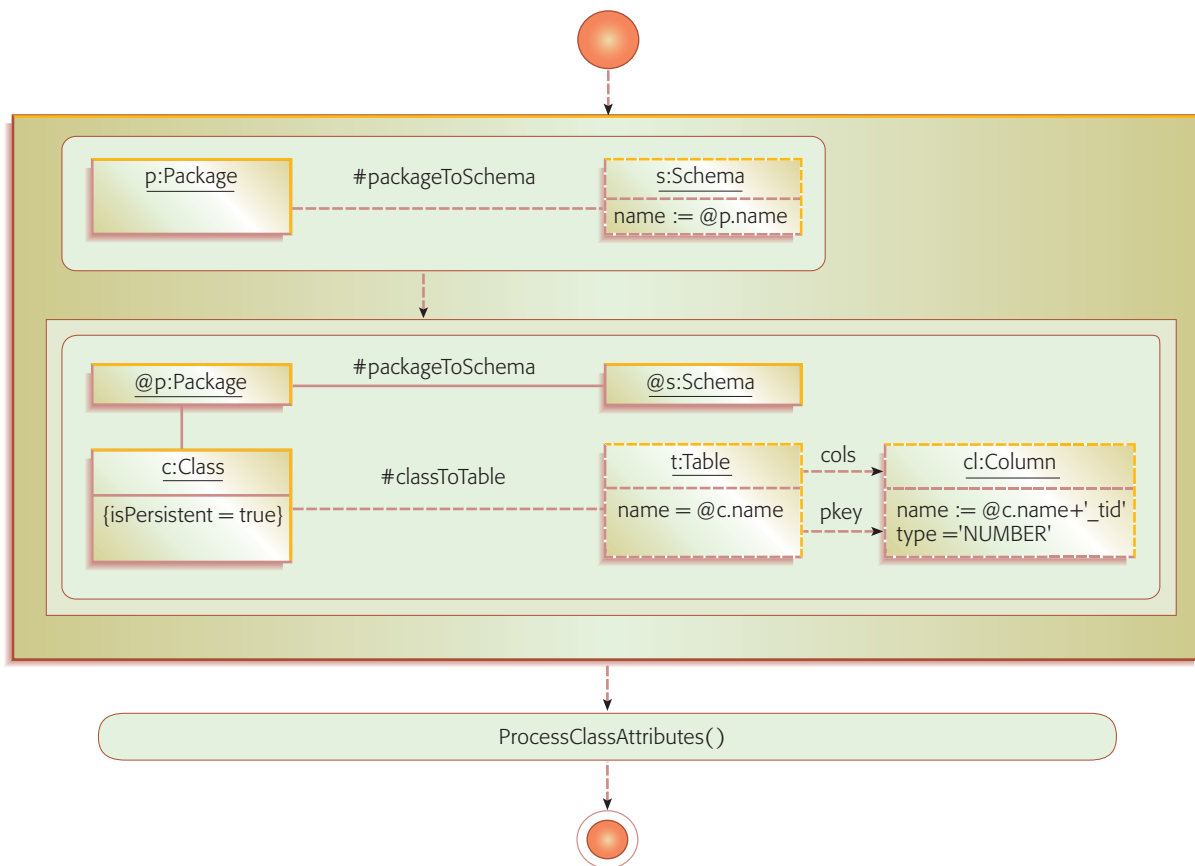


Figure 11
Graph transformation in MOLA

the application of metaprogramming to model transformation.

Because models can be serialized as Extensible Markup Language (XML) using the XML Metadata Interchange (XMI⁸²), implementing model transformations using XSLT, which is a standard technology for transforming XML, seems very attractive. Such an approach can be classified as term rewriting using a functional language. Unfortunately, the use of XMI and XSLT has 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 is to generate the XSLT rules from some more declarative rule descriptions, as demonstrated in the work by Peltier et al.^{83,84}; however, even this approach suffers from poor efficiency because of the copying required by the pass-by-value semantics of XSLT and the poor compactness of XMI.

A more promising direction in applying traditional metaprogramming techniques to model transformations has been proposed by Tratt.³⁷ His solution is a domain-specific language for model transformations embedded in a metaprogramming language.

