Title: Sans titre
Author:

CHAPTER 8 Model-to-Model Transformations

Models are dynamic entities that undergo various processes during an MDE process. These processes include *merging*, *aligning*, *refactoring*, *refining*, and *translating* models.

All these model operations are implemented as model transformations, either Model-to-Model (M2M) or Model-to-Text (M2T) transformations. In M2M, the input and output are models; in M2T, the output is a text string. Text-to-Model (T2M) transformations have a text string as input and a model as output; they're typically applied in reverse engineering (cf. Chapter 3).

Many efforts have been spent designing specialized languages for specifying M2M transformations, ranging from textual to visual, declarative to imperative, and semi-formal to formal, in the last decade. We review most of them in the next sections and focus on two protagonists to illustrate their main characteristics.

> **Résumé** :
>
> Models are dynamic entities undergoing ***merging***, ***alignment***, ***refactoring***, ***refinement***, and ***translation*** within an MDE process.
>
> Model transformations, implemented as Model-to-Model (M2M) or Model-to-Text (M2T), convert models to models or text strings, respectively. Text-to-Model (T2M) transformations, used in reverse engineering, convert text strings to models.
>
> Various M2M transformation languages have been designed, ranging from textual to visual and declarative to imperative. Two languages are highlighted to illustrate their characteristics.

8.1 MODEL TRANSFORMATIONS AND THEIR CLASSIFICATION

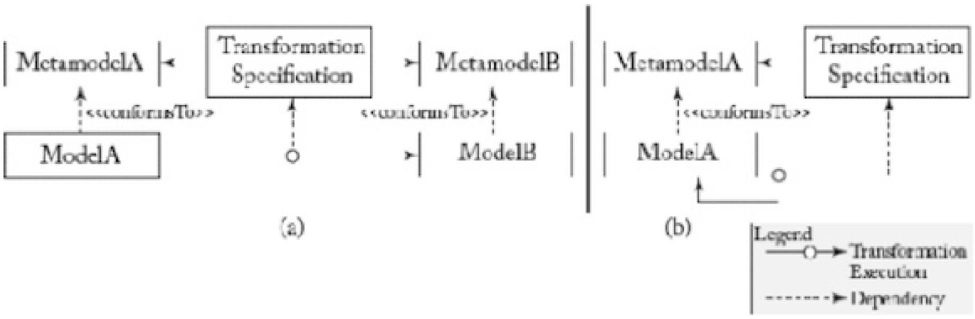
Transformations are a key software engineering technique since the introduction of high-level programming languages compiled to assembler. Model transformations are equally crucial in MDE and come in various flavors to address different tasks.

An M2M transformation is a program that takes one or more models as input and produces one or more models as output. *One-to-one* transformations are usually sufficient, but *one-to-many*, *many-to-one*, or *many-to-many* transformations may be required, for example, in a model merge scenario to unify multiple class diagrams.

Besides classifying model transformations based on input and output models, another dimension is whether the transformation is between models from different languages (*exogenous*) or within the same language (*endogenous*). An example of an exogenous transformation is transforming a platform-independent model (e.g., a UML model) to a platform-specific model (e.g., a Java model). An example of an endogenous transformation is model refactoring, which involves restructuring models to improve quality.

Exogenous model transformations are useful for both *vertical* and *horizontal* transformations. Vertical transformations change the abstraction level of the input and output models, as seen in the UML to Java scenario. Horizontal transformations change the abstraction level of the input and output models, but they remain similar. For example, horizontal transformations are used for model exchange between modeling tools, such as translating a UML class diagram to an ER diagram.

Two central execution paradigms for model transformations emerged in the last decade, as compared in Figure 8.1. First, *out-place* transformations *generate* the output model from scratch (Figure 8.1a), suitable for exogenous transformations. Second, *in-place* transformations *rewrite* the model by creating, deleting, and updating elements in the input model (Figure 8.1b), perfect for endogenous transformations like refactorings.



We present how to specify *exogenous* transformations as *out-place* transformations using ATL and *endogenous* transformations as *in-place* transformations using graph transformation languages.

> **Résumé** :
>
> - Model transformations are crucial in MDE, solving various tasks through different techniques.
> - M2M transformations convert models into other models, with ***one-to-one*** transformations being the most common, but ***one-to-many***, ***many-to-one***, and ***many-to-many*** transformations are also possible.
> - Model transformations are classified based on input/output models and language. ***Exogenous*** transformations involve different languages, while ***endogenous*** transformations occur within the same language, such as model refactoring.
> - Exogenous model transformations are useful for both ***vertical*** and ***horizontal*** transformations, enabling model exchange between different modeling tools.
> - Two model transformation paradigms emerged: ***out-place*** transformations for ***generating*** output models and ***in-place*** transformations for ***rewriting*** input models.
> - ***Exogenous*** transformations are specified as ***out-place*** transformations using ATL, while ***endogenous*** transformations are specified as ***in-place*** transformations using graph transformation languages.

8.2 EXOGENOUS, OUT-PLACE TRANSFORMATIONS

ATLAS Transformation Language (ATL) is the illustrative transformation language for developing exogenous, out-place transformations. It’s widely used in academia and industry, with mature tool support. ATL is a rule-based language that builds on OCL but provides dedicated features for model transformations missing in OCL, like creating model elements.

ATL, a hybrid model transformation language, combines declarative and imperative constructs. It's *uni-directional*, requiring two transformations for language A to B and vice versa. ATL transformations operate on *read-only* source models and produce *write-only* target models. For out-place transformations, we use *source model* for *input model* and *target model* for *output model*.

During ATL transformations, source models are queried but not modified. Target model elements are created but not queried directly. These restrictions ensure that query results from source and target models remain consistent regardless of the transformation's execution state, which contradicts the declarative nature of ATL.

