Feature-Based Survey of Model Transformation Approaches

Krzysztof Czarnecki
University of Waterloo
Waterloo, Canada

Simon Helsen
SAP AG
Walldorf, Germany

March 15, 2006

Abstract

Model transformations are touted to play a key role in model-driven development. While well-established standards for meta-modeling such as the Meta-Object Facility exist, there is currently no matured foundation for specifying transformations among models. In this paper, 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 the different design choices for model transformations explicit. Based on our analysis of the model transformation approaches, we propose a few major categories in which most the approaches fit.

Introduction

Model-driven software development is centered on the use of models.[72] Models are system abstractions allowing developers and other stakeholders to effectively address their concerns, such as answering a particular question about the system or effecting a particular change. Examples of model-driven approaches are Model-Driven Architecture (MDA),[4;39] Model-Integrated Computing (MIC),[74] and Software Factories.[43] 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.[28]

Model transformations are touted to play a key role in model-driven development. Their intended applications include

- generating lower-level models, and eventually code, from higher-level models;[54]
- mapping and synchronizing among models at the same level or different levels of abstraction;[47]
- creating query-based views on a system;[21;71]
- model evolution tasks such as model refactoring;[73;90] and
- reverse engineering of higher-level models from lower-level ones.[38]

A considerable amount of interest in model transformations has been generated by the Object Managements Group's (OMG) standardization effort. In April 2002, the OMG has issued a Request for Proposal (RFP) on Query / Views / Transformations (QVT),[6] which led to the release of the final adopted QVT specification in November 2005.[5] 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 writing, industrial-strength and matured model-to-model transformation systems are still not available, and the area of model transformations 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

- published in the literature: VIATRA (VIsual Automated model TRAnsformations) framework,[34;82] Kent Model Transformation Language,[11;12] Tefkat,[42;57] GReAT (Graph Rewriting and Transformation) language,[10] ATL (Atlas Transformation Language),[17;50] UMLX,[87] AToM^3 (A Tool for

Multi-formalism and Meta-Modeling) system,[35] BOTL (Bidirectional Object-oriented Transformation Language),[20;58] MOLA (MOdel transformation LAnguage),[51] AGG (Attributed Graph Grammar) system,[75] AMW (Atlas Model Weaver),[18] triple-graph grammars,[55] MTL (Model Transformation Language),[85] YATL (Yet Another Transformation Language),[65] Kermeta,[61] and MT (Model Transformation) language;[79]

- described in the final adopted QVT specification: the Core, Relations, and Operational languages;[5] older QVT submissions are also mentioned whenever appropriate;
- implemented within open-source tools: AndroMDA,[1] openArchitectureWare,[19] Fujaba (From UML to Java And Back Again),[40] Jamda (JAva Model Driven Architecture),[2] JET (Java Emitter Templates),[68] FUUT-je,[80] and MTF (Model Transformation Framework)[44] as a freely available prototype;
- implemented within commercial tools: XMF-Mosaic,[89] OptimalJ,[26] MetaEdit+,[78] ArcStyler,[46] and Codagen Architect.[25]

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

What Is Model Transformation?

Before delving into a discussion of approaches to model transformation, let us first try to characterize the concept of 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 lists and trees, is at the heart of programming. Type systems in programming languages help to ensure that operations are applied compatibly to the data. However, when the subject of a transformation approach is meta-data, i.e., data representing software artifacts such as data schemas, programs, interfaces, and models, then we enter the realm of metaprogramming. For example, one of the key challenges in metaprogramming is that metaprograms have to respect the rich semantics of the meta-data they operate on. Similarly, model transformation is also a form of metaprogramming and, thus, must face that same challenge too.

Model transformation is closely related to program transformation.[64] 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 flavors. 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.

Since model transformations operate on models, we need to clarify what models are. A model is an abstraction of a system and/or its environment. Czarnecki summarizes the role of models in engineering as follows:[29] "Models allow engineers to effectively address their concerns about the system such as answering particular questions or devising required design changes. [...] A particular model may be appropriate for answering a certain class of questions, where the answers to those questions will be the same for the model as for the actual system. However, the same model may not be appropriate for answering some other class of questions. Moreover, models are cheaper to build than a real system. For example, civil engineers create static and dynamic structural models of bridges to check structural safety since modeling is certainly cheaper and more effective than building real bridges to see under what scenarios they will collapse."

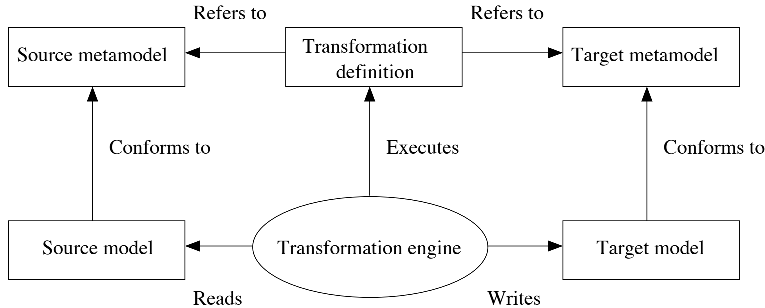


Figure 1: Basic concepts of model transformation

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. While being visual is not a defining characteristic of models, requirements and design models are often more visual than programs. Models are frequently expressed in focused languages specialized for a particular class of software applications and/or particular aspect of an application. For example, the Matlab Simulink/Stateflow environment offers notations specialized for modeling control software, while UML-style interaction diagrams are focused for 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. Since 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 towards transforming particular kinds of models. For example, as explained in Section Section "Discussion", most of the model transformation approaches based on graph transformations are better suited for transforming UML models than program code. However, it is important to note that 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,[63] 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 including 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., so-called n-way transformations), and the multi-directionality of transformations. While these requirements could also be the subject of program transformation approaches, they are typically not considered by program transformation systems.

Examples of Model Transformations

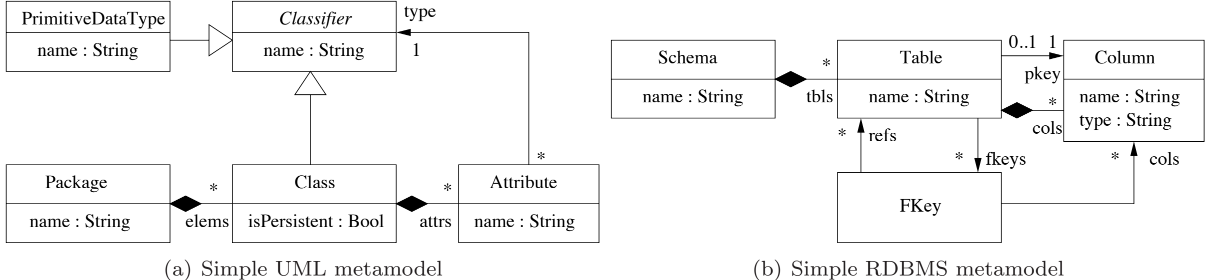
In order to make our discussion more concrete, we present two examples of model transformations: one mapping models to models, and another one mapping models to code.

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.

Before looking at sample definitions of model transformations, we need to present the metamodels for our examples.