DISCUSSION

In this section, we comment on the practical applicability of the different types of model transformation. These comments are based on our intuition and the application examples published together with the approaches. Because of the lack of controlled experiments and extensive practical experience, these comments are not fully validated, but we hope that they will stimulate discussion and further evaluation.

Direct manipulation is obviously the most low-level approach. In its basic form, it offers the user little or no support or guidance in implementing transformations. Essentially, all work has to be done by the user. The approach can be improved by adding

specialized libraries and frameworks implementing facilities such as pattern matching and tracing. Operational approaches are similar to direct ones except that they offer an executable metamodeling formalism through a dedicated language. Providing specialized facilities through libraries and frameworks seems to be an attractive way to improve the support for model transformations in an evolutionary way.

The structure-driven category covers pragmatic approaches that were developed in the context of (and seem to apply particularly well to) certain kinds of applications such as generating Enterprise JavaBeans** (EJB**) implementations and database schemas from UML models. These applications require 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, which can be system-defined. It is unclear how well these approaches can support other kinds of applications.

Template-based approaches make it easy for the developer to predict the resulting code or models just by looking at the templates. They also support iterative development in which the developer can start with a sample model or code and turn it into a template. Current template-based approaches do not have built-in support for tracing, although trace information can be easily encoded in the templates. Templates are particularly useful in code generation and model compilation scenarios.

Relational approaches seem to strike a good balance between flexibility and declarative expression. They can provide multidirectionality and incrementality, including the update of a manually modified target. On the other hand, their power is contingent on the sophistication of the underlying constraint-solving facilities. As a result, performance strongly depends on the kinds of constraints that need to be solved, which may limit their applicability. In any case, relational approaches seem to be most applicable to model synchronization scenarios.

Graph-transformation-based approaches are inspired by theoretical work in graph transformations. In their pure form, graph transformations are declarative and also seem intuitive; however, the usual fixpoint scheduling with concurrent application makes them rather difficult to use due to the

possible lack of confluence and termination. Existing theories for detecting such problems are not general enough to cover the wide range of transformations found in practice. As a result, tools such as GReAT, VIATRA, and MOLA provide mechanisms for explicit scheduling. It is often argued that graph transformations are a natural choice for model transformations because models are graphs. As Batory points out,⁸⁵ there are plenty of examples of graph structures in practice, including the objects in a Java program whose processing is usually not understood as graph transformations. In our opinion, a particular weakness of existing graph transformation theories and tools is that they do not consider ordered graphs, that is, graphs with ordered edges. As a consequence, they are applicable to models that contain predominantly unordered collections, such as class diagrams with classes having unordered collections of attributes and methods. However, they do not apply well to method bodies, where ordering is important, such as in a list of statements. Ordering can be represented by additional edges, but this approach leads to more complex transformations. It is interesting to note that ordering is well handled by classical program transformation, which uses term rewriting on abstract syntax trees (ASTs). Terms and ASTs are ordered trees, and the order of child nodes is used to encode lists of program elements such as statements. Edge ordering can be modeled in graph transformations by using edge attributes to attach an index to each edge; however, current tools based on graph transformation do not exploit this information for more efficient pattern matching. Nevertheless, graph transformation theory might turn out to be useful for ensuring correctness in some application scenarios. Fujaba is probably the largest and most significant example of applying graph transformations to models to date. It remains to be seen what impact these approaches will have on systems used in practice.

Hybrid approaches allow the user to mix and match different concepts and paradigms depending on the application. Given the wide range of practical scenarios, a comprehensive approach is likely to be hybrid. A point in case is the QVT specification, which also offers a hybrid solution.

RELATED WORK

The feature model and categorization presented in this paper is based on our earlier paper.⁸⁶ The

previous feature model has been widely discussed in workshops and in personal communications. It has also been used by other authors. For example, Jouault and Kurtev²⁴ give a classification of ATL and AMW using the earlier version of the model.