Anatomy of ATL transformations. An ATL transformation is represented as a *module* with a *header* and a *body* section. The header section specifies the module name and declares the source and target models, which are typed by their metamodels. ATL transformations can have multiple input and output models.

The ATL transformation body consists of *rules* and *helpers* stated in arbitrary order after the header section.

- **Rules**: Each rule describes how a part of the target model should be generated from a part of the source model. ATL has two kinds of declarative rules: *matched* rules and *lazy* rules. Matched rules are automatically matched by the ATL execution engine, while lazy rules require explicit calling from another rule, giving the transformation developer more control over the execution.
  Rules consist of *input* and *output* patterns. The input pattern filters source model elements relevant to the rule by defining input pattern elements. Each input pattern element requires an obligatory type (corresponding to a metaclass in the source metamodel) and an optional filter condition (expressed as an OCL expression). These constraints determine the applicable model elements. The output pattern details how the target model elements are created from the input. Each output pattern element can have multiple *bindings* to initialize the features of the target model elements. Binding values are calculated by OCL expressions.
- **Helpers**: A helper is an auxiliary function that enables the possibility of factoring ATL code used in different points of the transformation. There are two types of helpers: one simulates a derived attribute that needs to be accessible throughout the transformation,

and the other represents an operation that calculates a value for a given context object and input parameters. Unlike rules, helpers cannot produce target model elements; they can only return values that are further processed within rules.

Execution phases of ATL transformations. ATL, a powerful language, allows us to define transformations concisely. To understand its syntax, let's examine the actual transformation execution by the ATL virtual machine. ATL transformations are executed in three sequential phases.

Phase 1: Module initialization. In the first phase, the *trace model* for storing *trace links* between source and target elements is initialized. In phase 2, each matched rule execution creates a trace link pointing to the matched input and output elements. The trace model is crucial for exogenous transformations: (i) stopping transformations and (ii) assigning target element features based on source element values.

Phase 2: Target elements allocation. In the second phase, the ATL transformation engine finds valid configurations of source model elements that match the source pattern of the matched rules. When a matched rule's condition is fulfilled by a configuration, the engine allocates the corresponding target model elements based on the declared target pattern elements. Only the elements are created, not their features. Each match creates a trace link connecting the matched source elements to the generated target elements.

Phase 3: Target elements initialization. In the third phase, each allocated target model element is initialized by executing the bindings defined for the target pattern element. These bindings often invoke the *resolveTemp* operation, which allows references to any target model elements generated in the second execution phase for a given source model element. The resolveTemp operation has the signature `resolveTemp(srcObj: OclAny, targetPatternElementVar: String)`. The first parameter represents the source model element for which target elements need to be resolved, while the second parameter is the variable name of the target pattern element to retrieve. This parameter is required because multiple target pattern elements can be used in a rule to generate multiple target elements for a single source element.

Internal vs. external trace models. The internal trace model of the ATL transformation engine enables convenient setting of target element features and stopping rule execution. Rules are only executed if the matched input elements are not covered by an existing trace link.

If external traceability requires a persistent trace model, e.g., to see the impact of source model changes on the target model, the internal trace model (defaultly transient) may be persisted as a separate output model of the transformation.

Alternatives to ATL. Besides ATL, there are other dedicated transformation approaches for exogenous, out-of-place transformations.

QVT. The OMG's Query-View-Transformation (QVT) standard supports three languages for model transformation development: QVT Relational, QVT Operational, and QVT Core. QVT Relational defines bi-directional correspondences between metamodels, enabling the derivation of transformations in both directions and consistency checks. QVT Operational is an imperative language for uni-directional transformations with out-of-the-box tracing support, but transformation rule dispatching requires user definition, similar to lazy rules in ATL. QVT Core is a low-level language designed for the QVT Relational compiler, not for direct transformation writing. Eclipse provides tool support for QVT Relations, QVT Core, and QVT Operational languages.

TGG. Triple Graph Grammars (TGGs) define a correspondence graph between two metamodels, enabling bidirectional model transformations. They synchronize models and check consistency. Similar to QVT Relations, TGGs offer a strong theoretical basis. MOFLON and Eclipse support TGGs through the TGG Interpreter and the TGG tool at MDElab.

ETL. The Epsilon project developed several languages for model management. The Epsilon Transformation Language (ETL) supports out-of-place model transformations similar to ATL, but it also offers features like modifying input model elements during transformations.

> **Résumé** :
>
> - ATLAS Transformation Language (ATL), a widely used rule-based transformation language, is chosen for developing exogenous transformations due to its industry and academic adoption and mature tool support.
> - ATL is a hybrid model transformation language with declarative and imperative constructs. It performs ***uni-directional*** transformations on ***read-only*** source models to produce ***write-only*** target models.
> - During ATL transformations, source models are queried but not modified, while target models are created but not queried directly to maintain declarative consistency.