Sample Metamodels and Models

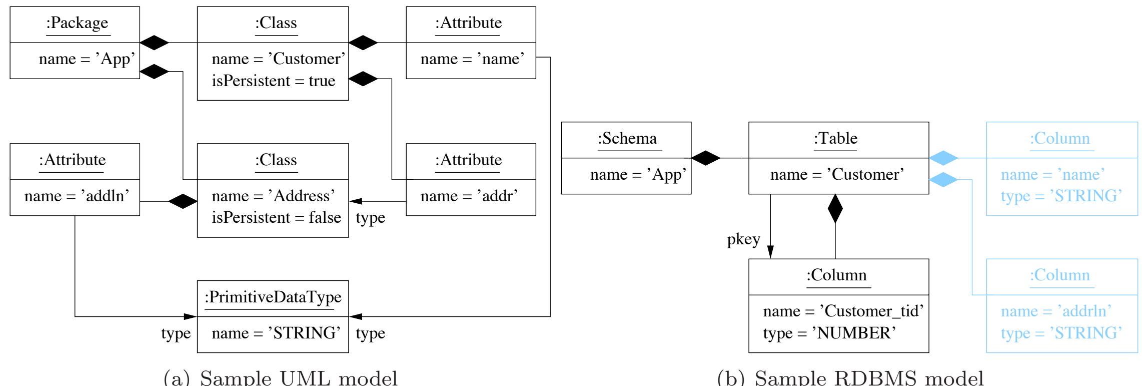
Figure 2 shows two sample metamodels expressed as UML class diagrams. Figure 2(a) gives a simplified metamodel for class models. The metamodel includes the abstract concept of classifiers, which comprises classes and primitive data types. Packages contain classes, and classes contain attributes. All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



(a) Simple UML metamodel (b) Simple RDBMS metamodel

Figure 2: Metamodels for the UML-to-RDBMS example

Sample instances of the metamodels are shown in Figure 3 using the UML object diagram notation. The instance in Figure 3(a) represents a class model with one package, `App`, containing two classes, `Customer` and `Address`. `Customer` is persistent, and `Address` is not. Figure 3(b) shows an instance of the schema metamodel. The instance represents a schema that can be used to persist `Customer` objects.



(a) Sample UML model (b) Sample RDBMS model

Figure 3: Sample models

UML-To-Schema Transformation Example

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 to it.

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 3(a) to the schema model in Figure 3(b). The part of the result in Figure 3(b) shown in black is handled by the first two mappings. The light-colored part corresponds to the result of the attribute-to-column mapping.

Figure 4 shows how this transformation can be expressed using the QVT Relations language, which is a declarative language for model-to-model transformations. The transformation declaration specifies two parameters, `uml` and `rdbms`, which will hold 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 model and vice versa. Only upon invoking the transformation, the user has to specify in which direction the transformation has to be executed.

```
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 {
  ...
  }
  ...
}
```

Figure 4: Transformation expressed in QVT Relations

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 using a *when clause*. The *where clause* specifies additional constraints among the involved elements. The *key definitions* are used by the transformation engine to identify target objects that need to be updated during a transformation execution. There is a lot more to say about the execution semantics of QVT Relations. The interested reader is invited to explore the QVT specification document.

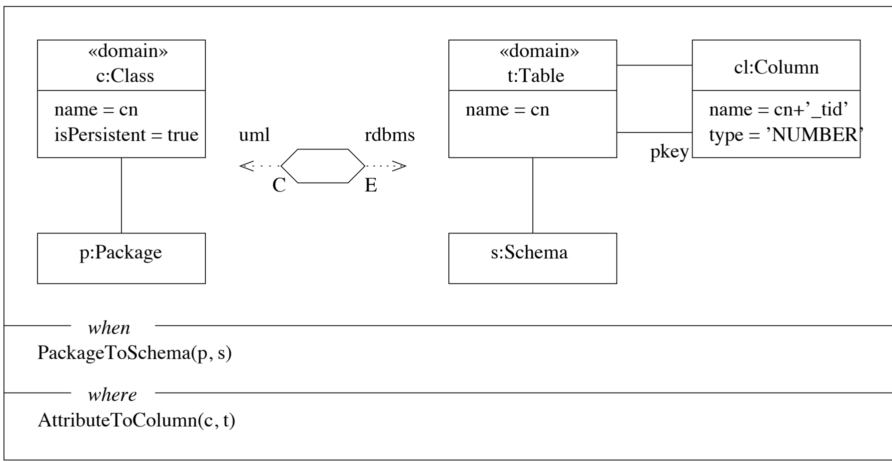


Figure 5: Graphical notation of a QVT relation

The QVT Relations language also has a graphical notation. Figure 5 shows the `ClassToTable` relation in that notation.

UML-To-Java Transformation Example

In the second example, we would like to generate Java code from class models conforming to the metamodel in Figure 2(a). In particular, a Java class with the appropriate attribute definitions and getters and setters should be generated for each class in the class model. Figure 6 shows the desired output for the input model from Figure 3(a).

```
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 name ) {
    this.addr = addr;
  }

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

Figure 6: Java code to be generated

The code can conveniently be generated using a textual template approach, such as the *openArchitectureWare* template language demonstrated in Figure 7. 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 Figure 7 is show in light color. The metacode has facilities to iterate over the elements of the input model (`FOREACH`), access the properties of the elements, and call templates from other templates (`EXPAND`).