The current feature model and categories take into account the feedback that we have received based on the original paper. They were also revised to cover approaches that were proposed after 2003, most prominently, the final adopted QVT specification. Introducing domains into transformation rules was one of the most important changes to the feature model based on that specification. Only five out of the 14 presented feature diagrams remained unchanged compared with the original model, namely, those in Figures 5A, 6, and 8A–8C. We also added two new categories of model-to-model approaches, namely, operational and template-based approaches.

In their review of the different QVT submissions, Gardner et al.⁸⁷ propose a unified terminology to enable a comparison of the different proposals. As their scope of comparison is considerably different from ours, there is not much overlap in terminology. While Gardner et al. focus on the eight initial QVT submissions, we discuss a wider range of approaches: In addition to the revised QVT submissions, we also discuss other approaches published in the literature and available in tools. Another difference is that Gardner et al. discuss model queries, views, and transformations, whereas we focus on transformations in more detail. The terms defined by Gardner et al. that are also relevant for our classification are *model transformation*, *unidirectional*, *bidirectional*, *declarative*, *imperative*, and *rules*.

In addition to providing the basic unifying terminology, Gardner et al. discuss practical requirements on model transformations such as requirements scalability, simplicity, and ease of adoption. Among others, they discuss the need to handle transformation scenarios of different complexities, such as transformations with different origin relationships between source and target model elements, for example, 1-to-1, 1-to- n , n -to-1, and n -to- m . Finally, they make some recommendations for the final QVT standard. In particular, they recommend a hybrid approach, supporting declarative specification of simpler transformations, but allowing for an imperative implementation of more complex ones.

Another account of requirements for model transformation approaches is given by Sendall and Kozaczynski.⁸⁸

Mens and Van Gorp⁶⁰ have also proposed a classification of model transformations, which they apply to graph transformation systems.⁸⁹ That work has been significantly influenced by our earlier classification. The main difference is that their classification is broader as it also covers different aspects of model transformation tools such as usability, extensibility, interoperability, and standards. In contrast, our feature model offers a more detailed treatment of model transformation approaches. Another difference is that Mens and Van Gorp present a flat list of dimensions, whereas our dimensions are organized hierarchically. An extensive comparison of graph transformation approaches using a common example is given by Taentzer et al.⁹⁰

CONCLUSIONS

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 definitions set a new context.

In this paper, we presented a feature model offering a terminology for describing model transformation approaches and making the different design choices for such approaches explicit. We also surveyed and classified existing approaches into visitor-based and template-based model-to-text categories and direct-manipulation, structure-driven, operational, template-based, relational, graph-transformation-based, and hybrid model-to-model categories.

Although 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 three years, but relatively little experience is available to assess 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, that is, without proper theoretical foundation.

Evaluation of the different design options for a model transformation approach will require more experiments and practical experience.

ACKNOWLEDGMENTS

The authors thank Karl-Trygve Kalleberg for his extensive feedback on earlier versions of the feature model, Markus Völter for providing the template examples in Examples 2 and 3, and Ulrich Eisenecker, Don Batory, Jeff Gray, Laurence Tratt, Gabriele Taentzer and the anonymous reviewers for their valuable comments on a previous draft of the paper.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Object Management Group, Inc., Sun Microsystems Inc., Compuware Corporation, MetaCase Consulting, Interactive Objects Software GmbH Corporation, The MathWorks, Inc., or Delta Software Technology GmbH, in the United States, other countries, or both.

