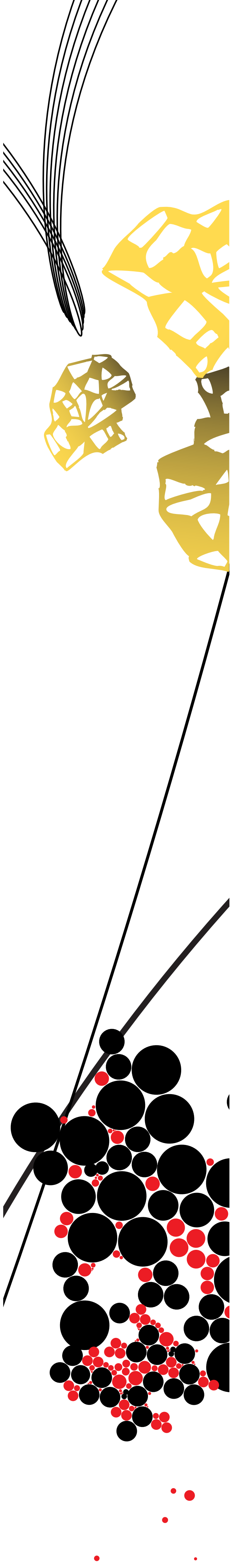


MASTER'S THESIS

Towards a Unifying Framework for Modelling and Executing Model Transformations

Ivo van Hurne

18th June, 2014

Supervisors: dr. L. Ferreira Pires
dr. C.M. Bockisch

UNIVERSITY OF TWENTE.

Faculty of EEMCS
Department of Computer Science

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of

Master of Science in Computer Science

at the

University of Twente

Enschede, June 2014

Abstract

Model-driven engineering is a software engineering technique which relies heavily on the use of models. They are not just used as documentation, but actually define the system. Transformations on the models are described using model transformation languages. There are a lot of different model transformation languages available, all having a different approach to model transformation.

We show that these languages are actually not that different at all. Based on the analysis of a varied selection of transformation languages we define a small number of primitive transformation operations that can be used to describe model transformations written in any transformation language. We define our own primitive transformation language, using just these operations, and verify our analysis by implementing a few well-known transformation languages in our own language, including a language not considered in the initial analysis.

We design an interpreter for our primitive language and show that the execution of model transformations with our interpreter is on par with their original interpreter.

Acknowledgements

I would like to thank a few people for their support during the writing of this thesis. First of all, I would like to thank Luís Ferreira Pires and Christoph Bockisch for taking over the supervision of the project. Without their help it probably would not have been possible to complete this thesis. I would also like to thank my fellow students at the SE-lab for their valuable feedback and support.

I'm indebted to Edwin Vlieg and Joost Diepenmaat for giving me the final push necessary to finish the project. Finally, I'm grateful to my family and friends for their never-ending support.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1. Motivation

The *Model-Driven Architecture* (MDA) [6] was introduced in 2001 by the Object Management Group (OMG) to aid the integration, interoperability and evolution of software systems. MDA is an approach to specifying software systems that separates specification of functionality from its implementation on a specific technology platform.

Kent [7] introduced the term *Model-Driven Engineering* (MDE), which in a sense extends MDA. MDE is not only concerned with the separation of platform-independent from platform-specific, but also with the development process and extra dimensions like versioning. Contrary to MDA, MDE is not necessarily based on OMG standards. Instead it uses general standard-independent modelling concepts.

MDE relies heavily on the use of models. They are not just used as documentation but define the system at a high abstraction level, omitting any technology-specific information. A model can describe, for example, the structure or the behaviour of a system. The actual implementation of the system can be generated from such a model via a model transformation. Transformations can also be used for reverse engineering (implementation to model) and the transfer of information between systems.

In MDE, model transformations are usually defined using a *model transformation language*. Research has led to the creation of several transformation languages and at present new languages continue to appear.

New model transformation languages can be useful to test and demonstrate novel ideas, and to specify languages tailored for a given domain. However, the development of these languages is a time-consuming process, as most of them are implemented from scratch and as such they are difficult to extend and adapt.

1.2. Objectives

Although the languages all have their own particular approach and features, they have the same main purpose. This means they are likely to share some common features. For example, in all transformation languages, models can be manipulated and navigated in some way (e.g. using OCL [8]). Languages often share common structures like transformation rules, and may provide control over the execution order of these rules.

The objective of this work is to create a unifying framework for executing transformation languages that exploits the commonalities between a set of languages, while being able to execute of each one of them. This would enable faster prototyping of new ideas, easier reuse of (parts of) transformations and the composition of transformations written in different languages.

1.3. Research Questions

> *How can we execute different model transformation languages in a common environment, while retaining the variability between the languages?*

This question has been decomposed into three subquestions:

1. What should a common execution environment for model transformation languages look like?
2. How do we build support for executing different transformation languages in this common environment?
3. What are the limitations of executing transformation languages in a common environment?

1.4. Research Approach

- We first take a look at the principles of models and model transformation languages. Out of the vast amount of transformation languages available, we select a few representative languages for a more detailed analysis.
- Second, we investigate the execution algorithms of these languages. We look into the commonalities and variabilities, which results in a set of primitive transformation operations.
- Third, we define a language for specifying transformation languages in our common environment.
- Fourth, we implement an interpreter for our language in order to execute the specified languages.
- Finally, we present a number of test cases to validate our framework and identify its limitations.

1.5. Outline

This thesis is organized as follows. Chapter 2 introduces model transformation and transformation languages. Chapter 3 analyses the commonalities and variabilities between the languages, and take a look at their execution algorithms. Chapter 4 defines our language for specifying transformation languages in our framework. Chapter 5 introduces our implementation of the framework and its architecture. Chapter 6 discusses test cases for validating the implementation and its limitations. Chapter 7 draws conclusions about the results and sketches future work.

2 Model Transformation Languages

We start with an introduction to model transformation and provide an overview of existing transformation languages.

2.1. Models

We mentioned before that models are the cornerstone of MDE. The word *model* is used to describe lots of different concepts in different contexts. We will use the definition by Kurtev [9], which summarizes some of the existing definitions: "A model represents a part of the reality called the object system and is expressed in a modelling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system."

Most models conform to a *metamodel*, which is a model of a modelling language and basically describes its abstract syntax. Because a metamodel is itself a model, it is an instance of another metamodel (also known as a *meta-metamodel*).

2.2. Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a framework for modelling and code generation of applications based on structured data models [5]. In EMF, metamodels are defined as instances of the *Ecore model*. Additionally, the Ecore model itself is defined as an Ecore model.

Figure 2.1 shows the main part of the Ecore model [1]. EClasses are used to define classes along with their attributes (EAttribute) and references (EReference). EPackages can be used to group the classes.