> - An ATL transformation is a ***module*** with a ***header*** and ***body*** section. The header declares the transformation module name and source/target models, which can be multiple.
> - ATL transformation body consists of ***rules*** and ***helpers***.
>   - ATL has two rule types: ***matched*** and ***lazy***. Matched rules are automatically executed, while lazy rules require explicit calls, providing more control.
>   - Rules consist of ***input*** and ***output*** patterns. The input pattern filters source model elements using metaclass types and OCL expressions, while the output pattern defines target model elements and their feature initialization.
>   - Helpers are auxiliary functions in ATL code, enabling code factorization. They simulate derived attributes or represent operations, returning values processed within rules.
> - ATL transformation execution consists of three sequential phases.
>   - Phase 1 initializes the ***trace model***, storing ***trace links*** between source and target elements. Phase 2 stores each matched rule execution in the trace model.
>   - In Phase 2, the ATL transformation engine finds source model element configurations that match source patterns, allocating corresponding target model elements and creating trace links.
>   - Phase 3 initializes target model elements by executing bindings defined for target pattern elements. The ***resolveTemp*** operation is used to reference generated target model elements.
> - The internal trace model in ATL is crucial for setting target element features and stopping rule execution based on existing trace links.
> - Persistent trace model may be needed for external traceability, such as tracking source model changes.
> - Other transformation approaches exist besides ATL for exogenous transformations.
>   - The OMG’s Query-View-Transformation (QVT) standard includes three languages for model transformations: QVT Relational (declarative, bi-directional), QVT Operational (imperative, uni-directional), and QVT Core (low-level, compiler target). Eclipse provides tool support for all three languages.
>   - Triple Graph Grammars (TGG) define correspondence graphs between metamodels, enabling model transformation, synchronization, and consistency checks. TGGs are supported by MOFLON and Eclipse.
>   - The Epsilon Transformation Language (ETL) supports out-place model

> transformations and allows modifying input model elements.

8.3 ENDOGENOUS, IN-PLACE TRANSFORMATIONS

Models were manually modified in modeling editors by adding, removing, or updating elements. However, many scenarios require automated modifications. Out-place transformations require building the output model from scratch, copying the source model, and deleting or modifying elements. In-place transformations provide changes without copying static elements, requiring only transformation rules for dynamic changes.

Graph transformation is an elegant approach to in-place model transformations. It's selected for demonstrating their development and usage.

Graph Transformation Basics. Graph transformation is a declarative, rule-based technique for expressing in-place model transformations. Models and meta-models can be expressed as graphs with typed, attributed nodes and edges, allowing for model manipulation using graph transformation techniques.

Graph transformations are useful for defining in-place transformations, supporting model simulation, optimization, execution, evolution, and refactoring. However, they are so general that out-place transformations could be formulated by representing the source, target, and trace models as one integrated graph.

Graph transformations are popular due to their visual form (intuition) and formal nature (analysis). They can describe the operational semantics of modeling languages for implementing model execution engines. This allows the use of the abstract and sometimes concrete syntax of the language to define the transformation rules, making them intuitive to designers.

A *graph grammar* consists of transformation rules and an initial graph (*host graph*) to which the rules apply. A rule has a left-hand side (LHS) graph expressing pre-conditions and a right-hand side (RHS) graph containing post-conditions. The actions of the rule are implicitly defined by both sides. Execution of a transformation rule produces the following effects: (i) all elements in the LHS are deleted; (ii) all elements in the RHS are added; and (iii) all elements in both sides are preserved. To mark equivalence between elements in the LHS and RHS, they must have the same identifier.

To apply a rule to the initial graph, a *morphism* (also called *occurrence* or *match*) of the LHS must be found in it. If multiple matches are found, one is selected randomly. The rule is then applied to the selected match by substituting it with the RHS. The grammar execution proceeds by applying the rules in non-deterministic order until none are applicable. Other graph transformation systems provide more control over the transformation execution. More details are presented later.

Graph transformation rules can be applied fully automatically by the graph transformation engine in arbitrary order. However, some cases require an interactive execution mode. For example, if a refactoring needs to be applied to a specific model element, the user must select the element as an input parameter for the transformation rule. Some graph transformation approaches allow for pre-bound LHS elements by providing explicit user input before executing transformation rules. Additionally, some approaches allow users to provide additional input during transformation execution, such as setting values for features of model elements that cannot be derived from the current model state.

Advanced graph transformations techniques.

Several advanced techniques exist for specifying, executing, and analyzing graph transformations.

Alternative notations. Transformation rules have been defined in the abstract syntax of the modeling language. Using the concrete syntax may improve readability, but only a few graph transformation tools provide this capability. Some approaches use a condensed graphical notation by merging Negative Application Conditions (NACs), LHS, and RHS into one graph with annotations for forbidden, preserved, deleted, or created elements. Others use a textual concrete syntax to define the rules. We discuss these approaches in the end of this section.

Rule scheduling. A graph transformation system consists of rules. The order in which they're executed is non-deterministic until no rule matches. This can lead to multiple output models. However, a deterministic execution is preferred. Two extensions support this. First, define *priorities* for rules. If multiple rules match, the highest/lowest priority rule is executed. Second, use programming-like control structures, called graph transformation units, to *orchestrate* rules. These structures allow for loops, conditional branches, and so on. These approaches are called *programmable graph transformations*.

Analysis. Graph transformations, a declarative approach based on a strong theoretical basis, offer several analysis methods. For non-deterministic systems, determining if the same unique model is produced regardless of rule order is crucial. Critical pairs, mutually exclusive rules that hinder or enable each other's application, can be computed. Additionally, some approaches allow reasoning about the termination of graph transformation systems.

Tool Support. Several Eclipse projects focus on providing graph transformation technologies for EMF. These projects support diverse features for implementing and executing graph transformations. Thus, the choice of graph transformation tool depends on the specific transformation problem.

Henshin, successor to EMF Tiger, introduces advanced features like programmable graph transformations and model checking support. It also integrates with AG, enabling advanced graph transformation techniques like computing critical pairs between transformation rules.

Fujaba, a protagonist of programmable graph transformation approaches, orchestrates graph transformation rules using story diagrams (similar to UML activity diagrams). Java is used to define application conditions for the LHS and assignments for the RHS within these rules.

e-Motions, an Eclipse plugin, uses graph transformation rules to graphically specify the behavior of modeling languages. It allows specifying time-related attributes for rules, such as duration or periodicity. Once the behavior of a modeling language is defined in e-Motions, models can be simulated and analyzed by translating them and the transformation to Maude, a programming framework based on rewriting logic.