```
<<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>>
```

Figure 7: Model-to-code transformation with openArchitectureWare

Table 1: Symbols used in cardinality-based feature modeling

Symbol	Explanation
F	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
F	Solitary feature with cardinality [0..1], i.e., *optional* feature
[n..m] F	Solitary feature with cardinality $[n..m]$, $n \geq 0 \wedge m \geq n \wedge m > 1$, i.e., *mandatory clonable* feature
F	Grouped feature with cardinality [0..1]
F ▸	Feature model reference F
	Feature group with cardinality $\langle 1\text{–}1 \rangle$, i.e. *xor*-group
	Feature group with cardinality $\langle 1\text{–}k \rangle$, where k is the group size, i.e. *or*-group

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.[27] We document our results using feature diagrams.[52;31] 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 a representation of the design choices for model transformation approaches. We should note that 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. However, 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 8 shows the top-level feature diagram, where each subnode represents a major point of variation. The fragment of the *cardinality-based feature modeling notation*[33;53] 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 Section "Major Categories". The description of the top-level features in Figure 8 follows.

- *Specification.* Some transformation approaches provide a dedicated specification mechanism, such as pre- and post conditions expressed in OCL.[22] 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-Partners[62] submission distinguished between relations as potentially non-executable specifications of transformations and their

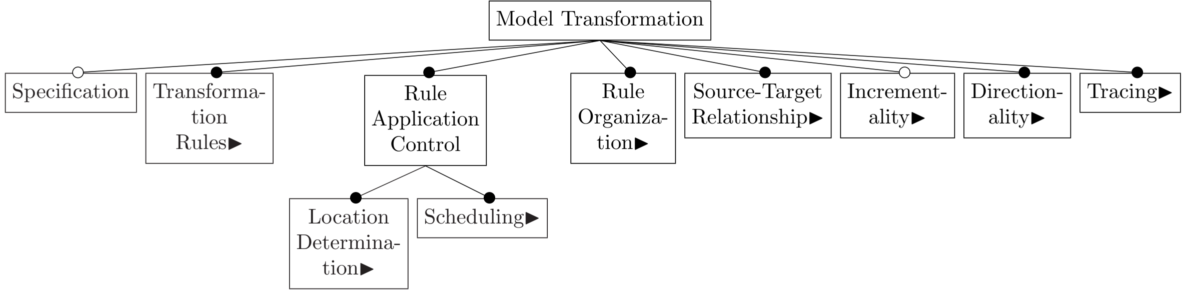


Figure 8: Top-level feature diagram

executable implementations. The QVT specification[5] still keeps this distinction, although the Relations language in that submission 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 left-hand side (LHS) and a right-hand 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 or ML resemble rules with patterns on the left and expressions on the right. Templates can be considered as a degenerate form of rules, as it will be discussed later in Section "Template-Based Approaches".

- *Rule application control.* Rule application control has two aspects: *scheduling* and *location determination.* Scheduling determines the order in which transformation rules are executed. Location determination is the strategy for determining the model locations to which transformation rules are applied. Although control mechanisms usually address both aspects at the same, for presentation purposes, we discuss them separately.

- *Rule organization.* Rule organization comprises general structuring issues such as modularization and reuse mechanisms.

- *Source-target relationship.* Source-target relationship is concerned with issues such as whether source and target are one and the same model or two different models.

- *Incrementality.* Incrementality refers to the ability to update existing target models based on changes in the source models.

- *Directionality.* Directionality describes whether a transformation can be executed in only one direction (unidirectional transformation) or multiple directions (multidirectional transformation).

- *Tracing.* Tracing is concerned with the mechanisms for recording different aspects of transfomation execution such as creating and maintaining trace links between source and target model elements.

Each of the following subsections elaborates on one major area of variation represented as a reference in Figure 8 by giving its feature diagram, describing the different choices in the text, and providing examples of approaches supporting a given feature. The diagrams remain at a certain level of detail in order to fit into 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 programming paradigms (see Figure 12) will be 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 9. Their description follows.

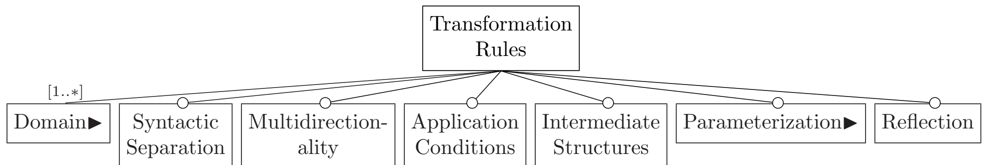


Figure 9: Features of transformation rules

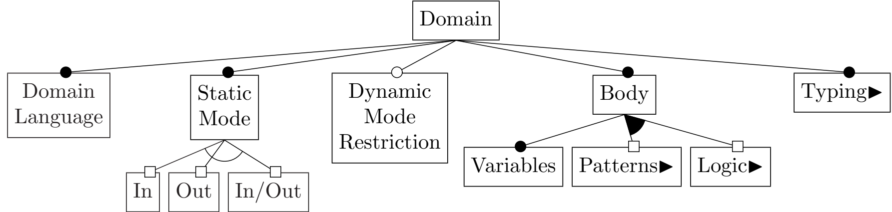


Figure 10: Transformation rule domains

- *Domains.* A domain is the part of a rule that is responsible for accessing one of the models the rule operates on. 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*,[18] 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 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 (Figure 4). In an 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 and/or creates model elements 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 Figure 7 corresponds to the metacode travering the source model, whereas the target domain has the form of string patterns.

  The features of a domain are shown in Figure 10.

  - *Domain language.* A domain has an associated 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).[9] 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*.[84;59]
  - *Static mode.* 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 restricting the static modes at execution time. For example, MTF allows marking any of the participating domains as read-only, i.e., restricting them to *in* for a particular execution of a transformation. Essentially, such restrictions define the execution direction.
  - *Variables.* 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.* 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 expres-

sions and statements of the meta-language. Depending on the internal representation of the models being transformed, we can have *string*, *term*, or *graph* patterns (see Figure 11). String patterns are used in textual templates, as discussed in Section "Template-Based Approaches" on page 16. Model-to-model transformations usually apply term or graph patterns. Patterns can be represented using *abstract* or *concrete* syntax of the corresponding source or target model language, and the syntax can be textual and/or graphical.

  - *Logic*. Logic expresses computations and constraints on model elements (see Figure 12). Logic may follow different programming paradigms such as the object-oriented or functional paradigm and be non-executable or executable. Non-executable 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 APIs to manipulate models directly. For instance, the Java Metadata Interface (JMI)[49] 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*. Variables, logic, and patterns can be untyped, syntactically typed, or semantically typed (see Figure 13). An example of untyped patterns are textual templates (Section "Template-Based Approaches" on page 16). 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 guarantee 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 clearly separate the parts of a rule operating on one domain from the parts operating on other domains. For example, classical rewrite rules have a LHS operating on the source domain and a RHS operating on the target domain. 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. Rules supporting multidirectionality are usually defined over in/out-domains. Multidirectional rules are available in MTF and QVT Relations.