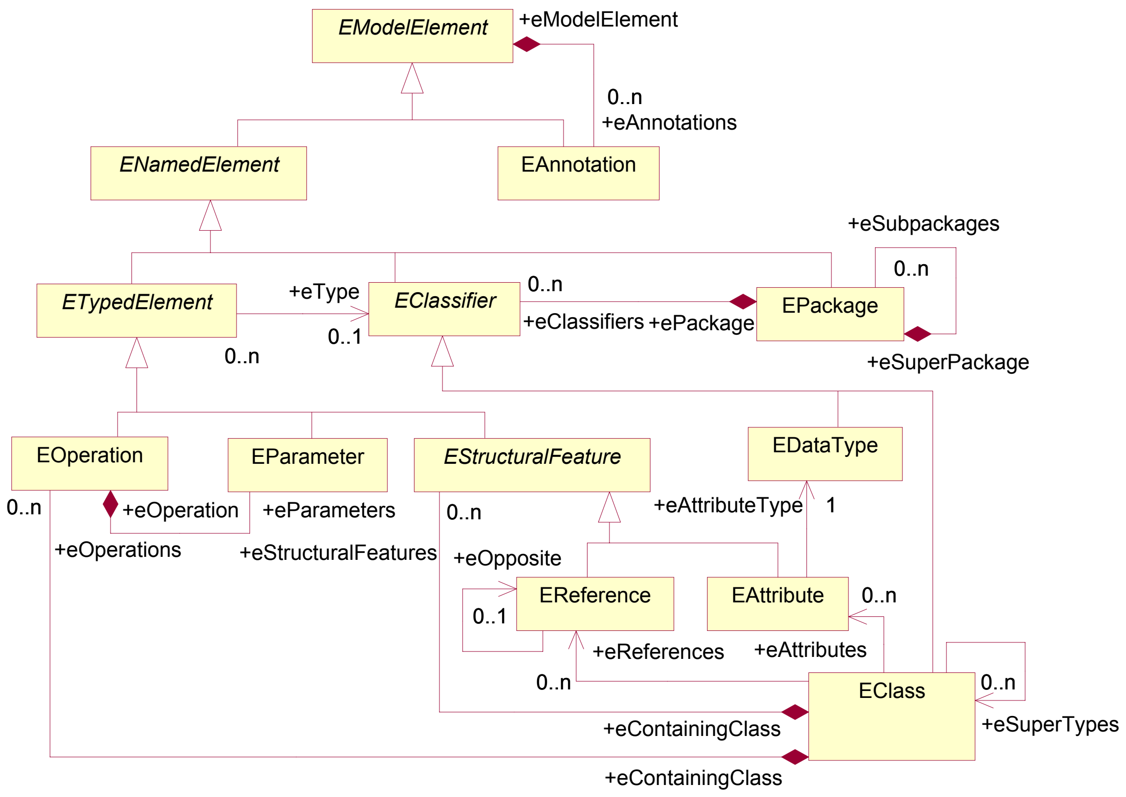


Figure 2.1.: Kernel of Ecore model [1]

EMF provides basic Java code generation for its models, including a simple editor, support for basic model modification operations and persistence.

2.3. Model Transformation

In the Object Management Group's MDA Guide, model transformation is defined as "the process of converting one model to another model of the same system" [6]. However, this definition does not include the specification of a transformation and unnecessarily restricts transformations to models of the same system. Additionally, it does not take into account transformation to and from multiple models.

We therefore use the definition by Mens and van Gorp [10]: "A model transformation is a process of automatic generation of one or multiple target models from one or multiple source models, according to a transformation definition, which is expressed in a model transformation language."

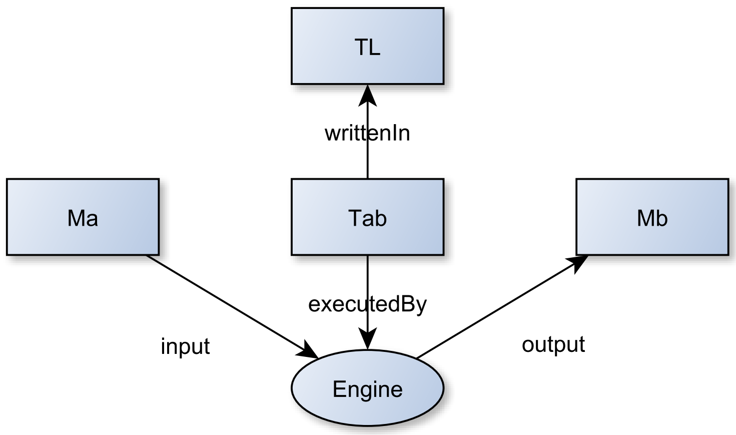


Figure 2.2.: Model Transformation Pattern

2.4. Model Transformation Pattern

Model transformations can be described using the pattern shown in Figure 2.2. We have a source model M_a, a target model M_b and a transformation definition T_{ab}. The transformation definition is written in a transformation language *TL*. When executed, T_{ab} transforms M_a to M_b.

As mentioned before multiple source and target models can be used in the same transformation, extending the pattern accordingly.

As a model is an instance of a metamodel, we can extend the pattern to include the metamodels (Figure 2.3). Some transformation languages do not need metamodels because they operate on generic graphs or ASTs (e.g. GROOVE, Stratego). Other languages do need metamodels.

2.5. Transformation Languages

There are many different model transformation languages available, but they can nevertheless be classified into a number of categories. Because we are mainly interested in the execution algorithms of model transformation languages, we consider four aspects for classification: *transformation approach*, *rule application strategy*, *model representation* and *tracing* [11].

2.5.1. Transformation Approach

Mens and van Gorp [10] state that the most important distinction between the languages is whether they use a *declarative* or an *operational* approach.

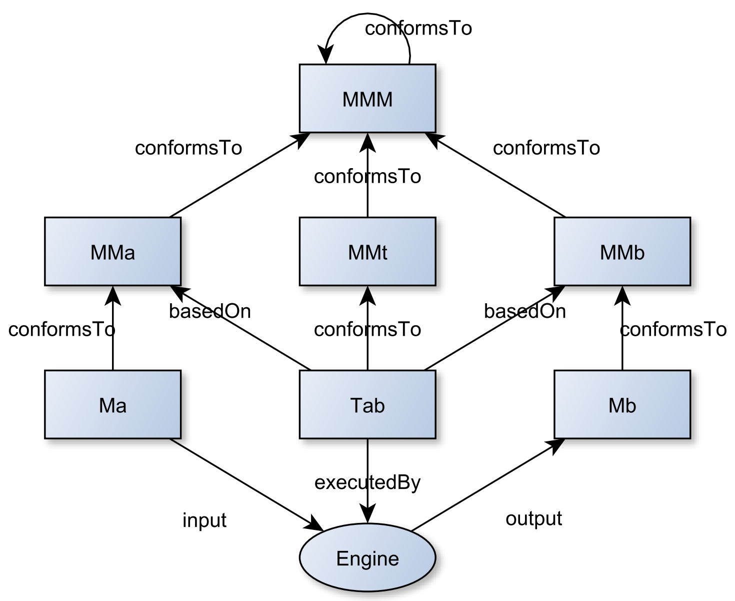


Figure 2.3.: Model transformation pattern (including metamodel) [2]

Declarative Approach The declarative approach is also known as the *relational* approach. Developers specify relations between elements of the source and target models. The transformation language has to figure out the necessary transformation steps by itself. Examples of declarative languages are Henshin [3] and QVT Relations [12].