ATL Refining. *ATL Refining* adds a new execution mode for in-place transformations. Activate it by replacing the *from* keyword with *refining* in an ATL transformation header. In-place transformation rules are syntactically like out-place rules, but their execution differs. If an output pattern element shares a variable name with an input pattern element, it's updated by bindings but not created. New output pattern elements (without input counterparts) are generated by the rule. An output pattern element can be annotated with *drop* to delete it and all its contained elements. ATL Refining is an in-place transformation language with a textual syntax.

> **Résumé** :
>
> - In-place transformations are more efficient for automated model modifications than out-place transformations, as they only require rules for dynamic changes, avoiding

> the need to copy unmodified elements.
>
> - ***Graph transformations*** are used to demonstrate in-place model transformations.
> - Graph transformation is a declarative, rule-based technique for expressing in-place model transformations.
> - Graph transformations are useful for in-place model transformations, supporting simulation, optimization, execution, evolution, and refactoring. They can also be used for out-place transformations by representing source, target, and trace models as an integrated graph.
> - Graph transformations are popular for their visual and formal nature, making them intuitive and amenable to analysis. They can describe operational semantics of modeling languages.
> - A ***graph grammar*** consists of transformation rules and an initial graph. The rules, defined by left-hand side and right-hand side graphs, specify actions that delete, add, or preserve elements based on their presence in the left-hand side (LHS) and right-hand side (RHS).
> - A morphism of the LHS is found in the initial graph, and a rule is applied to the selected match by substituting it with the RHS.
> - Graph transformation rules can be applied automatically or interactively, with user input required for selecting model elements and setting feature values.
> - Advanced graph transformation techniques are discussed.
>   - Alternative notations for defining transformation rules include concrete syntax, condensed graphical notation, and textual concrete syntax, each with varying levels of readability and tool support.
>   - Two extensions support deterministic execution of graph transformation systems: rule ***prioritization*** and ***programmable graph transformations*** using control structures like loops and conditional branches.
>   - Graph transformation systems have analysis methods, including critical pair computation for non-deterministic systems and termination reasoning.
> - Several Eclipse projects support graph transformation technologies for EMF, each with diverse features. The choice of tool depends on the specific transformation problem.
>   - Henshin, successor to EMF Tiger, introduces advanced features like programmable graph transformations and model checking support, with integration with AG for graph transformation techniques.
>   - Fujaba uses story diagrams to orchestrate graph transformation rules, with Java

> used for defining application conditions and assignments.
> - E-Motions is an Eclipse plugin that uses graph transformation rules to specify modeling language behavior, enabling simulation and analysis through Maude.
> - ***ATL Refining***, a new execution mode for ATL, enables in-place transformations using a textual syntax. It updates existing elements, generates new ones, and deletes elements with the ***drop*** keyword.

CHAPTER 9 Model-to-Text Transformations

Several concepts, languages, and tools have been proposed in the last decade to automate text derivation from models using Model-to-Text (M2T) transformations. These transformations have been used to automate software engineering tasks like generating documentation and task lists.

M2T transformations are primarily used for code generation, which is the main goal of model-driven software engineering. Current execution platforms are mostly code-based, with a few exceptions that allow direct model interpretation. M2T transformations focus on generating code to transition from the model level to the code level. Not only system code can be derived from models, but also other code-related artifacts like test cases and deployment scripts. Formal code descriptions can also be derived for analyzing system properties. Models offer both constructive derivation of the system and analytical exploration or verification of system properties, making them a valuable tool for both construction and analysis. M2T transformations serve as the bridge between execution platforms and analysis tools.

This chapter introduces model-based code generation using M2T transformations, discusses various approaches to implementing them, and provides techniques for mastering code generator complexity.

> **Résumé** :
>
> - Model-to-Text (M2T) transformations automate text derivation from models, streamlining software engineering tasks like documentation and task list generation.
> - Model-to-text (M2T) transformations are crucial for code generation, enabling the

> transition from models to running systems. M2T transformations can also generate test cases, deployment scripts, and formal code descriptions for system analysis.
> - This chapter covers model-based code generation using M2T transformations, implementation approaches, and complexity management techniques.

9.1 BASICS OF MODEL-DRIVEN CODE GENERATION

Code generation has a long history in software engineering, dating back to the early days of high-level programming languages. However, compilers and MDE have different goals for code generation. In compilers, code generation transforms source code into machine code. In MDE, it transforms models into source code. MDE code generation relies on existing compilers for programming languages.

Three essential questions for developing a model-based code generator are:

- **How much is generated?** The main question is which parts of the code can be automatically generated from models. Is *full* or *partial* code generation possible? Partial can mean one layer of the application is completely generated while another is manually developed, or one layer is partially generated with missing parts requiring manual completion. Partial code generation may also refer to the model level, where code generation is used for certain parts while others are manually implemented.
- **What is generated?** It’s crucial to determine the source code to generate. The code should be concise, and it should be readable by developers, as emphasized in Chapter 3’s Turing test for code generators. Therefore, exploit current high-level programming languages, APIs, and frameworks to avoid reinventing the wheel. Choose the right programming paradigm to minimize code generation and system representation.
- **How to generate?** Once the requirements for code generation are specified, including which parts are generated and which target languages are used, it’s decided *how* to implement these requirements. Several languages, from GPLs to DSLs, can be used to generate code from models.

Code generation is the vertical transition from higher-level models to lower-level artifacts. MDE benefits heavily from this transition, so code generators must bridge this abstraction gap in various ways.

Closing the gap between models and code. When a modeling language is used to build models, not all underlying technologies may be expressible. MDE aims to abstract from technology details. For instance, if defining a UML class diagram for a domain, you may need to introduce attributes of type String. If deploying the diagram in a space-limited environment, you may define a restricted length for these attributes instead of the maximum possible String length. As in the MDA approach, additional information may be needed to achieve executable software.