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

- *Intermediate structures*. The execution of a rule may require the creation of some additional structures 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 in order to prevent multiple firings of a rule for the same input element.

- *Parameterization*. The simplest kind of parameterization are *control parameters*, which allow passing values as control flags (see Figure 14). Control parameters are useful to implement 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. The importance of control parameters has been emphasized by Kleppe et al.[54] *Generics* allow passing data types, including model element types, as parameters. Generics can help making transformation rules more reusable. Generic transformations have been described by Varró and Pataricza.[81] Finally, *higher-order rules* take other rule as parameters. Higher-order rules may provide even higher levels of reuse and abstraction. Stratego[83] 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.

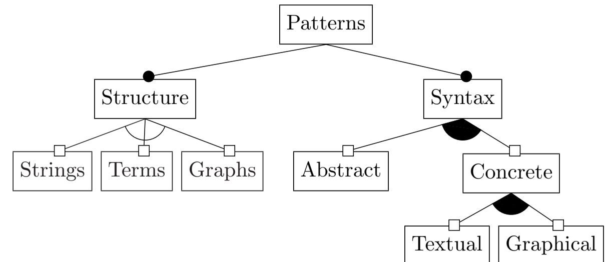


Figure 11: Patterns

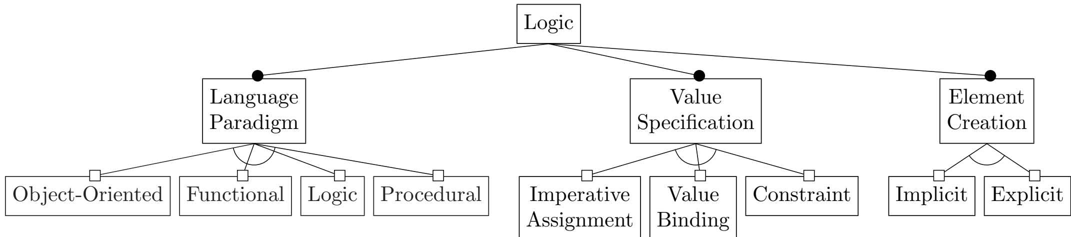


Figure 12: Logic

- *Reflection and aspects.* Some authors advocate the support for reflection and aspects in transformation languages. Reflection is supported by 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.[70] Reflection and aspects can be used to express concerns that crosscut several rules, such as custom traceability management policies.[56]

Location Determination

A rule needs to be applied to a specific location within its source scope. Since there may be more than one match for a rule within a given source scope, we need a strategy for determining the application locations. The strategy could be *deterministic*, *non-deterministic*, or *interactive*. For example, a deterministic strategy could exploit some standard traversal strategy (such as depth-first) over the containment hierarchy in the source. Stratego[83] is an example of a term rewriting language with a rich mechanism for expressing traversal in tree structures. Examples of non-deterministic strategies include *one-point* application, where a rule is applied to one non-deterministically selected location, and *concurrent* application, where one rule is applied concurrently to all matching locations in the source. Concurrent application is supported in AToM^3, AGG, and VIATRA. AGG offers so-called *critical pair analysis* to verify for a set of rules that there will be no rules competing for the same source location. Some tools, e.g., AToM^3, 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 some target element that was created by some other rule and use the element as its own target. In the case of in-place update, the source location usually becomes the target location, although traceability links can also be used.

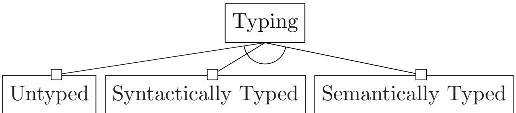


Figure 13: Typing

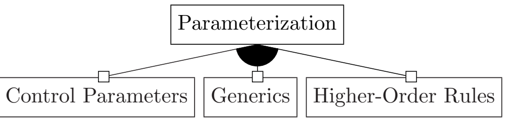


Figure 14: Parameterization

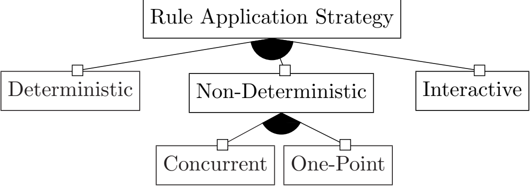


Figure 15: Location determination

Rule Scheduling

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

- *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, e.g., BOTL. The only way a user can influence the system-defined scheduling algorithm is by designing the patterns and logic of the rules to guarantee certain execution orders. For example, a given rule could check for some information that only some other rule would produce. Explicit scheduling has dedicated constructs to explicitly control the execution order. Explicit scheduling 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 would be a mechanism allowing a transformation rule to directly invoke other rules as in ATL.

- *Rule selection.* Rules can be selected by an explicit condition as in MOLA. Some approaches, such as BOTL, offer non-deterministic 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 AToM^3.

- *Rule iteration.* Rule iteration mechanisms include recursion, looping, and fixpoint iteration (i.e., repeated application until no changes detected). For example, ATL supports recursion; MOLA has a looping construct; and VIATRA supports fixpoint iteration.

- *Phasing.* The transformation process may be organized into several phases, where each phase has 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 [45] 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 (Section "Structure-Driven Approaches").

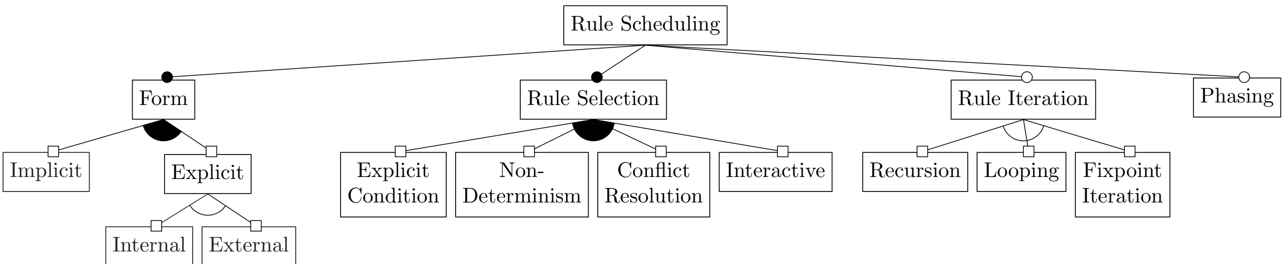


Figure 16: Rule scheduling

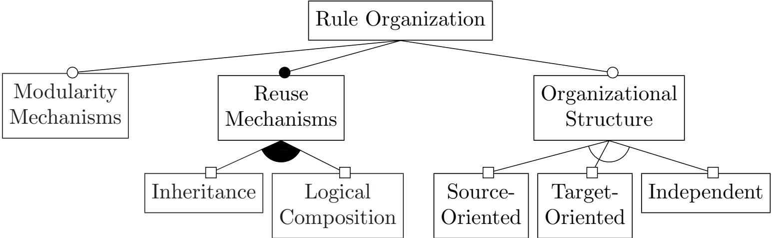


Figure 17: Rule organization

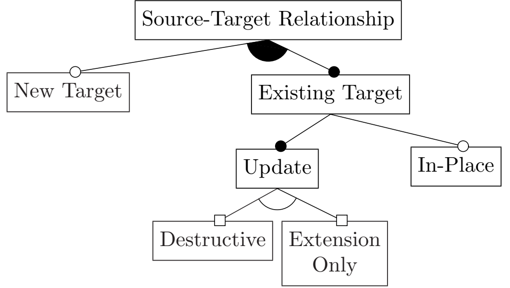


Figure 18: Source-target relationship

Rule Organization

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

- *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.
- *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,[13] derivation,[45] extension,[23] and specialization[62]), inheritance between modules (e.g., unit inheritance[13]), and logical composition.[62]
- *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.[45] 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. 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, i.e., they only support in-place update. Yet other approaches, e.g., 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, i.e., where existing model elements cannot be removed. Approaches using non-deterministic selection and fixpoint iteration scheduling (Section "Rule Scheduling") 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.

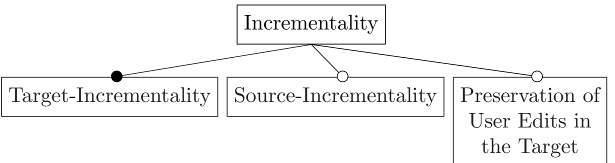


Figure 19: Incrementality

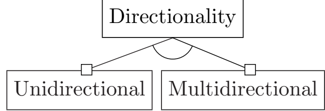


Figure 20: Directionality

Incrementality

Incrementality involves three different features (see Figure 19):

- *Target-incrementality.* The basic feature of all incremental transformations is target-incrementality, i.e., 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.[5] Obviously, target-incrementality corresponds to the feature *update* in Figure 18, but it is now seen from the change-propagation perspective. A target-incremental transformation will create 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, 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.

- *Source-incrementality.* Source-incrementality is about minimizing the amount of source that needs to be re-examined 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 when working with large source models.