CITED REFERENCES

1. T. Stahl and M. Völter, *Model-Driven Software Development—Technology, Engineering, Management*, John Wiley and Sons, Ltd., Chichester, England (in press June 2006), ISBN: 0470025700.
2. Object Management Group, *MDA Guide, Version 1.0.1*, OMG Document omg/2003-06-01 (2003).
3. D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley Press, Hoboken, NJ (2003).
4. J. Sztipanovits and G. Karsai, “Model-Integrated Computing,” *Computer* **30**, No. 4, 110–111 (1997).
5. J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley and Sons, Indianapolis, IN (2004).
6. K. Czarnecki, “Overview of Generative Software Development,” *Proceedings of Unconventional Programming Paradigms*, Mont Saint-Michel, France (2004), pp. 313–328.
7. A. Kleppe, J. Warmer, and W. Bast, *MDA Explained, The Model Driven Architecture: Practice and Promise*, Addison-Wesley, Boston, MA (2003).
8. I. Ivkovic and K. Kontogiannis, “Tracing Evolution Changes of Software Artifacts through Model Synchronization,” *Proceedings of the 20th IEEE International Conference on Software Maintenance*, Washington, DC (2004), pp. 252–261.
9. R. I. Bull and J.-M. Favre, “Visualization in the Context of Model Driven Engineering,” *Proceedings of the Workshop on Model Driven Development of Advanced User Interfaces*, Montego Bay, Jamaica (2005), <http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-159/paper8.pdf>.
10. A. Solberg, R. France, and R. Reddy, “Navigating the MetaMuddle,” *Proceedings of the 4th Workshop in Software Model Engineering*, Montego Bay, Jamaica (2005), <http://www.planetmde.org/wisme-2005/NavigatingTheMetaMuddle.pdf>.
11. G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel, “Refactoring UML Models,” *Proceedings of the 4th International Conference, Unified Modeling Language Conference*, Toronto, Canada (2001), pp. 134–148.
12. J. Zhang, Y. Lin, and J. Gray, “Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine,” Chapter 9 in *Model-Driven Software Development*, S. Beydeda, M. Book, and V. Gruhn, Editors, Springer-Verlag, Heidelberg, Germany (2005), pp. 199–218.
13. J.-M. Favre, “CacOphoNy: Metamodel-Driven Architecture Reconstruction,” *Proceedings of the Working Conference on Reverse Engineering*, Delft, The Netherlands (2004), pp. 204–213.
14. Object Management Group, *MOF 2.0 Query/Views/Transformations RFP*, OMG Document ad/2002-04-10 (revised on April 24, 2002).
15. Object Management Group, *MOF QVT Final Adopted Specification*, OMG Adopted Specification ptc/05-11-01 (2005).
16. D. Varró, G. Varró, and A. Pataricza, “Designing the Automatic Transformation of Visual Languages,” *Science of Computer Programming* **44**, No. 2, 205–227 (2002).
17. D. Varró and A. Pataricza, “Generic and Meta-Transformations for Model Transformation Engineering,” *Proceedings of the 7th International Conference on the Unified Modeling Language*, Lisbon, Portugal (2004), pp. 290–304.
18. D. H. Akehurst and S. J. H. Kent, “A Relational Approach to Defining Transformations in a Metamodel,” *Proceedings of the 5th International Conference on the Unified Modeling Language*, Dresden, Germany (2002), pp. 243–258.
19. D. H. Akehurst, W. G. Howells, and K. D. McDonald-Maier, “Kent Model Transformation Language,” *Proceedings of Model Transformations in Practice Workshop, MoDELS Conference*, Montego Bay, Jamaica (2005), http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/akehurst_howells_mcdonald-maier_kent_model_transformation_language.pdf.
20. A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, “Transformation: The Missing Link of MDA,” *Proceedings of the 1st International Conference on Graph Transformation*, Barcelona, Spain (2002), pp. 90–105.
21. M. Lawley and J. Steel, “Practical Declarative Model Transformation with Tefkat,” *Proceedings of Model Transformations in Practice Workshop, MoDELS Conference*, Montego Bay, Jamaica (2005), http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/lawley_steel_practical_declarative_model_transformation_with_tefkat.pdf.
22. A. Agrawal, G. Karsai, and F. Shi, *Graph Transformations on Domain-Specific Models*, Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37203 (2003).
23. J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui, “First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery,” *Proceedings of the Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA

- (2003), <http://www.softmetaware.com/oopsla2003/bezivin.pdf>.
24. F. Jouault and I. Kurtev, "Transforming Models with ATL," *Proceedings of Model Transformations in Practice Workshop (MTIP), MoDELS Conference*, Montego Bay, Jamaica (2005), http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev_transforming_models_with_atl.pdf.
 25. E. D. Willink, "UMLX: A Graphical Transformation Language for MDA," *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA (2003), pp. 13–24 (2003).
 26. J. de Lara and H. Vangheluwe, "AToM: A Tool for Multi-Formalism and Meta-Modeling," *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, Grenoble, France (2002), pp. 174–188.
 27. P. Braun and F. Marschall, *The Bi-directional Object-Oriented Transformation Language*, Technical Report TUM-I0307, Technische Universität München 85748, München, Germany (May 2003).
 28. F. Marschall and P. Braun, "Model Transformations for the MDA with BOTL," *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, Enschede, The Netherlands (2003), pp. 25–36.
 29. A. Kalnins, J. Barzdins, and E. Celms, "Model Transformation Language MOLA," *Proceedings of Model Driven Architecture: Foundations and Applications*, Linköping, Sweden (2004), pp. 14–28.
 30. G. Taentzer, "AGG: A Graph Transformation Environment for Modeling and Validation of Software," *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)* **3062**, pp. 446–453 (2003).
 31. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez, "Modeling in the Large and Modeling in the Small," *Proceedings of the European MDA Workshops: Foundations and Applications*, Twente, The Netherlands (2003), and Linköping, Sweden (2004), pp. 33–46.
 32. A. Königs, "Model Transformation with Triple Graph Grammars," *Proceedings of Model Transformations in Practice Workshop at MoDELS Conference*, Montego Bay, Jamaica (2005), http://www.es.tu-darmstadt.de/download/publications/koenigs/model_transformation_with_triple_graph_grammars.pdf.
 33. D. Vojtisek and J.-M. Jézéquel, "MTL and Umlaut NG: Engine and Framework for Model Transformation," http://www.ercim.org/publication/Ercim_News/enw58/vojtisek.html.
 34. O. Patrascoiu, "YATL: Yet Another Transformation Language," *Proceedings of the 1st European MDA Workshop*, Twente, The Netherlands (2004), pp. 83–90.
 35. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving Executability into Object-Oriented Metalanguages," *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica (2005), pp. 264–278.
 36. J. Gray, Y. Lin, and J. Zhang, "Automating Change Evolution in Model-Driven Engineering," *IEEE Computer (Special issue on Model-Driven Engineering)* **36**, No. 2, 51–58 (February 2006), <http://www.cis.uab.edu/gray/Pubs/computer.pdf>.
 37. L. Tratt, "The MT Model Transformation Language," *Proceedings of ACM SIGAPP Symposium on Applied Computing*, Dijon, France (2006), <http://portal.acm.org/affiliated/citation.cfm?id=1141277.1141577&coll=ACM&dl=ACM&type=series&idx=1141277&part=Proceedings&WantType=Proceedings&title=Symposium%20on%20Applied%20Computing&CFID=15151515&CFTOKEN=6184618>.
 38. AndroMDA 2.0.3, <http://www.andromda.org>.
 39. openArchitectureWare (oAW), <http://www.openarchitectureware.org/>.
 40. Fujaba Tool Suite 4, University of Paderborn Software Engineering, <http://www.fujaba.de>.
 41. JAMDA, Java Model Driven Architecture 0.2, <http://sourceforge.net/projects/jamda>.
 42. R. Pompa, *Java Emitter Templates (JET) Tutorial*, Azzurri Ltd. (June 2005), http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.
 43. G. van Emde Boas, *The Fantastic, Unique, UML Tool for the Java Environment (FUUT-je)*, <http://www.bronstee.com/index.php?id=FUUT-je>.
 44. *Model Transformation Framework (MTF)*, IBM United Kingdom Laboratories Ltd., IBM alphaWorks (2004), <http://www.alphaworks.ibm.com/tech/mtf>.
 45. XMF-Mosaic, Xactium, <http://xactium.com>.
 46. *OptimalJ 4.0, User's Guide*, Compuware (June 2005), <http://www.compuware.com/products/optimalj>.
 47. J.-P. Tolvanen, "Making Model-Based Code Generation Work," *Embedded Systems Europe*, pp. 36–38 (August/September 2004), <http://i.cmpnet.com/embedded/europe/esesep04/esesep04p36.pdf>.
 48. Domain-Specific Modeling with MetaEdit+, MetaCase, <http://www.metacase.com/>.
 49. ArcStyler 5.1, Interactive Objects Software GmbH, <http://www.arcstyler.com>.
 50. Codagen Architect 3.0, Codagen Technologies Corp., <http://www.codagen.com/products/architect/default.htm>.
 51. H. Partsch and R. Steinbrüggen, "Program Transformation Systems," *ACM Computing Surveys* **15**, No. 3, 199–236 (1983).
 52. H. Partsch, *Specification and Transformation of Programs—a Formal Approach to Software Development*, Springer-Verlag, Berlin, Germany (1990).
 53. K. Czarnecki, "Domain Engineering," in *Encyclopedia of Software Engineering* Second Edition, J. J. Marciniak, Editor, John Wiley and Sons, Inc., Hoboken, NJ (2002), pp. 433–444.
 54. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Nowak, and A. S. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213 (1990).
 55. K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley Professional, Boston, MA (2000).
 56. K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing Cardinality-Based Feature Models and Their Specialization," *Software Process: Improvement and Practice* **10**, No. 1, 7–29 (2005).
 57. C. H. P. Kim and K. Czarnecki, "Synchronizing Cardinality-Based Feature Models and their Specializations," *Proceedings of the European Conference on Model Driven Architecture*, Nuremberg, Germany (2005), [swen.uwaterloo.ca/~kczarneck/cmda05.pdf](http://www.uwaterloo.ca/~kczarneck/cmda05.pdf).
 58. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien, *Model Transformation Contracts and Their Definition in UML and OCL*, Technical Report LIFL 2004-n°08, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, 59655 Villeneuve d'Ascq Cedex, France (2004).
 59. *MOF Query/Views/Transformations*, OMG Document ad/03-08-08, Object Management Group, Inc., (revised in August 2003).