Information can be provided by the modeler using model augmentations, the convention-over-configuration principle for code generation, or leaving the specification open on the model level and filling details at the code level. The first approach allows tweaking details for the derived implementation, requiring more effort in preparing models. The second approach doesn't need this effort but optimizations may only be done on the code level. A hybrid approach combines convention-over-configuration with manual tweaking by model augmentations. Using the third approach, only partial implementations are generated, requiring developers to complete them on the code level. Special care must be taken, as models and code contradict the single source of information principle. Countermeasures include protected areas in the code requiring manual code or explicit extension points. Base classes can be generated and extended by user-defined classes for missing functionality that can't be specified on the model level.

> **Résumé** :
>
> - Code generation in MDE transforms models into source code, utilizing existing compilers for programming languages.
> - Three questions are essential for developing a model-based code generator.
>   - The main question is which parts of the code can be automatically generated from models, and if full or partial code generation is possible.
>   - Code generators should produce concise, readable code using current high-level languages, APIs, and frameworks to avoid reinventing the wheel. The goal is to generate the least amount of code possible.
>   - Code generation requires specifying requirements, including generated parts and target languages, and deciding on the implementation approach.
> - Code generation bridges the gap between higher-level models and lower-level artifacts, enabling the benefits of Model-Driven Engineering (MDE).
> - Modeling languages abstract from technology details, requiring additional

> information for executable software.
>
> - Code generation from models can be achieved through model augmentations, convention-over-configuration, or by leaving details open and filling them in at the code level. The first approach allows for more control over the derived implementation, while the second approach is more straightforward but limits optimization.

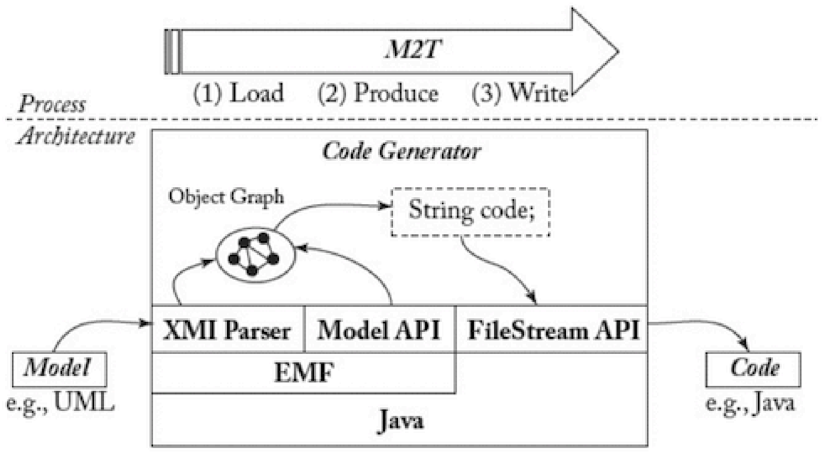
9.2 CODE GENERATION THROUGH PROGRAMMING LANGUAGES

A code generator can be implemented based on MDE principles or a traditional programming approach. In the latter case, it's a program that uses the model API generated from the metamodel to process input models and print code statements to a file using standard stream writers provided by the programming language's APIs.

The model API is implemented in EMF using an M2T transformation that reads an Ecore-based metamodel and generates a Java class for each Ecore class. The Ecore-to-Java mapping is straightforward, a design goal of Ecore. For each metaclass feature, corresponding getter and setter methods are generated on the Java side. This allows reading, modifying, and creating a model from scratch using generated Java code instead of modeling editors. For more information on using the generated model APIs and the powerful EMF API, refer to the interested reader.

Before discussing specific M2T transformation languages in Section 9.3, we show how a GPL can be used to develop a code generator. This demonstrates: (i) processing models using a model API generated from the metamodel and (ii) features needed for a code generator.

The following phases must be supported by a code generator, as illustrated in Figure 9.1.



1. **Load models**: Models must be deserialized from their XMI representation to an in-memory object graph. Current metamodeling framework APIs offer specific operations for this.
2. **Produce code**: Use the model API to process models and collect the necessary model information for generating code. Typically, the object graph is traversed from the root element of a model to its leaf elements.
3. **Write code**: Code is saved in String variables and persisted to files via streams.

This approach requires no additional programming skills. It suffices to know the chosen programming language and the model API. No additional tools are needed for design or runtime. However, it has drawbacks.

- **Intermingled static/dynamic code:** *Static code*, generated for every model element, and *dynamic code* derived from model information, such as class and variable names, are not separated.
- **Non-graspable output structure:** The code generator specification doesn't clearly show the output structure. The generated code is embedded in the producing code, so the control structure is explicit but the output format isn't. This issue is also seen in other GPL-based generator approaches, like Java Servlets1 for producing HTML code with embedded statements.
- **Missing declarative query language:** No declarative query language exists for accessing model information. This leads to unnecessary code from iterators, loops, conditions, and type casts. Also, knowledge of the generated model API is required. For instance, getter methods must be used to access model element features instead of querying feature values using metamodel names.
- **Missing reusable base functionality:** Code must be developed to read input models

and persist output code repeatedly for each code generator.

DSLs, developed to eliminate disadvantages, generate text from models. This led to the OMG standard *MOF Model to Text Transformation Language* (MOFM2T). We show how a Java-based code generator can be re-implemented in a dedicated M2T transformation language and discuss its benefits.