- *Preservation of user edits in the target.* Practical scenarios in the context of *model synchronization* require the ability to re-run a transformation on an existing user-modified target in order to re-synchronize 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.[53]

Directionality

Transformations may be unidirectional or multidirectional (see Figure 20). 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 are usually designed to 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 and/or patterns) can often be applied in the inverse direction, too. However, since different inputs may lead to the same output, the inverse of a rule may not be a function. In this case, the inversion could enumerate a number of possible solutions (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 invertability of a transformation depends not only on the invertability of the transformation rules, but also on the

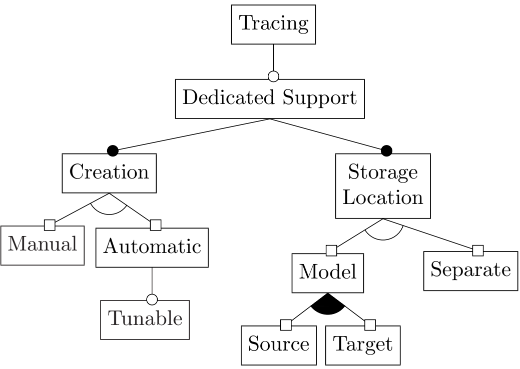


Figure 21: Tracing

invertability of the scheduling logic. Inverting a set of rules may fail to produce any result due to non-termination.

Tracing

Tracing can be understood as the runtime footprint of transformation execution. A common form of trace information in model transformation are *traceability links* 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 any dedicated support for tracing, 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 for tracing, such as Tefkat, require developers to manually encode the creation of traceability links in the transformation rules, while other, 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 and/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. 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 non-code 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 simplicity, code is often generated simply as text, which is then fed into a compiler. OMG has issued an RFP for a MOF 2.0 Model-to-Text Transformation Language in April 2004,[3] which will eventually lead to a standard for mapping MOF-based models to text.

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

For completeness, we should also mention the concept of *text-to-model transformation*. Essentially, this category comprises traditional parsing technologies, which are beyond the scope of this paper.

Model-To-Text Approaches

Visitor-Based Approaches

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

Template-Based Approaches

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: Velocity[77] and XDoclet.[7] We have already shown an example of the template-based approach in Figure 7.

A template usually consists of the target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion (see Cleaveland[24] for an introduction to template-based code generation). 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 one.

The LHS logic accessing the source model may have different forms. The logic could be simply Java code accessing the API provided by the internal representation of the source model such as JMI, or it could be declarative queries, e.g., in OCL or XPath[88]. 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 to a visitor-based transformation, the structure of a template resembles more closely the code to be generated. Templates lend themselves to iterative development as they can be easily derived from examples. Since the template approaches discussed in this section operate on text, the patterns they contain are untyped and can represent syntactically or semantically incorrect code fragments. On the other hand, textual templates are independent of the target language and simplify the generation of any textual artifacts, including documentation.

A related technology is frame processing, which extends templates with more sophisticated adaptation and structuring mechanisms (Bassett's frames,[15] XVCL,[48] XFramer,[37] ANGIE[36]). To our knowledge, XFramer and ANGIE have been applied to generate code from models.

Model-To-Model Approaches

Direct-Manipulation Approaches

These approaches offer an internal model representation plus some API to manipulate it, such as JMI. They are usually implemented as an object-oriented framework, which may also provide some minimal infrastructure to organize the transformations (e.g., abstract class for transformations). However, users have to usually implement transformation rules, scheduling, tracing, and other facilities, mostly from scratch in a programming language such as Java.

Structure-Driven Approaches

Approaches in this category have two distinct phases: the first phase is concerned with creating the hierarchical structure of the target model, whereas the second phase sets the attributes and references in the target. The overall framework determines 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 so-called incremental copiers that users have to subclass to define their own transformation rules. The basic metaphor is the idea of copying model elements from the source to the target, which then can be 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[45]. 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 Approaches

This category groups approaches that are similar to direct manipulation but offer more dedicated support for model transformation. 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, and Kermeta. Specialized facilities such as tracing may be offered through dedicated libraries.

Figure 22 shows our sample transformation from class models to schemas expressed in the QVT Operational language. In contrast to the QVT Relations solution from Figure 4, the transformation declaration specifies the parameter modes, i.e., the transformation is executed only in one direction from `uml` to `rdbms`. The entry point for the execution is the function `main()`, which invokes the mapping `packageToSchema` on all packages and then the mapping `attributeToColumn` on all attributes contained in the input model `uml`. The mappings are defined 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.

Template-Based Approaches

Model templates are models with embedded metacode computing 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 an expression language to be used in the metalanguage is OCL.

A concrete model-template approach is given by Czarnecki and Antkiewicz.[30] In that approach, a template of a UML model, such as class or activity diagram, is created by annotating model elements with conditions and/or expressions represented as stereotypes. A very simple example is shown in Figure 23, which reuses the class model from Figure 3(a). This time, however, the model is rendered using 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 lightly colored, are removed.