Imperative Approach The imperative approach is also known as the *operational* approach. Developers specify the exact operations that have to be performed to transform source models to target models. An example of an imperative transformation language is QVT Operational Mappings [12].

Hybrid Approach A third approach has yet to be mentioned: the *hybrid* approach. Strictly speaking this is not an approach separate from declarative or imperative, but simply makes both available to the developer in a single language. An example of a hybrid language is ATL [13].

2.5.2. Rule Application Strategy

A transformation usually consists of a number of units. In most languages these units are rules [11]. Rules can appear in many guises, for example a rewrite rule with a left-hand side and a right-hand side; or a function that takes an input pattern and produces output by using an expression.

The rules in a transformation can often be applied in different orders. Furthermore, rules can match at multiple model regions at the same time. Transformation languages therefore use a strategy to choose the order in which transformation rules should be applied and at which model region. There are two kinds of strategies [11]:

Deterministic A deterministic strategy can either be *explicit* or *implicit*. In the first case the application order is defined by an explicit strategy, possibly separate from the transformation rules themselves (example: Stratego [14]).

In the second case the developer has but indirect control over the application order. The result of the transformation is deterministic, but can be changed by changing the matching patterns and logic of the rules (example: ATL [13]).

Non-deterministic In a non-deterministic strategy a rule and/or application location is chosen randomly from matching rules or model regions (example: graph transformation languages [15]). It is important to note that even if the application strategy is non-deterministic, the result of the transformation can still be deterministic, in case all the different application orders result in the same output model.

2.5.3. Model Representation

Models used by transformation languages can be represented in various ways. Some possibilities are graphs (e.g. graph transformation), abstract syntax trees (e.g. Stratego) and Eclipse Modelling Framework models [16] (e.g. Henshin, ATL).

2.5.4. Tracing

Traces record information about the execution of a transformation. Commonly traces are used to map source elements to target elements [11]. They can be used for analysing how changes in the source model affect the target models, for synchronizing models and for debugging.

There are various types of traces:

- Different kinds of information can be recorded, such as, the rule that created a trace or the time of creation.
- Information can be recorded at different abstraction levels, for example, only for top-level transformations.
- Information can be recorded for different scopes, for example, only for particular transformation rules.
- Traces can be stored in a number of locations, such as, in the source model, target model or in a separate location.

Some transformation languages provide support for tracing in the language itself, or even create traceability links automatically (ATL, QVT). In other languages tracing has to be done manually as part of the transformation definition.

2.6. Languages to Consider

We choose to analyse five declarative model transformation languages. Based on the classification above, they have significantly different characteristics and they are well-known representatives of their kind.

Graph transformation Not really a language as such, but rather a class of transformation languages with graph-based models, a non-deterministic rule application strategy and no inherent tracing support.

GROOVE Graph transformation language extended with explicit deterministic rule application strategy. No inherent tracing support.

Henshin Eclipse modelling framework-based language with explicit deterministic rule application strategy. No inherent tracing support.

Stratego Abstract syntax tree-based term-rewriting language with explicit deterministic rule application strategy. No inherent tracing support.

ATL EMF-based language with implicit deterministic rule application strategy and inherent tracing support. Although ATL is a hybrid language we only consider its declarative part.

We discuss these languages in detail in Chapter 3. We choose only declarative languages, so we can validate our analysis later on using a language with a different transformation approach (imperative ATL).

2.7. Conclusions

We have briefly introduced models and model transformation concepts. We have shown that there are many model transformation languages with different characteristics. Furthermore we have explained our choice of languages for analysis. Next, we discuss the execution algorithms of the chosen languages.

3 Transformation Execution Algorithms

In Chapter 2 we have selected five transformation languages for further analysis: graph transformation, GROOVE, Henshin, Stratego and declarative ATL. In this chapter we take a look at the way they are executed and try to find commonalities between them.

3.1. Graph Transformation

The first execution algorithm we consider is the algorithm used in graph transformation. In this kind of transformation a model is described as a graph. There are multiple ways to do this, the most simple being the mapping of entities to nodes and relations to edges. However, once more complicated relations are used (e.g. multiplicity) more complicated mappings are necessary [17]. This, however, is outside the scope of this thesis.

To transform a model, graph rules can be defined that specify pre-conditions for the application of a rule (called *left-hand side*) and post-conditions that have to be satisfied after the application of a rule (*right-hand side*) [15].

There are actually a number of different approaches to graph transformation that can be used. The way in which new elements are added and old elements are removed is slightly different for these approaches, but the general idea is the same. In this thesis we illustrate graph transformation using the *single-pushout approach* (SPO).

Algorithm 3.1 describes the approach in pseudo-code. We first select an arbitrary transformation rule. We try to find an arbitrary occurrence of its left-hand side L (Figure 3.1a) in a graph G (Figure 3.1b). This happens in line 10 of Algorithm 3.1 using `G.getMatches()`. During the matching that is taking place here, any possible negative application conditions are also taken into account. Negative application conditions are conditions under which a rule should not be applied. For example, one could add a condition to the rule in Figure 3.1a that prevents guests from occupying multiple rooms.

Once matches have been found, we iterate over them in a non-deterministic order. All graph elements that exist in L but not in R are then deleted from G (lines 12-14). Subsequently, we check for any dangling edges in G, and delete them (lines 16-19). Finally in lines 21-23, we add the objects that exist in R but not in L to G (Figure 3.1c) [18].

If a transformation rule has been applied successfully, the transformation is restarted and we again start to look for matches for an arbitrary transformation rule. If no matches were found for a transformation rule we try a different rule. The transformation ends when none of the transformation rules can be applied.

Because of the non-deterministic choice of rules there is no guaranteed confluence. This means executing the transformation multiple times on the same model does not necessarily yield the same result. Moreover, termination of a transformation is not guaranteed and a rule might not be executed even if there exists a match for this rule in the graph.