> **Résumé** :
>
> - Code generators can be implemented using MDE principles or a traditional programming approach. The latter approach involves using a program to process input models and generate code statements.
> - The model API is realized in EMF using an M2T transformation that generates Java classes from Ecore classes, allowing for direct model manipulation.
> - A GPL is used to develop a code generator, demonstrating model processing and required features.
> - Code generator must support phases illustrated in Figure 9.1.
>   - Models are deserialized from XMI to an in-memory object graph.
>   - Model API processes models to generate code.
>   - Code is saved in String variables and persisted to files.
> - The approach requires only programming language knowledge and model API familiarity, eliminating the need for additional programming skills or tools. However, it also has several drawbacks.
>   - Static and dynamic code are intermingled, lacking separation between code generated for every model element and code derived from model information.
>   - The output structure of the code generator is not easily graspable due to embedded code. This issue is also present in other GPL-based generator approaches.
>   - Lack of declarative query language leads to excessive code and reliance on generated model API knowledge.
>   - Code generators require repeated development for input model reading and output code persistence.
> - DSLs and ***MOF Model to Text Transformation Language*** (MOFM2T) were developed to generate text from models, improving upon previous Java-based code generators.

9.3 CODE GENERATION THROUGH M2T TRANSFORMATION LANGUAGES

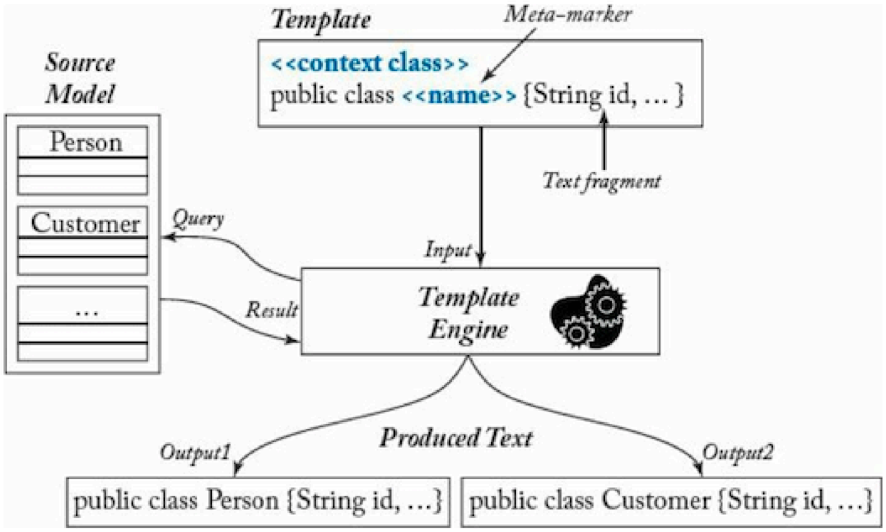
This section shows how M2T transformation languages ease code generator development, overviews existing protagonists, and demonstrates how to use one to implement a code generator.

> **Résumé** :
>
> - M2T transformation languages simplify code generator development, and existing languages are discussed.

9.3.1 BENEFITS OF M2T TRANSFORMATION LANGUAGES

A Java-based code generator was presented in the previous section to illustrate how necessary features must be implemented in GPLs. M2T transformation languages aim to improve code generator development by addressing the drawbacks of GPL-based code generators.

Separated static/dynamic code. M2T transformation languages separate static and dynamic code using a *template*-based approach. Templates define static text elements shared by all artifacts and dynamic parts filled with case-specific information. Templates contain simple text fragments for the static part and *meta-markers* for the dynamic part. Meta-markers are placeholders interpreted by a *template engine* that processes templates and queries additional data sources (models) to produce dynamic parts. Figure 9.2 summarizes template-based text generation.



Explicit output structure. Templates explicitly represent the structure of output text by embedding code for dynamic parts in the static part, reversing the previous Java-based code generator. This mechanism, similar to Java Server Pages (JSPs), allows explicit representation of HTML page structures and embedded Java code, unlike Java Servlets. Explicitly represented output structures lead to more readable and understandable code generation specifications than using String variables. Templates also ease code generator development. For instance, developers can create templates by adding example code and substituting dynamic parts with meta-markers (i) simplifying the abstraction process from concrete code examples to template specifications and (ii) templates have a similar structure and format as the code to be produced, enabling traceability of template effects to the code level.

Declarative query language. We need to access information stored in models within meta-markers. As presented, OCL is the choice for most M2M transformation languages. Current M2T transformation languages also allow us to use OCL (or a dialect of OCL) for specifying meta-markers. Template languages not tailored to models but supporting various sources use standard programming languages like Java for specifying meta-markers.

Reusable base functionality. Current M2T transformation languages provide tool support for reading models and serializing text into files using configuration files. This eliminates the need for manual redefinition of model loading and text serialization.

> **Résumé** :
>
> - M2T transformation languages improve code generator development by addressing drawbacks of GPL-based generators.

> - M2T transformation languages use a ***template***-based approach to separate static and dynamic code. Templates contain static text elements and ***meta-markers***, which are interpreted by a ***template engine*** to generate dynamic parts from models.
> - Templates explicitly represent output text structure, embedding dynamic code within static text. This approach, similar to Java Server Pages (JSPs), enhances code generator readability and development ease.
> - OCL is used to access model information in M2M transformation languages, while other template languages use standard programming languages like Java.
> - M2T transformation languages offer tool support for model reading and text serialization, eliminating manual development.

9.3.2 TEMPLATE-BASED TRANSFORMATION LANGUAGES: AN OVERVIEW

Various template-based languages can be used to generate text from models.