```
transformation umlRdbms(
  in uml : SimpleUML,
  out rdbms : 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 {
  ...
}
...
```

Figure 22: Transformation expressed in QVT operational language

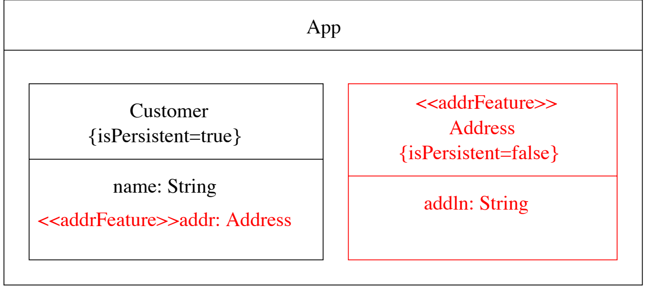


Figure 23: Example of a model template

Relational Approaches

This category groups declarative approaches where 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 Transfomation 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 non-executable (e.g., relations,[62;11] and mapping rules[13]). 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.[42] 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. We have already shown an example of the relational approach in Figure 4.

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 multidirectional rules. They sometimes also provide backtracking. Most relational approaches require strict separation between source and target models, i.e., they do not allow in-place update.

Graph-Transformation-Based Approaches

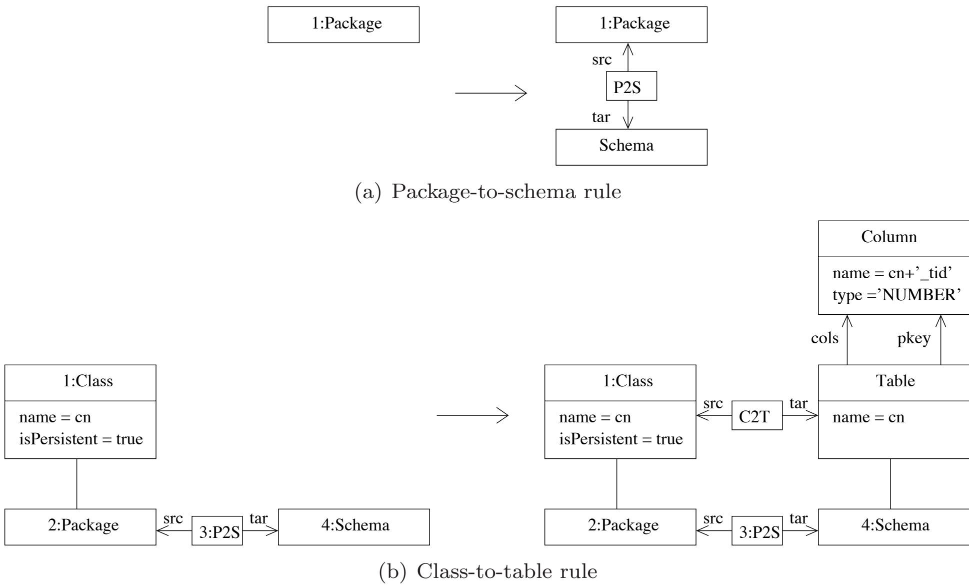
This category of model transformation approaches draws on the theoretical work on graph transformations. In particular, these approaches operate on typed, attributed, labeled graphs,[14] which can be thought of as a formal representation of simplified class models. Examples of model transformation approaches based on graph transformations include AGG, AToM^3, VIATRA, GReAT, UMLX, BOTL, MOLA, and Fujaba.

Graph transformation rules have a LHS graph pattern and a 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, e.g., negative conditions. Some additional logic, e.g., in string and numeric domains, is needed in order to compute target attribute values such as element names. GReAT offers an extended form of patterns with multiplicities on edges and nodes.

The 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 to be much more concise than patterns in the corresponding abstract syntax (see the work by Marschall and Braun[58] 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 AToM^3 are systems directly implementing the theoretical approach to attributed graphs and transformations on such graphs. They have built-in fixpoint scheduling with non-deterministic 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 24 illustrates how the transformation from class models to schemas can be expressed in AGG. Only two rules are shown. The rule in Figure 24(a) maps packages to schemas. The mapping from classes to tables is given in Figure 24(b). The mapping of attributes to columns is not shown. The RHS of an AGG rule contains a mixture of elements from the LHS, as indicated by the indices prefixing their names, and new elements. When the LHS is matched, the 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 plus the correspondence objects of `P2S`. Each rule has a negative application condition, which is implicitly assumed to be its RHS. Thanks to 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.



(a) Package-to-schema rule

(b) Class-to-table rule

Figure 24: Graph transformation in AGG

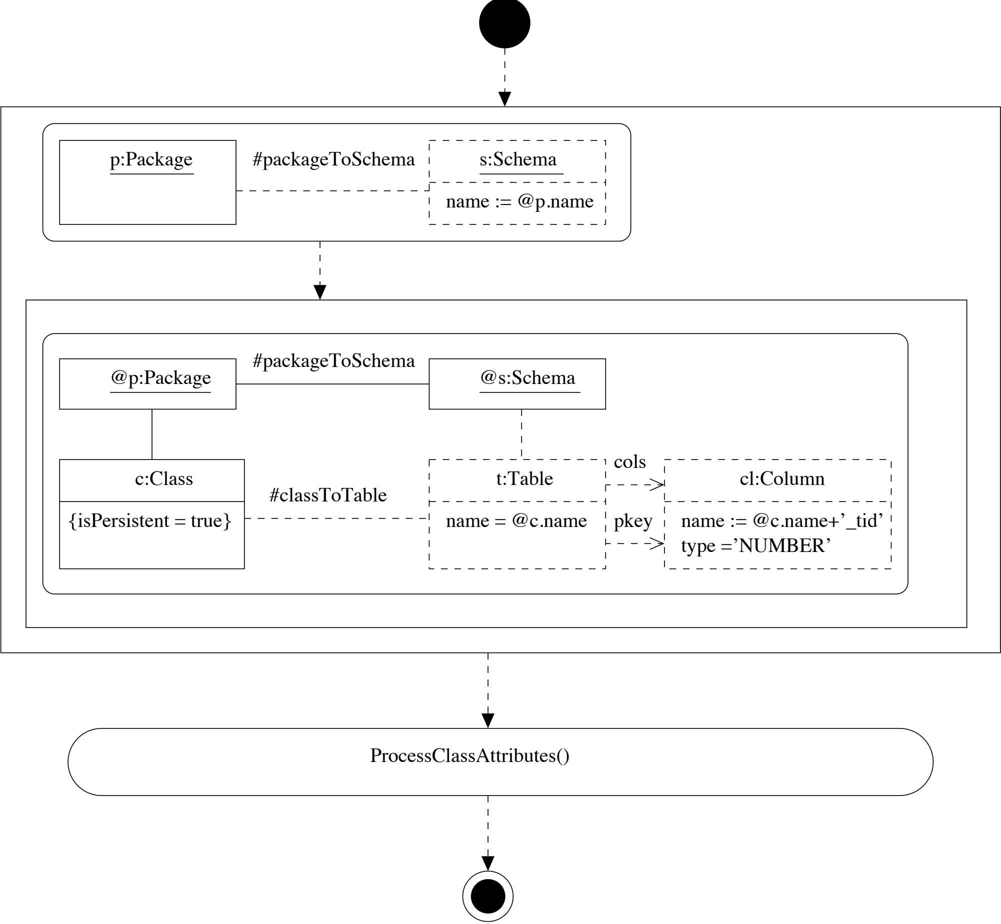


Figure 25: Graph transformation in MOLA

Systems such as VIATRA, GReAT, MOLA, and Fujaba extend the basic flavor of AGG and AToM^3 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 25. Each enclosing rectangular box represent a looping construct. Boxes with rounded corners represent looping conditions. The elements to be matched are drawn using solid line; dashed line is 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 `ProcessClassAtributes`, which is a subprogram mapping attributes to columns.

Relational-style, multidirectional approaches based on graph transformations are also possible. For example, Königs[55] discusses using a transformation approach based on triple-graph grammars to simulate QVT Relations.

Hybrid Approaches

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 and/or target pattern 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

At least two more approaches should be mentioned for completeness: transformation implemented using XSLT[86] and the application of metaprogramming to model transformation.

Since models can be serialized as XML using the XML Metadata Interchange (XMI),[8] 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, using XMI and XSLT has severe scalability limitations. Manual implementation of model transformations in XSLT quickly leads to non-maintainable implementations because of the verbosity and poor readability of XMI and XSLT. A solution to overcome this problem is to generate the XSLT rules from some more declarative rule descriptions, as demonstrated in the work by Peltier et al.[67],[66] 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.[79] His solution is a domain-specific language for model transformations embedded in a metaprogramming language.

Discussion

In this section, we offer some comments on the practical applicability of the different flavors 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. Esentially, all work has to be done by the user. The appoach 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 facilites 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 groups pragmatic approaches that were developed in the context of (and seem particularly well applicable to) certain kinds of applications such as generating EJB implementations and database schemas from UML models. These applications require 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 bult-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 different forms of incrementallity, 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 kind 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,[16] there are plenty 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, i.e., 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 are not well applicable to method bodies, where ordering is important, such as in a list of statements. Ordering can be represented by additional edges, but this approach leades 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 statements. Nevertheless, graph trasformation 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.[32] 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 Kurtev[50] 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 fourteen presented feature diagrams remained unchanged compared to the original model, namely those in Figures 11, 13, 15, 16, and 17. 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.[41] propose a unified terminology to enable a comparison of the different proposals. Since their scope of comparison is considerably different from ours, there is not much overlap in terminology. While Gardner et al. focus on the 8 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, e.g., 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 also allowing for an imperative implementation of more complex ones.

Another account of requirements for model transformation approaches is given by Sendall and Kozaczynski.[69]

Mens and Van Gorp[59] have also proposed a classification of model transformations, which they apply to graph transformation systems.[60] 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.[76]

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.

While there are satisfactory solutions for transforming models to text (such as template-based approaches), this is not the case for transforming models to models. Many new approaches to model-to-model transformation have been proposed over the last two years, but little experience is available to 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, i.e., 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 would like to thank the following people: Karl-Trygve Kalleberg for his extensive feedback on earlier versions of the feature model; Markus Völter for providing the template examples in Figure 6 and Figure 7; and Ulrich Eisenecker, Don Batory, and the anonymous reviewers for their valuable comments on a previous draft of the paper.