3.2. Controlled Graph Transformation

In a graph transformation, rules are self-contained units that specify the exact pre-conditions for their application [19]. Rule ordering is often a major reason for specifying these conditions. Because the conditions can be very complex and rules can implicitly depend on other rules, the transformation is not always easy to understand. A possible solution is to move the complex pre-conditions out of the rules, specifying them using so-called control expressions. In other words: an explicit deterministic rule application strategy is introduced.

3.2.1. GROOVE

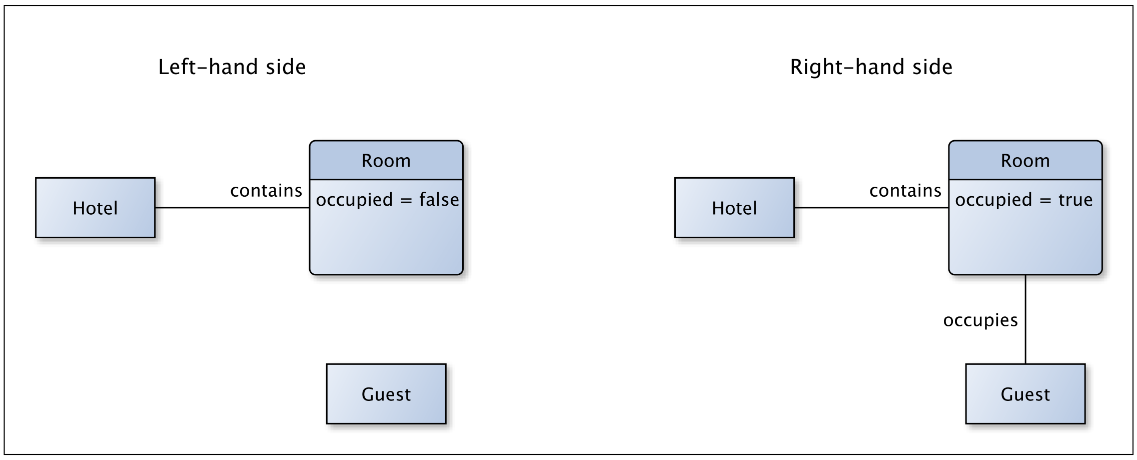
Staijen has defined a language for specifying such strategies for the GROOVE graph-transformation toolkit [20].

Table 3.1 lists the language constructs available in this language. The constructs take one or more expressions, which can be either rules or other constructs.

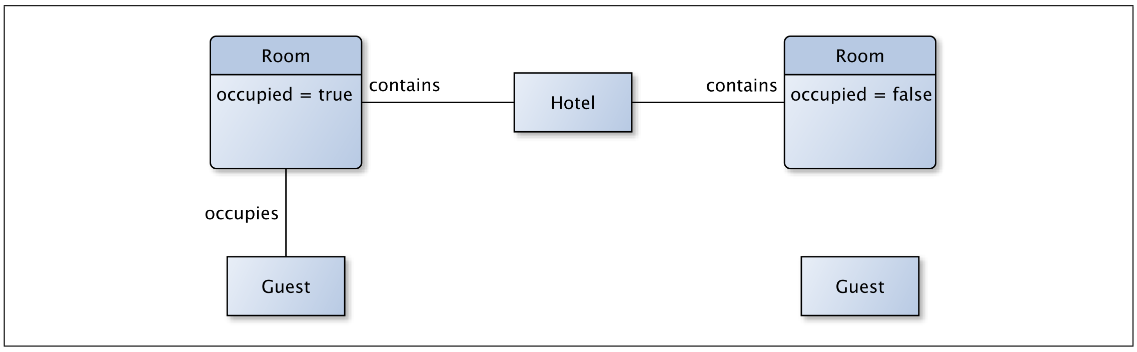
3.2.2. Henshin

Another language for graph transformations with explicit rule application strategy is Henshin [21], developed by Arendt et al. [3]. Henshin operates on EMF models, but the models are represented as graphs.

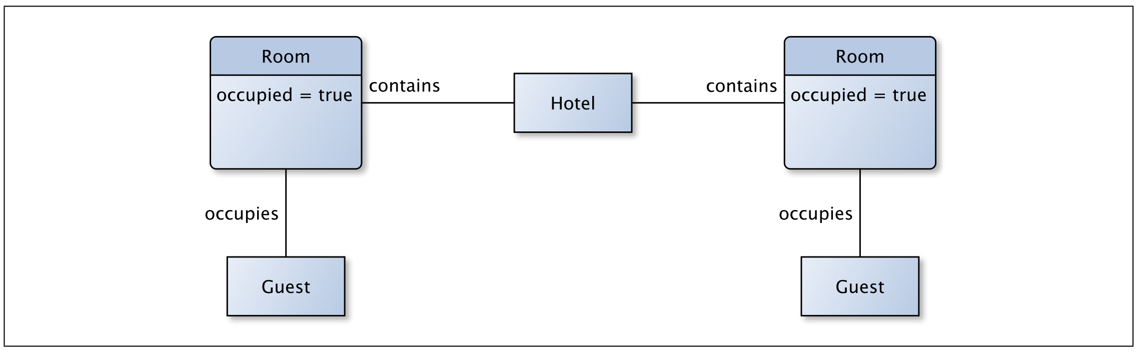
The application strategy is defined using *transformation units* (Figure 3.2). A rule is itself the most basic transformation unit. Other kinds of transformation units can have subunits that are executed in a particular order, or under particular circumstances. A transformation unit can be applicable, which means it will be executed during the transformation. Arendt et al. discuss some of these transformation units in their articles:



(a) Graph transformation rule



(b) Graph before transformation application



(c) Graph after transformation application

Figure 3.1.: Simple graph transformation example

Algorithm 3.1 Pseudo-code representation of single-pushout graph transformation.

```
InputModel G
Transformation transformation

// Continue until no application is possible
while True:
    boolean ruleApplied = false

    forall rule in transformation.getRules():
        // Find occurences of rule in graph
        forall oL in G.getMatches(rule.getL()):
            // Find elements in $L \setminus R$
            Set deletedElements = oL - rule.getR()
            // Delete elements from G, creating D
            Model D = G - deletedElements
            // Check for dangling edges
            forall e in D.edges():
                if e.source == null || e.target == null:
                    // Delete edge if dangling
                    D = D - e
            // Find elements in $R \setminus L$
            Set addedElements = rule.getR() - oL
            // Glue new elements to D, creating new G (H)
            G = D + addedElements
            ruleApplied = true
            break

        if ruleApplied:
            // rule applied, start from beginning
            break

    if !ruleApplied:
        // no rules left, stop execution
        break
```

Table 3.1.: Language constructs in GROOVE control language

Construct	Description
`ruleName`	Execute a rule.
`true`	Behaves like a rule that is always successful and does not change the underlying structure.
`E1 | E2`	Non-deterministic choice. Execute either `E1` or `E2`.
`E1 ; E2`	Sequential composition. First execute `E1`, then `E2`.
`E*`	Execute `E` an arbitrary number of times.
`alap E`	Repeat the execution of `E` as long as it applies.
`try E1`	Execute `E1`, skip if it does not apply.
`try E1 else E2`	First try to execute `E1`, then execute `E2` only if `E1` fails.
`if (E1) E2`	First execute `E1`, then afterwards `E2` if `E1` succeeds.
`if (E1) E2 else E3`	First execute `E1`, then afterwards `E2` if `E1` succeeds. If `E1` fails, execute `E3`.
`while (E1) do E2`	Execute `E1` and afterwards `E2`, and again `E1` until the execution of `E1` fails.
`until (E1) do E2`	Try to execute `E1`, then execute `E2` and again `E1` if `E1` fails.