60. *Meta Object Facility (MOF) 2.0 Core Specification*, OMG Adopted Specification ptc/03-10-04, Object Management Group, Inc., (2003).
61. T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation and Its Application to Graph Transformation," *Proceedings of the International Workshop on Graph and Model Transformation*, Tallinn, Estonia (2005), pp. 7–23.
62. E. Visser, Program-Transformation.Org: The Program Transformation Wiki, <http://www.program-transformation.org/Transform/ProgramTransformation>.
63. Java Metadata Interface 1.0 (JMI), Sun Microsystems, Inc. (June 2002), <http://java.sun.com/products/jmi>.
64. E. Visser, "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9," *Proceedings of the International Domain-Specific Program Generation Seminar*, Dagstuhl, Germany (2003), pp. 216–238.
65. R. Silaghi, F. Fondement, and A. Strohmeier, "Weaving MTL Model Transformations," *Proceedings of the European MDA Workshops: Foundations and Applications*, Twente, The Netherlands (2003), and Linköping, Sweden (2004), pp. 123–138.
66. I. Kurtev, *Adaptability of Model Transformations*, PhD thesis, University of Twente, Enschede, The Netherlands (2005), <http://wwwhome.cs.utwente.nl/~kurtev/files/thesis.pdf>.
67. Object Management Group, *Interactive Objects and Project Technology, MOF Query/Views/Transformations*, OMG Document ad/03-08-11, ad/03-08-12, and ad/03-08-13 (revised submission, 2003).
68. Object Management Group, *Response to the MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10)*, OMG Document ad/2003-08-05 (2003).
69. Object Management Group, *MOF Query/Views/Transformations, First Revised Submission*, OMG Document ad/03-08-03 (2003).
70. Object Management Group, *MOF Model to Text Transformation Language RFP*, OMG Document ad/2004-04-07 (2004).
71. Velocity 1.4, The Apache Jakarta Project, The Apache Software Foundation, <http://jakarta.apache.org/velocity>.
72. XDoclet—Attribute Oriented Programming, <http://xdoclet.sourceforge.net/xdoclet/index.html>.
73. J. C. Cleaveland, *Program Generators with XML and Java*, Prentice-Hall, Upper Saddle River, NJ (2001).
74. *XML Path Language (XPath) 2.0*, A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon, Editors, W3C Candidate Recommendation (November 3, 2005), <http://www.w3.org/TR/xpath20/>.
75. P. G. Bassett, *Framing Software Reuse: Lessons from the Real World*, Prentice-Hall, Inc., Upper Saddle River, NJ (1997).
76. S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang, "XVCL: XML-Based Variant Configuration Language," *Proceedings of the International Conference on Software Engineering*, Portland, OR (2003), pp. 810–811.
77. M. Emrich, *Generative Programming Using Frame Technology*, Diploma thesis, University of Applied Sciences, Department of Computer Science and Micro-System Engineering, Kaiserslautern, Germany (2003).
78. Frame Processor ANGIE, Delta Software Technology, http://www.d-s-t-g.com/neu/media/pdf/facts_e/ DLT21474.pdf.
79. K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, Tallinn, Estonia (2005), pp. 422–437.
80. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer, *Graph Transformation for Specification and Programming*, Technical Report 7/96, University of Bremen, 1-D-28359 Bremen, Germany (1996).