References

1. AndroMDA 2.0.3. `http://www.andromda.org`, 2003. viewed: October 2005.

2. Jamda: The Java Model Driven Architecture 0.2. `http://sourceforge.net/projects/jamda`. viewed: October 2005.

3. MOF 2.0 Model-to-Text Transformation Language RFP. OMG Document ad/2004-04-07, Object Management Group, Apr. 2004. Available from `http://www.omg.org/docs/ad/04-04-07.pdf`.

4. The Model-Driven Architecture Guide, Version 1.0.1. OMG Document omg/2003-06-01, Object Management Group, June 2003. Available from `http://www.omg.org/docs/omg/03-06-01.pdf`.

5. MOF QVT Final Adopted Specification. OMG Adopted Specification ptc/05-11-01, Object Management Group, Nov. 2005. Available from `http://www.omg.org/docs/ptc/05-11-01.pdf`.

6. MOF 2.0 Query / Views / Transformations RFP (revised). OMG Document ad/2002-04-10, Object Management Group, Apr. 2002. URL `http://www.omg.org/docs/ad/02-04-10.pdf`.

7. XDoclet—attribute oriented programming. `http://xdoclet.sourceforge.net/xdoclet/index.html`. viewed: October 2005.

8. XML Metadata Interchange (XMI), Version 2.1. OMG Document formal/05-09-01, Object Management Group, Sept. 2005. Available from `http://www.omg.org/docs/formal/05-09-01.pdf`.

9. Meta Object Facility 2.0 Core. OMG Document ptc/2003-10-04, Object Management Group, Oct. 2003. Available from `http://www.omg.org/cgi-bin/doc?ptc/03-10-04`.

10. 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, 2015 Terrace Place, Nashville, TN 37203, 2003. Available from `http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_11_0_2003_Graph_Tran.pdf`.

11. D. H. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002—The Unified Modeling Language 5th International Conference, Dresden, Germany, September 30–October 4, 2002. Proceedings*, volume 2460 of *Lecture Notes in Computer Science*, pages 243–258, Heidelberg, Germany, 2002. Springer-Verlag.

12. D. H. Akehurst, W. G. Howells, and K. McDonald-Maier. Kent model transformation language. In *Proceedings of Model Transformations in Practice Workshop (MTIP) at MoDELS Conference, Montego Bay, Jamaica, 2005*, Oct. 2005. URL `http://sosym.dcs.kcl.ac.uk/events/mtip05/programme.html`.

13. Alcatel, Softeam, Thales, TNI-Valiosys, and Codagen Corporation. MOF Query/Views/Transformations. OMG Document ad/03-08-05, Object Management Group, 2003. URL `http://www.omg.org/docs/ad/03-08-05.pdf`. Revised Submission.

14. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Kuske, D. Plump, A. Schürr, and G. Täntzer. Graph transformation for specification and programming. Technical Report 7/96, Universität Bremen, 1996. Available from `http://citeseer.nj.nec.com/article/andries96graph.html`.

15. P. Bassett. *Framing Software Reuse: Lessons from the Real World*. Prentice Hall, Upper Saddle River, NJ, 1997.

16. D. Batory. Multi-level models in model driven development, product-lines, and metaprogramming. Submitted., Nov. 2005.

17. J. Bézivin, G. Dupé, F. Jouault, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *Online proceedings of the OOPSLA03 Workshop on Generative Techniques in the Context of the MDA*, 2003. `http://www.softmetaware.com/oopsla2003/mda-workshop.html`.

18. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In U. Assmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linkping, Sweden, June 10-11, 2004. Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46, Heidelberg, Germany, 2005. Springer-Verlag.

19. b+m Informatik AG. open ArchitectureWare: Generator Framework. `http://architecturware.sourceforge.net`. viewed: October 2005.

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

21. R. I. Bull and J.-M. Favre. Visualization in the context of model driven engineering. In A. Pleuß, J. V. den Bergh, H. Hußmann, and S. Sauer, editors, *MDDAUI '05, Model Driven Development of Advanced User Interfaces 2005, Proceedings of the MoDELS'05 Workshop on Model Driven Development of Advanced User Interfaces, Montego Bay, Jamaica, October 2, 2005*, volume 159 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

22. 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 dInformatique Fondamentale de Lille, Universite des Sciences et Technologies de Lille, Apr. 2004. Available from `http://www.lifl.fr/evenements/publications/2004-08.pdf`.

23. CBOP, DSTC, and IBM. MOF Query/Views/Transformations. OMG Document ad/03-08-03, Object Management Group, 2003. Revised Submission.

24. C. Cleaveland. *Program Generators with XML and Java*. Prentice-Hall, Upper Saddle River, NJ, 2001.

25. Codagen Technologies Corp. Codagen Architect 3.0. `http://www.codagen.com/products/architect/default.htm`. viewed: October 2005.

26. *OptimalJ 4.0, User's Guide*. Compuware, June 2005. `http://www.compuware.com/products/optimalj`.

27. K. Czarnecki. Domain engineering. In J. Marciniak, editor, *Wiley Software Engineering Encyclopedia*, pages 433–444. Wiley and Sons, Inc., second edition, Feb. 2002.

28. K. Czarnecki. Overview of Generative Software Development. In *Proceedings of Unconventional Programming Paradigms (UPP) 2004, 15-17 September, Mont Saint-Michel, France, Revised Papers*, volume 3566 of *Lecture Notes in Computer Science*, pages 313–328. Springer-Verlag, 2004. `http://www.swen.uwaterloo.ca/~kczarnec/gsdoverview.pdf`.

29. K. Czarnecki. Foreword. Stahl and Völter [72], pages xv–xvi. In press.

30. K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Glück and M. Lowry, editors, *GPCE 2005 - Generative Programming and Component Enginering. 4th International Conference, Tallinn, Estonia, Sept. 29 – Oct. 1, 2005, Proceedings*, volume 3676 of *LNCS*, pages 422–437. Springer, 2005.

31. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

32. K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Online proceedings of the 2nd OOPSLA03 Workshop on Generative Techniques in the Context of MDA. Anaheim, October, 2003*, 2005. URL `http://swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf`.

33. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.

34. G. V. D. Varro and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, 2002.

35. J. de Lara and H. Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *FASE'02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer-Verlag, 2002. ISBN 3-540-43353-8. URL `http://atom3.cs.mcgill.ca`.

36. Delta Software Technology. Frame Processor ANGIE. `http://www.d-s-t-g.com/neu/pages/pageseng/et/common/tech` 2003. viewed: October 2005.

37. M. Emrich. Generative programming using frame technology. Diploma thesis, University of Applied Sciences Kaiserslauten, Department of Computer Science and Micro-System Engineering, Zweibrücken, 2003. URL `http://www.informatik.fh-kl.de/~eisenecker/polite/polite_emtech.pdf`.

38. J.-M. Favre. Cacophony: Metamodel-driven architecture reconstruction. In A. Pleuß, J. V. den Bergh, H. Hußmann, and S. Sauer, editors, *Working Conference on Reverse Engineering (WCRE 2004), Delft, The Netherlands, November 8th-12, 2004*, pages 204– 213. IEEE Computer Society, 2004.

39. D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

40. Fujaba. Fujaba tool suite. `http://www.fujaba.de`. University of Paderborn; viewed: December 2005.

41. T. Gardner, C. Griffin, J. Koehler, and R. Hauser. A review of OMG MOF 2.0 Query / Views / Transformations submissions and recommendations towards the final standard. OMG Document ad/03-08-02, Object Management Group, 2003. Available from `www.omg.org/cgi-bin/doc?ad/03-08-02`.

42. A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Transformation: First International Conference (ICGT 2002), Barcelona, Spain, October 7-12, 2002. Proceedings*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105, Heidelberg, Germany, 2002. Springer-Verlag.

43. J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Indianapolis, IN, 2004.