Rule A transformation rule [3]. It is applicable if there is a match for its left-hand side in the model.

IndependentUnit Non-deterministic execution of its subunits. An `IndependentUnit` is always applicable. Subunits may be applied repeatedly. The `IndependentUnit` terminates if no applicable subunit remains [3].

PriorityUnit Execution of a subunit that is applicable and has the highest priority. A `PriorityUnit` is always applicable. Subunits may be applied repeatedly. The `PriorityUnit` terminates if no applicable subunit remains [22].

SequentialUnit Execution of its subunits in a predefined sequence. A `SequentialUnit` is only applicable if if all subunits are applicable in the given order. The `SequentialUnit` terminates if all subunits terminate [3].

CountedUnit Execution of its subunit a specified number of times. It is only applicable if its subunit is applicable the given number of times. It terminates if the subunit terminates [22].

ConditionalUnit Conditional execution using references to subunits called `if`, `then`, and `else`. It terminates if its subunits terminate [3].

AmalgamationUnit Allows for the definition of multiple rules with common parts. It consists of a *kernel rule* and a set of *multi-rules* [3].

- The kernel rule is matched only once and serves as a common partial match for each multi-rule.
- The multi-rules are matched as often as possible.

Algorithm 3.2 describes the execution algorithm for Henshin in pseudo-code. Henshin supports the passing of parameters between transformation units, but we omit this in the pseudo-code.

3.3. Term Rewriting

The next transformation language we discuss is called Stratego. It was developed by Visser [14] and is based on term rewriting. Instead of graphs, term-rewriting transformations use *abstract syntax trees* (AST) to represent the model. Abstract syntax trees consist of *terms*, i.e. applications $C(t_1, \ldots, t_n)$ of a constructor C to terms t_i, lists, strings or integers.

An AST can be transformed by replacing its terms with other terms in a manner specified by term rewriting rules. However, as with graph transformation, a term rewriting system does not necessarily terminate, or yield the same result if rules are applied in a different order (*confluence*). Therefore, extra rules are often added to specify where rules should be applied and in what order.

Algorithm 3.2 Pseudo-code representation of Henshin's execution algorithm.

```
Transformation transformationSystem

forall unit in transformationSystem:
    processUnit(unit)


function processUnit(unit):
    if unit instanceof Rule:
        Match match = unit.findMatch()
        if match != null:
            unit.applyRule(match)

    else if unit instanceof IndependentUnit:
        // Works in the same way as the
        // basic graph transformation algorithm

    else if unit instanceof PriorityUnit:
        TransformationUnit nextUnit = unit.getNextSubUnit()
        boolean success = false
        while nextUnit != null && !success:
            success = processUnit(nextUnit)
            nextUnit = unit.getNextSubUnit()

    else if unit instanceof SequentialUnit:
        TransformationUnit nextUnit = unit.getNextSubUnit()
        boolean success = true
        while nextUnit != null && success:
            success = processUnit(nextUnit)
            nextUnit = unit.getNextSubUnit()

    else if unit instanceof CountedUnit:
        int counter = unit.getCount()
        boolean success = true
        while counter > 0 && success:
            success = processUnit(unit.getSubUnit())
            counter--

    else if unit instanceof AmalgamationUnit:
        Match kernelMatch = unit.getKernelRule().findMatch()
        if kernelMatch != null:
            if unit.getKernelRule().applyRule(kernelMatch):
                forall multiRule in unit.getMultiRules():
                    forall match in multiRule.findMatch():
                        multiRule.applyRule(match)

    else if unit instanceof ConditionalUnit:
        boolean success = processUnit(unit.getIf())
        if success:
            processUnit(unit.getThen())
        else if unit.getElse() != null:
            processUnit(unit.getElse())
```

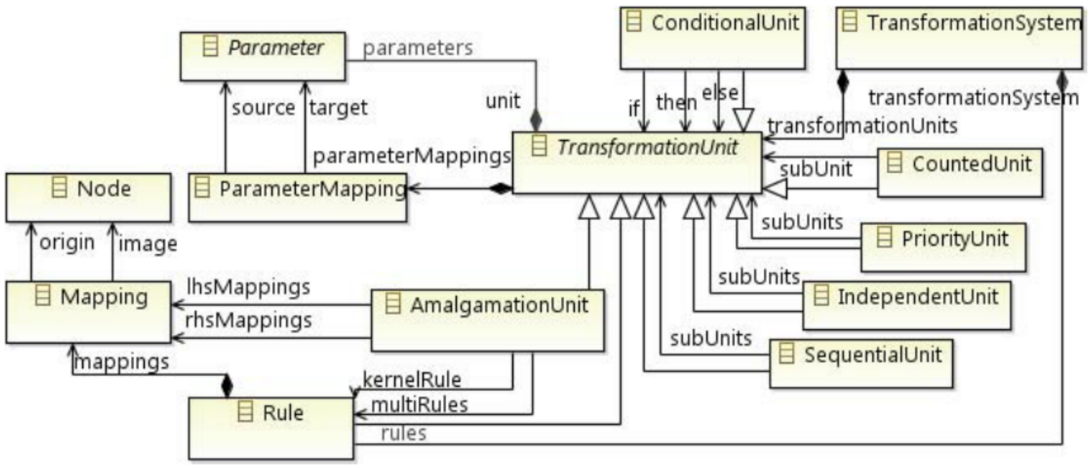


Figure 3.2.: Henshin transformation units [3]

In Stratego the rule application strategy is specified explicitly using higher-order rewriting rules. That is, the developer can specify rules that combine other rules and execute them in a particular order.

Stratego has a number of language constructs to facilitate this, described in Table 3.2.

Algorithm 3.3 describes Stratego's rule execution algorithm in pseudo-code. First a rule is matched against the source model and its variables are bound to specific terms. Afterwards any possible rule preconditions are checked and additional variables are bound. Finally the rule is applied, transforming the model.

Visser provides a number of transformation idioms to illustrate the rule application strategies that could be expressed using Stratego. We take a look at one of them, namely cascading transformation.