81. *XSL Transformations (XSLT), Version 1.0*, James Clark, Editor, W3C Recommendation (November 16, 1999), <http://www.w3.org/TR/xslt>.
82. *MOF 2.0/XMI Mapping Specification, Version 2.1*, OMG Document formal/05-09-01, Object Management Group, Inc. (2005).
83. M. Peltier, F. Ziserman, and J. Bézivin, "On Levels of Model Transformation," *XML Europe, Paris, France (2000)*, pp. 1–17. Graphic Communications Association, 2000.
84. M. Peltier, J. Bézivin, and G. Guillaume, "MTRANS: A General Framework, Based on XSLT, for Model Transformations," *Proceedings of the Workshop on Transformations in UML*, Genova, Italy (April 2001), <http://www.sciences.univ-nantes.fr/lina/atl/www/papers/peltier-bezivin-guillaume.pdf>.
85. D. Batory, "Multilevel Models in Model-Driven Engineering, Product Lines, and Metaprogramming," *IBM Systems Journal* **45**, No. 3, 527–540 (2006, this issue).
86. K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," *Proceedings of the 2nd Workshop on Generative Techniques in the Context of MDA*, Anaheim, CA (2003), http://www.swen.uwaterloo.ca/~kczarneck/ECE75077/czarnecki_helsen.pdf.
87. T. Gardner, C. Griffin, J. Koehler, and R. Hauser, "A Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations Toward the Final Standard," Object Management Group, OMG Document ad/03-08-02 (2003), <http://www.omg.org/cgi-bin/doc?ad/03-08-02>.
88. S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE Software* **20**, No. 5, 42–45 (2003).
89. T. Mens, P. Van Gorp, D. Varró, and G. Karsai, "Applying a Model Transformation Taxonomy to Graph Transformation Technology," *Proceedings of the International Workshop on Graph and Model Transformation*, Tallinn, Estonia (2005), pp. 24–39.
90. K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Leventovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay, "Model Transformation by Graph Transformation: A Comparative Study," *Proceedings of Model Transformations in Practice Workshop, MoDELS Conference*, Montego Bay, Jamaica (2005), <http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/mtip05.pdf>.

Accepted for publication January 25, 2006.

Published online July 25, 2006.

Krzysztof Czarnecki

University of Waterloo, Department of Electrical & Computer Engineering, 200 University Avenue West, Waterloo, ON N2L 3G1, Canada (kczarneck@swen.uwaterloo.ca).

Dr. Czarnecki is an Assistant Professor at the University of Waterloo. He is a coauthor of *Generative Programming: Methods, Tools, and Applications* (Addison-Wesley, 2000), regarded as a foundational work in its area and used as a

graduate text at universities around the world. Dr. Czarnecki's current work focuses on realizing the synergies between generative and model-driven software development.

Simon Helsen

SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany (simon.helsen@sap.com). Dr. Helsen is a senior developer at SAP AG, where he works on modeling infrastructure software for SAP NetWeaver® development tools. He has an Informatics degree from the University of Leuven (Belgium) and a Ph.D. degree in computer science from the University of Freiburg (Germany). Dr. Helsen's current interests are in scalable model-driven software engineering, domain-specific languages, and model transformations. ■