- **XSLT**: The XMI serializations of the models can be processed with XSLT, the W3C standard for transforming XML documents into arbitrary text documents. However, the code generation scripts require implementing based on the XMI serialization, which necessitates knowledge of how models are encoded as XML files. Therefore, approaches directly operating on the model level are more favorable.
- **JET**: The Java Emitter Template (JET) project was one of the first approaches for developing code generation for EMF-based models. JET is not limited to EMF-based models. It transforms every Java-based object to text. JET provides a JSP-like syntax for writing templates for M2T transformations. Arbitrary Java expressions can be embedded in JET templates. JET templates are transformed to pure Java code for execution. However, JET lacks a dedicated query language for models.
- **Xtend**: Xtend, a modern programming language, is mainly based on Java but offers additional features. It supports code generation with template expressions and functional programming, which is beneficial for querying models (many OCL iterator-based operations are available out-of-the-box).
- **MOFScript**: This project offers another M2T transformation language with features similar to Xtend. MOFScript, a candidate proposal for the OMG standardization effort, provides a standardized language for M2T transformations. It’s available as an Eclipse plug-in and supports EMF-based models.
- **Acceleo**: This project aims to provide a practical version of the OMG’s M2T

transformation standard for EMF-based models. It supports full OCL querying and has mature tool support, useful in industry.

> **Résumé** :
>
> - Template-based languages generate text from models.
>   - XSLT can process XMI serializations of models, but model-level approaches are more favorable.
>   - Java Emitter Template (JET) enables code generation for EMF-based models and any Java-based object. It uses a JSP-like syntax for templates, allowing arbitrary Java expressions.
>   - Xtend7 is a Java-based programming language with additional features like code generation and functional programming.
>   - MOFScript is an M2T transformation language similar to Xtend, developed as an OMG standardization proposal. It is available as an Eclipse plug-in and supports EMF-based models.
>   - Acceleo is a pragmatic version of the OMG M2T transformation standard for EMF-based models. It provides full OCL support and mature tool support.

9.3.3 ACCELEO: AN IMPLEMENTATION OF THE M2T TRANSFORMATION STANDARD

Acceleo is selected as a protagonist to demonstrate M2T language transformation due to its practical relevance and mature tool support. Other M2T transformation languages like Xtend and MOFScript also support Acceleo's language features.

Acceleo offers a template-based language for defining code generation templates. It supports OCL and additional operations for working with text-based documents, including advanced string manipulation functions. Acceleo includes powerful tooling such as an editor with syntax highlighting, error detection, code completion, refactoring, debugger, profiler, and a traceability API for model element tracing to generated code and vice versa.

Before defining templates in Acceleo, a container module is created to hold them. This module imports the metamodel definition for the templates, making them aware of the metamodel classes that can be used as template types. Templates in Acceleo are always defined for a specific metamodel class. In addition to the model element type, a pre-condition can be defined, similar to a filter condition in ATL, to apply a template only to model elements with a specific type and values.

The Acceleo template language has several meta-markers, called tags in Acceleo, that are also common in other M2T transformation languages.

- **Files**: To generate code, files must be opened, filled, and closed, as in the Java-based code generator. In Acceleo, a special `file` tag prints the content between its start and end. The tag's path and file name are defined by its attributes.
- **Control Structures**: There are tags for defining control structures like loops (`for` iterating over collections of elements) and conditional branches (`if`).
- **Queries**: OCL queries, similar to ATL helpers, can be defined with a `query` tag. They can be called throughout the template to factor out recurring code.
- **Expressions**: Expressions are used to compute values for the dynamic parts of the output text. They can also call other templates to include their generated code in the caller template's code. Calling other templates is similar to method calls in Java.
- **Protected Areas**: M2T languages support projects with partial code generation. Special protection is needed to safeguard manually added code from file modifications in subsequent generator runs. Acceleo's protected tag supports protected areas, which mark sections in generated code that shouldn't be overridden again. These sections usually contain manually written code.

> **Résumé** :
>
> - Acceleo is used to demonstrate M2T transformation languages due to its practical relevance and mature tool support.
> - Acceleo is a template-based language for code generation with a powerful API, tooling, and traceability.
> - Acceleo templates require a module as a container, importing the metamodel definition. Templates are defined for specific metamodel classes, with optional pre-conditions for filtering model elements.
> - Acceleo template language uses tags, common in M2T transformation languages.
>   - Acceleo uses a special `file` tag to print content between start and end tags for a given file, with path and filename defined by attributes.
>   - Control structures include loops for iteration and conditional branches for navigation.
>   - OCL queries, similar to ATL helpers, can be defined and called throughout templates.

>   - Expressions compute values for dynamic output text and call other templates, similar to Java method calls.
>   - Protected areas in M2T languages safeguard manually added code from subsequent generator runs. Acceleo supports this through the protected tag.

9.4 MASTERING CODE GENERATION

To address the complexity of model-based code generators, several advanced techniques are briefly described below.

Abstracting templates. Abstracting code generation templates from concrete reference examples is a proven approach to ensure that generated code is accepted by developers, especially when only partial code generation is applied. Acceleo offers the possibility to refactor a concrete reference example to a template, with several refactoring operations to substitute example code with tags. For instance, a concrete Java class name can be refactored to an expression that queries the name value from a model element. This tool support allows for systematic extraction of templates from example code, ensuring developers are familiar with the produced output.

Reusing templates. Current M2T transformation languages offer features for structuring and reusing transformation logic. Templates can be nested in an inheritance tree, enabling polymorphic template calls based on model element type. This is useful in metamodels where inheritance is used to reuse features. Templates can be reused and extended for subclasses defined for superclasses to avoid duplicated code. M2T languages also support aspects for defining and weaving cross-cutting concerns to code generators or refining complete transformation modules by overriding templates instead of extending them by inheritance.

Step-by-step generation. Dividing a code generation process into several steps by chaining transformations is useful when there’s a significant gap between the model and code levels. This leads to verbose templates that hide the final output structure. Applying the divide-and-conquer principle with transformation chains (cf. Section 8.4) results in more readable, reusable, and maintainable components. For instance, if a code generator for flat state machines exists, an M2M transformation from nested state machines to flat state machines can be provided for reuse.