Cascading transformation is the most basic strategy for term rewriting (Algorithm 3.4). Small independent transformations are cumulatively applied in order to reach a desired result. The strategy tries to apply any of the rules, starting at the bottom of the abstract syntax tree. Each successful application restarts this process. If no rule can be applied, the algorithm moves to a higher level in the tree. This continues until we reach the root and there are no rules left that apply.

Listing 3.1 shows how this strategy can be expressed in Stratego notation. We first traverse the tree to get to the bottom level using `bottomup(s)`. We try to apply the sequence of rules R_1 to R_n until one of the rules succeeds. If a successful rule application has taken place, `innermost(s)` restarts the transformation on the current subtree. Once no applicable rules remain, `bottomup(s)` moves us to the next level of the tree.

Table 3.2.: A selection of Stratego language constructs

Construct	Description
`ruleName : l -> r where s`	Define a rewrite rule with label `ruleName`, left-hand side `l` and right-hand side `r`. Optionally, preconditions `s` for the rule can be specified by adding a `where` part to the rule.
`ruleName`	Execute the rewrite rule labelled `ruleName`.
`E1 ; E2`	Sequential composition. Execute `E1` and after that `E2`.
`E1 <+ E2`	Deterministic choice. First try `E1`, execute `E2` only if `E1` fails.
`E1 + E2`	Non-deterministic choice. The same as <+, but the first expression to try is chosen randomly.
`E1 < E2 + E3`	Guarded choice. If `E1` succeeds, execute `E2`. If `E1` fails, execute `E3`.
`where(E)`	Test whether `E` applies, but ignore the result of the application.
`not(E)`	Negation. Succeeds if `E` fails to apply.
`id`	Identity. Always succeeds with the original term as the result.
`all(E)`	Apply `E` to each direct subterm.
`one(E)`	Apply `E` to one direct subterm.
`try(E) = E <+ id`	Try to execute `E`, continue if the execution fails.
`repeat(E) = try(E; repeat(E))`	Repeat the execution of `E` until it fails.

Algorithm 3.3 Pseudo-code representation of the Stratego rule application algorithm.

```
function applyRule(Rule rule, AST tree):
    // check for match and collect variable bindings
    Set bindings = rule.l.match(tree)
    // check extra preconditions
    if rule.s != null:
        Set precondition_bindings = rule.s.match(tree)
    if rule.s == null || precondition_bindings != null:
        bindings += precondition_bindings
        if bindings != null:
            // replace variables in term and replace tree
            tree = rule.r.replaceVars(bindings)
```

Algorithm 3.4 Pseudo-code representation of Stratego's cascading transformation strategy.

```
// cascading transformations
function cascade(Set rules, AST tree):
    // traverse bottom-up
    forall child in tree.getChildren():
        cascade(rules, child)

    // try to apply rules
    forall rule in rules:
        if applyRule(rule, tree):
            // successful application => start applying all
            // rules again bottom-up
            cascade(rules, tree)
            break
```

Listing 3.1: Cascading transformation expressed in the Stratego language. [14]

```
bottomup(s) = all(bottomup(s)); s

innermost(s) = bottomup(try(s; innermost(s)))

simplify = innermost(R1 <+ ... <+ Rn)
```

3.4. ATL

The last transformation language we discuss in this chapter is ATL [13]. Although ATL is a hybrid language we only consider its declarative part here. As explained in Chapter 2, we use the imperative part for validation purposes in Chapter 4.

Declarative ATL has three kinds of transformation rules: *matched*, *lazy* and *unique lazy*.

Matched rules Matched rules match a specific type of source elements, possibly further restricted by a guard condition. A developer specifies the target elements that have to be created from these source elements, and how their properties should be initialized. Each source element may only be matched by a single matched rule.

ATL executes matched rules automatically in an arbitrary order. It also creates traceability links between source and target elements.

Lazy rules Lazy rules are matched rules that are not executed automatically. They can be triggered by other rules as many times as necessary. Traceability links are not created for these rules.

Unique lazy rules Unique lazy rules are lazy rules that are applied at most once for a specific match. If triggered more than once, they return the target elements that were already created earlier.

Algorithm 3.5 describes ATL's transformation execution algorithm in pseudo-code. First, rules are matched and a list of matches is created. Second, the target elements are created. Last, the properties of the target elements are initialized.

Although we do not explicitly include the execution algorithm for (unique) lazy rules here, the algorithm for matched rules sans the matching can easily be reused for those kinds of rules.

3.5. Analysis of Commonality and Variability

Now that we have introduced the transformation execution algorithms of the various transformation languages, we proceed by analysing their commonalities and variabilities. Looking at the execution of a model transformation, we can distinguish two levels at which the execution takes place: the rule level and the transformation level.

3.5.1. Rule Level

At the rule level we look at the operations that take place during rule execution. As we are considering declarative languages, developers do not explicitly specify most of these operations, i.e. they happen 'behind the scenes'.

The transformation operations we identified are listed in Table 3.3. In every language the source model is navigated and elements from the source patterns are matched. Creating elements and setting properties is also something that every language is capable of.

In contrast, deleting elements or properties is not always possible. In ATL it is not possible to navigate the target model, and consequently it is not possible to delete parts of this model. Deletion of elements is possible when using ATL's *refining mode*, but this mode can only be used if source and target models conform to the same metamodel. Moreover, some language features are disabled in this mode.

ATL does have two operations that are not available in other languages, namely for creating and querying a trace. Traces can be created in other languages, but are not supported natively in the implementation of these languages. Instead, they have to be specified explicitly in the transformation definition using a custom-made metamodel.

3.5.2. Transformation Level

At the transformation level we look at operations that make up a rule application strategy. The operations we identified are listed in Table 3.4.

The most basic operation in such a strategy is executing a rule. In some languages (e.g. Stratego) it is possible to check if there are matches for a rule in the source model,

Algorithm 3.5 Pseudo-code representation of the ATL transformation algorithm.

```
Transformation transformation
InputModel inputModel
List matches
List targetElements
Dictionary traces

// match rules
forall rule in transformation.getMatchedRules():
    forall match in inputModel.getMatches(rule):
        matches += match

// execute matched rules
forall match in matches:
    // create target elements
    forall t in match.getRule().getTargets():
        Object target = new (t.getType())()
        targetElements += target
        // create trace links
        traces[match.getSource()] = target

// initialize target-element properties
forall target in targetElements:
    forall property in target.getProperties():
        // if primitive, assign directly
        if isPrimitive(property.getContent()):
            target.setProperty(property.getName(), property.getContent())
        else:
            // if source element, resolve, then assign
            Trace trace = traces[property.getContent()]
            if trace != null:
                target.setProperty(property.getName(), trace)

            // if trace query, resolve, then assign
            else if isQuery(property.getContent()):
                Trace trace = traces[property.getContent().getSource()]
                target.setProperty(property.getName(), trace)

            // if target element, assign directly
            else:
                target.setProperty(property.getName(), property.getContent())
```

Table 3.3.: Comparison of transformation operations at the rule level.