44. C. Griffin. *Model Transformtion Framework (MTF)*. IBM Hursley, available from IBM Alphaworks, 2004.

45. Interactive Objects and Project Technology. MOF Query/Views/Transformations. OMG Document ad/03-08-11, ad/03-08-12, and ad/03-08-13, Object Management Group, 2003. Revised submission.

46. Interactive Objects Software GmbH. ArcStyler 5.1. `http://www.arcstyler.com`, 2005. viewed: October 2005.

47. I. Ivkovic and K. Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2213-0.

48. S. Jarzabek. XML-based variant configuration language. `http://xvcl.comp.nus.edu.sg/overview_brochure.php`. viewed: October 2005.

49. Java Community Process. *Java Metadata Interface 1.0*. Sun, June 2002. Available from `http://java.sun.com/products/jmi`.

50. F. Jouault and I. Kurtev. Transforming models with ATL. In *Proceedings of Model Transformations in Practice Workshop (MTIP) at MoDELS Conference, Montego Bay, Jamaica, 2005*, Oct. 2005. URL `http://sosym.dcs.kcl.ac.uk/events/mtip05/programme.html`.

51. A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In U. Assmann, editor, *Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004)*, pages 12–26, Linköping, Sweden, June 2004. Research Center for Integrational Software Engineering, Linköping University. URL `http://www.ida.liu.se/~henla/mdafa2004/proceedings.pdf`.

52. K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

53. C. H. P. Kim and K. Czarnecki. Synchronizing cardinality-based feature models and their specializations. In *Proceedings of ECMDA'05*, 2005. `swen.uwaterloo.ca/~kczarnec/ecmda05.pdf`.

54. A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, MA, 2003.

55. A. Königs. Model transformation with triple graph grammars. In *Proceedings of Model Transformations in Practice Workshop (MTIP) at MoDELS Conference, Montego Bay, Jamaica, 2005*, Oct. 2005. URL http://sosym.dcs.kcl.ac.uk/events/mtip05/programme.html.

56. I. Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, Enschede, The Netherlands, May 2005. URL http://wwwhome.cs.utwente.nl/~kurtev.

57. M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *Proceedings of Model Transformations in Practice Workshop (MTIP) at MoDELS Conference, Montego Bay, Jamaica, 2005*, Oct. 2005. URL http://sosym.dcs.kcl.ac.uk/events/mtip05/programme.html.

58. F. Marschall and P. Braun. Model Transformations for the MDA with BOTL. Number TR-CTIT-03-27, pages 25–36, June 2003. Available from http://trese.cs.utwente.nl/mdafa2003.

59. T. Mens and P. Van Gorp. A taxonomy of model transformation. In *Preproceedings of the International Workshop on Graph and Model Transformation at GPCE 2005, Tallinn, Estonia, September, 2005*, pages 7–23. Institute of Cybernetics, Tallinn Technical University, 2005. URL http://www.win.ua.ac.be/~lore/refactoringProject/publications/Mens2004MtransTaxoGT.pdf.

60. T. Mens, P. V. Gorp, D. Varro, and G. Karsai. Applying a model transformation taxonomy to graph transformation technology. In *Preproceedings of the International Workshop on Graph and Model Transformation at GPCE 2005, Tallinn, Estonia, September, 2005*, pages 24–39. Institute of Cybernetics, Tallinn Technical University, 2005. URL http://www.win.ua.ac.be/~lore/refactoringProject/publications/Mens2004MtransTaxoGT.pdf.

61. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In L. C. Briand and C. Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005. ISBN 3-540-29010-9. URL http://www.kermeta.org.

62. Q. Partners. MOF Query/Views/Transformations. OMG Document ad/03-08-08, Object Managment Group, Aug. 2003. URL http://www.omg.org/docs/ad/03-08-08.pdf. Revised Submission.

63. H. Partsch. *Specification and Transformation of Programs. A Formal Approach to Software Development.* Springer-Verlag, 1990.

64. H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3): 199–236, 1983.

65. O. Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004. URL http://modeldrivenarchitecture.esi.es/pdf/paper4-1.pdf.

66. M. Peltier, F. Ziserman, and J. Bézivin. On levels of model transformation. In *XML Europe 2000, Paris, France, June 2000*, pages 1–17. Graphic Communications Association, 2000.

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

68. R. Pompa. *Java Emitter Templates (JET) Tutorial*. Azzurri Ltd., June 2005. Available from http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.

69. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231150.

70. R. Silaghi, F. Fondement, and A. Strohmeier. “Weaving” MTL model transformations. In U. Assmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linkping, Sweden, June 10-11, 2004. Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 123–138, Heidelberg, Germany, Aug. 2005. Springer-Verlag.

71. A. Solberg, R. France, and R. Reddy. Navigating the metamuddle. In *Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005)*, Montego Bay, Jamaica, 2005. URL http://www.planetmde.org/wisme-2005/NavigatingTheMetaMuddle.pdf.

72. T. Stahl and M. Völter. *Model-Driven Software Development—Technology, Engineering, Management.* John Wiley and Sons, Ltd., Chichester, England, 2006. In press.

73. G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel. Refactoring UML models. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.

74. J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.585163.

75. G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003. ISBN 3-540-22120-4.

76. G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *Proceedings of Model Transformations in Practice Workshop (MTIP) at MoDELS Conference, Montego Bay, Jamaica, 2005*, Oct. 2005. URL http://sosym.dcs.kcl.ac.uk/events/mtip05/programme.html.

77. The Apache Jakarta Project. Velocity 1.4. http://jakarta.apache.org/velocity. viewed: October 2005.

78. J.-P. Tolvanen. Making model-based code generation work. *EmbeddedSystems Europe*, pages 36–38, aug/sep 2004. URL http://i.cmpnet.com/embedded/europe/esesep04/esesep04p36.pdf.

79. L. Tratt. The MT model transformation language. In *Proceedings of ACM SAC '06, April 2006*, 2006. URL http://tratt.net/laurie/research/publications.

80. G. van Emde Boas. FUUT-je. Online. URL http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/ viewed: October 2005.

81. D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar and A. Strohmeier, editors, *UML 2004 - The Unified Modeling Language, Proceedings*, volume 3273 of *Lecture Notes in Computer Science*, pages 290–304. Springer-Verlag, 2004.

82. D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, *Proc. UML 2004: 7th International Conference on the Unified Modeling Language, Lisbon, Portugal*, volume 3273 of *LNCS*, pages 290–304. Springer-Verlag, October 10–15 2004. URL http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2004/uml2004_vp.pdf.

83. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004. URL http://www.cs.uu.nl/research/techreps/UU-CS-2004-011.html.

84. E. Visser. Program transformation taxonomy. Online, 2004. URL http://www.program-transformation.org/Transform/ProgramTransformation. Program Transformation Wiki; viewed: November 2005.

85. D. Vojtisek and J.-M. Jézéquel. MTL and Umlaut NG: Engine and framework for model transformation. Online, July 2004. URL http://www.ercim.org/publication/Ercim_News/enw58/vojtisek.html. ERCIM News No. 58, viewed: October 2005.

86. W3C. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, Nov. 1999. viewed: October 2005.

87. E. D. Willink. UMLX: A graphical transformation language for MDA. pages 13–24, 2003.

88. *XML Path Language (XPath) 2.0.* World Wide Web Consortium, 2005. `http://www.w3.org/TR/xpath20/`.

89. Xactium. Xmf-mosaic. `http://xactium.com`. December 2005.

90. J. Zhang, Y. Lin, and J. Gray. Generic and domain-specific model refactoring using a model transformation engine. In S. Beydeda, M. Book, and V. Gruhn, editors, *Model-driven Software Development*, chapter 9, pages 199–218. Springer, 2005.