Separating transformation logic from text. Refactor complex computations, such as queries on the input model or complex String computations, into queries or libraries that can be imported in templates when needed. This makes the final output structure more visible in templates and previous code reusable in other templates or code generators.

Mastering code layout. The code layout is determined by the template. It can be challenging to produce the desired layout when several loops and conditions are in the template. Xtend allows us to have a code beautifier as a postprocessing unit for the code generator output. Besides using a code beautifier, there are language features in M2T languages that control the actual layout of the output, such as operators for ignoring line breaks in templates and special signs produced in loops.

Be aware of model/code synchronizing problems. Protected areas are helpful for partial code generation, but manually added code may not be placed correctly in the next generator run. Especially when using class-method names as protected area IDs, renaming refactorings on the class and method on the model level makes synchronization impossible. Such refactorings result in a new Java file with a « new » operation, requiring manual copying, addition, and adaptation of the implementation code. Some M2T transformation frameworks, like Acceleo, report if manually added code cannot be merged with the new code. In such cases, the code generator's log file summarizes the unintegrated lines of code.

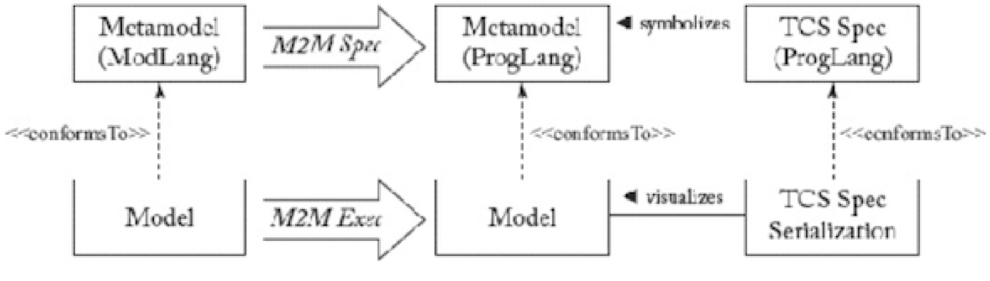
Be cautious when refactoring a model with partial code generation. One approach is to execute model-level refactorings and corresponding code-level refactorings, then run the code generator after propagation. Another approach is to use immutable internal IDs of model elements for protected area IDs. The former approach requires more effort to build model/code synchronization support, but it can automatically adapt manually added code to the new model version. The latter approach doesn't require additional effort.

> **Résumé** :
>
> - Advanced techniques can be used to manage complexity when implementing model-based code generators.
>   - Acceleo offers a tool to refactor concrete code examples into templates, ensuring generated code is accepted by developers. This approach is especially useful for partial code generation.
>   - M2T transformation languages support template reuse through inheritance, aspects, and template overriding, enabling code generation with cross-cutting concerns and module refinement.
>   - Dividing code generation into steps using transformation chains improves

    readability, reusability, and maintainability. This approach, exemplified by a nested state machine generator, allows for modular development and code reuse.
    - Complex computations should be refactored into queries or libraries for better visibility and reusability in templates and code generators.
    - Code layout in M2T languages is determined by templates, with challenges arising from complex loops and conditions. Language features like operators for ignoring line breaks and separating elements aid in controlling output layout.
    - Model/code synchronization issues arise when manually added code cannot be placed correctly in newly generated code, especially after class and method renaming. This can lead to manual code copying and adaptation.
- Refactoring models with partial code generation requires careful consideration. Two approaches are proposed: synchronizing model and code refactorings or using immutable model element IDs for protected areas.

9.5 EXCURSUS: CODE GENERATION THROUGH M2M TRANSFORMATIONS AND TCS

If the target language of an M2T transformation is supported by a metamodel and a TCS, we can use an M2M transformation instead. Figure 9.3 illustrates the resulting process. First, we transform the metamodel of the modeling language to the metamodel of the text-based language. Then, we serialize the target model into text using the mapping between the abstract syntax and the TCS.



This approach seems beneficial: (i) Reusing the textual concrete syntax definition ensures valid textual artifacts; (ii) An explicit metamodel for the text-based language allows better reasoning about language concepts and mappings; (iii) Transformation definitions can be validated based on the metamodels; (iv) Output models representing text-based artifacts can be validated against the target metamodel; and (v) Changes in the target language's concrete syntax are independent of the transformation, so no additional code generator maintenance is needed.

However, this approach has limitations. Most available metamodels for programming languages lack TCS definitions or are incomplete. Text-based languages are known only by their TCS, making reasoning on abstract syntax concepts time-consuming, unlike code generation templates. Copying and substituting code with meta-markers doesn't support abstraction from reference code to transformation definitions, unlike M2T transformations. Partial code generation features like protected areas are not covered. For code generators, only a restricted part of the target language is needed, which is easier to implement with M2T transformations without reasoning on the complete language definition.

From a pragmatic viewpoint, it's easier to start with building a code generator using M2T transformation languages, especially when no metamodel and TCS definitions for the target language exist. The M2M/TCS approach for code generation should only be considered when the target language is supported by a metamodel, TCS, and a full code generation approach.

> **Résumé** :
>
> - M2T transformations can be realized using M2M transformations when the target language is supported by a metamodel and TCS.
> - Reusing textual concrete syntax ensures valid artifacts, allows reasoning about language concepts, and enables validation of transformation definitions and output models. Changes to the target language's concrete syntax are independent of the transformation.
> - Current metamodels for programming languages lack TCS definitions, and text-based languages require learning time for abstract syntax reasoning. Additionally, the approach lacks support for partial code generation scenarios and only requires a restricted part of the target language.
> - M2T transformation languages are more practical for code generation, especially when target languages lack metamodel and TCS support. The M2M/TCS approach is only viable when a full code generation approach is followed.