Operations	GT	GROOVE	Henshin	Stratego	ATL
Navigate model	X	X	X	X	X
Match pattern	X	X	X	X	X
Create element	X	X	X	X	X
Delete element	X	X	X	X	
Set property	X	X	X	X	X
Delete property	X	X	X	X	
Create trace					X
Query trace					X

but not actually execute the rule. ATL is a special case: this feature is not available to developers, but is nonetheless an essential part of the language's execution algorithm.

Other available operations are:

- Choosing a rule non-deterministically (i.e. randomly) from the available rules in a transformation.
- Executing a rule under the condition that a certain other rule does not apply.
- Executing a sequence of rules in some predefined order.
- Executing a sequence of rules in an undefined order. ATL's execution algorithm works in this way: all matched rules are executed once for each match, in an undefined order.
- Executing a rule an undefined number of times.
- Executing a rule a specified number of times.
- Repeating the execution of a rule as long as possible. After executing the rule, we check whether or not the rule still applies. If so, we execute it again.
- Trying to execute a rule, but continue if it fails to apply. One of its uses is to stop recursion over a list once all elements have been processed, and continue the execution at a higher level.

3.6. Conclusions

We analysed the structure and primitive operations within the execution algorithms of five different transformation languages. We described the commonalities and variabilities between them. In the next chapter we define, based on this analysis, our own language for defining model transformation languages in our common execution environment.

Table 3.4.: Comparison of transformation operations at the transformation level.

Primitive	GT	GROOVE	Henshin	Stratego	ATL
Execute rule	X	X	X	X	X
Match rule (no execution)				X	*
Choose a rule non-deterministically	X	X	X	X	
Execute a rule conditionally		X	X	X	
Specify a sequence of rules with defined order		X	X		
Specify a sequence of rules with undefined order					X
Execute rule an undefined number of times		X			
Execute rule a specified number of times			X		
Repeat execution of a rule as often as it is possible	X	X	X	X	
Continue if rule fails to apply		X		X	

4 Primitive Model Transformation Language

Using our observations discussed in the previous chapter, we can now define our own model transformation language, consisting of just a few primitive operations, in which all of the afore discussed transformation execution algorithms can be expressed.

4.1. Language Definition

4.1.1. Model Navigation Operations

First of all we need basic model navigation operations. The navigation operations available are quite dependent on the representation of the model in an actual implementation. Because the availability of such operations is not related to the way a transformation is executed, we assume that all operations necessary to navigate a model efficiently are available. Some example model navigation operations that are used in our primitive representations of the execution algorithms are described below.

transformation.getRules() Retrieve the rules in a transformation (graph transformation).

transformation.getSourceModel() Retrieve the source model of a transformation (graph transformation).

transformation.getTransformationUnits() Retrieve the transformation units in a transformation (Henshin).

transformation.getRules() Retrieve the rules in a transformation (Stratego).

rule.getTransformation() Retrieve the transformation to which a rule belongs (graph transformation).

rule.getTarget() Retrieve the target elements of a rule (graph transformation).

rule.getPreconditions() Retrieve the preconditions for a rule (Stratego).

unit.getSubUnits() Retrieve the subunits of a transformation unit (Henshin).

tree.getChildren() Retrieve the children of an AST element (Stratego).

4.1.2. Collection Operations

Second, we assume that basic operations to navigate and manipulate collections are also available. Some example collection operations that we use in our primitive representations of the execution algorithms are described below.

collection.add(element) Add an element as the last element of a collection.

collection.unshift() Add an element as the first element of a collection.

collection.shift() Remove the first element from a collection and return the element.

collection.remove(element) Remove an element from a collection.

collection.get(index) Retrieve the element at the specified index of a collection.

collection.contains(element) Check if a collection contains an element. Returns a boolean.

collection.copy() Returns a deep copy of a collection.

4.1.3. Logical and Arithmetic Operations

Third, we assume support for basic logical and arithmetic operations. Some examples of these operations are:

==	Equals
!=	Not equal to
>	Greater than
<	Less than
=	Assignment

+	Addition
-	Subtraction
*	Multiplication
/	Division

4.1.4. Functions

Fourth, we assume that functions are supported. Strictly speaking, functions are not primitives, but we use them to structure the algorithms and to enable recursion. Functions can have parameters that pass references. A function does not return any value. A function named `myFunction` with a parameter named `parameter1` and type `TypeX` can be specified as follows:

```
function myFunction(TypeX parameter1):
    operation
```

4.1.5. Exception Handling

Fifth, we have exception handling. This was not observed in any of the selected transformation algorithms, but can be useful to convey information about errors during the transformation execution. An exception is raised using `raise()`. The `raise` primitive is never applicable and thus causes the execution to return to a higher level. The exception can be caught using `catch()`, which is applicable if the specified exception has been raised and was not yet caught.

4.1.6. Transformation-level Operations

The primitive transformation operations are summarized in Table 4.1. We discuss them one by one.

Find match for rule `match_rule(`*`rule`*`[, `*`model`*`])` returns `Set` of `matches`
Searches the source model for elements that match the left-hand side of the specified rule. It returns a set of matches. If the model parameter is provided, the specified (sub)model is searched instead of the source model.
We do not define the search algorithm that is used to search the model. The choice of search algorithm is left to the implementers of the primitive language.

Non-deterministic choice `nd_choice(`*`set of expressions`*`)` returns `expression`
Chooses one element of the set of expressions at random and returns the element. An expression can be an operation, a call to a function or a literal value (eg. 42).

Ordered sequence *`expression`* `;` *`expression`*
Executes the expressions in the defined order. If the first expression fails to execute, the second will not be executed.

Continue on failure `try {` *`expression`* `}`
Tries to execute the expression. If it fails, continue as if the execution was successful.

Conditional execution `try {` *`expression`* `} else {` *`expression`* `}`
Tries to execute the first expression. If it fails, try to execute the second expression.

4.1.7. Rule-level Operations

Create element *`model`*`.add(`*`element`*`)` `create_element(`*`class`*`)`
The first operation (add) adds an element to a model. This is used in algorithms that make copies of target elements and include the copies in the target model.
The second operation (`create_element`) creates a new element that is an instance of the specified class.

Delete element `model.remove(element)`
Deletes an element from a model.

Set property `element.setProperty(name, value)`
Sets the value of the property 'name' on the element to 'value'.

Delete property `element.deleteProperty(name)`
Deletes (or nullifies) the property 'name' on the element.

Create trace `add_trace(rule, sourceElement, targetElement)`
Creates a trace that connects the source element to the target element in the context of a rule.

Query trace `match_trace_source(sourceElement)` returns `Trace`
Looks for a trace with the specified source element and returns it. The execution of this operation fails if no trace is found. A `Trace` object contains a reference to a rule, a source element and a target element.

4.1.8. Operations not included

Some of the operations observed earlier are not included in our primitive language:

- Execute a rule. This is a primitive operation at the transformation level, but one that is tied closely to the execution algorithm that is used. For example, in graph transformation, rules are executed on a per-rule basis, while in ATL the execution is split into three phases (match, create, initialize) and all rules have to be processed before the next phase is started. Therefore we cannot provide a single language construct that can be used in all algorithms.
- Execute rule an undefined number of times. This operation is only present in GROOVE and was most likely added to the language because of its roots in defining automata. We see no practical use for this operation in a general-purpose transformation language and it can be emulated with the other primitives anyway.
- Execute a rule a specified number of times. This is another operation that can be emulated using the other primitives on a per-algorithm basis, for example:

  ```
  function for(Rule r, int n):
    try { n > 0 ; executeRule(r) ; for(r, n-1) } else { n == 0 }
  ```

  In this example `executeRule()` is the function the algorithm in question uses to execute a transformation rule.
- Repeat execution of a rule as often as it is possible. This can be achieved by means of recursion, for example:

Table 4.1.: Primitive transformation language operations

Operation	Representation
Function	`function` *`functionName`*`(`*`TypeX parameter1`*`):`
Find match for rule	`match_rule(`*`rule`*` [, `*`model`*`])` returns `Set` of `matches`
Non-deterministic choice	`nd_choice(`*`set of expressions`*`)` returns `expression`
Ordered sequence	*`expression`*` ; `*`expression`*
Continue on failure	`try { `*`expression`*` }`
Conditional execution	`try { `*`expression`*` } else { `*`expression`*` }`
Create element	*`model`*`.add(`*`element`*`)` `create_element(`*`class`*`)`
Delete element	*`model`*`.remove(`*`element`*`)`
Set property	*`element`*`.setProperty(`*`name`*`, `*`value`*`)`
Delete property	*`element`*`.deleteProperty(`*`name`*`)`
Create trace	`add_trace(`*`rule`*`, `*`sourceElement`*`, `*`targetElement`*`)`
Query trace	`match_trace_source(`*`sourceElement`*`)` returns `Trace`
Raise exception	`raise(`*`exceptionName`*`)`
Catch exception	`catch(`*`exceptionName`*`)`

```
function repeat(Rule r): try { executeRule(r) ; repeat(r) }
```

- Specify a sequence of rules with undefined order. This can be emulated by choosing rules non-deterministically and removing them from a set (or from the transformation model) once they have been executed.

4.2. Application of the Primitive Language

We have defined our primitive model transformation language for specifying transformation algorithms, and we now express all of the aforediscussed algorithms using our own language.

4.2.1. Graph Transformation

An implementation of the single-pushout graph transformation algorithm in our primitive language is included in Appendix A.1. We compare this implementation (*PL*) to the pseudo-code version (*PC*) described in Algorithm 3.1 in Chapter 3.

First, we select an arbitrary transformation rule (PL line 11, PC line 8). We search for all occurrences of the rule's left-hand side in the source model (PL line 12, PC line 10) and select an arbitrary match (PL line 14, PC line 10).

We execute the rule with this match (PL lines 22-30, PC lines 11-25). We remove all matched source elements that do not exist in the rule's right-hand side (PL line 26, PC lines 12-14). We add the elements that exist in rule's right-hand side but not in the rule's left-hand side (PL line 28, PC lines 21-23). Finally, we restart the transformation (PL line 5, PC lines 25-29).

If no match is found for a rule, we try matching another rule (PL lines 15-19, PC lines 8-10). When no rule matches, the transformation is complete (PL lines 3-6, PC lines 31-33).

4.2.2. Henshin

An implementation of Henshin's execution algorithm in our primitive language is included in Appendix A.2. This implementation reuses the `executeRule()` function from the graph transformation algorithm implementation. Again we compare this implementation (*PL*) to the pseudo-code version (*PC*) described in Algorithm 3.2 in Chapter 3.

We process all transformation units that are subunits of the `Transformation` one at a time (PL lines 6-8, PC lines 3-4).

- If the unit is a `Rule`, we look for a match in the source model and execute the `Rule` if one is found (PL lines 12-15, PC lines 8-11).
- If the unit is an `IndependentUnit`, we execute its subunits using the graph transformation algorithm (PL lines 17-9, 46-63; PC lines 13-15).
- If the unit is a `PriorityUnit`, we execute its subunits according to their priorities (PL lines 21-23, 65-73; PC lines 17-22).
- If the unit is a `SequentialUnit`, we execute its subunits in the predefined sequence (PL lines 25-27, 75-82; PC lines 24-29).
- If the unit is a `CountedUnit`, we execute its subunits the specified number of times (PL lines 30-31, 84-91; PC lines 31-36).
- If the unit is an `AmalgamationUnit`, we first try to find a match for the `kernel rule` (PL lines 33-35, 93-94; PL lines 38-41). If a match is found, we try to find matches for the `multi-rules` and execute them (PL lines 95-101, PC lines 42-44).
- If the unit is a `ConditionalUnit`, we first try to execute the `if` subunit (PL lines 37-38, 103-105; PC lines 46-47). If the execution of `if` succeeds, we execute the `then` subunit next (PL line 106, PC lines 48-49). If the execution of `if` fails, we execute the `else` subunit next (PL lines 107-109, PC lines 50-51).

4.2.3. Term Rewriting

An implementation of Stratego's cascading transformation strategy in our primitive language (PL) is included in Appendix A.3. The pseudo-code version (PC) was described in Algorithm 3.4 in Chapter 3.

To start, we traverse the AST to get to the bottom level (PL lines 4-6, PC lines 2-5). We try to execute the first rule in the list (PL lines 7-13, PC lines 8-9) and continue with the next rule if the first fails to execute (PL lines 16-20, PC lines 8-9). Once a successful rule execution has taken place, we restart the transformation on the current subtree (PL lines 14-15, PC lines 12-13). When no executable rules remain we move to the next level of the tree (PL lines 6-8, PC lines 4-5).

The execution of a rule was described in Algorithm 3.3 in Chapter 3. We look for matches of the left-hand side of the rule in the source model (PL line 25, PC line 3). We bind its variables to the source elements that matched (PL line 26, PC line 3). If appropriate, we check if the rule's preconditions match and bind additional variables to source elements (PL lines 27-32, PC lines 5-8). Finally, we replace the bound source elements by their respective target elements from the right-hand side of the rule (PL line 33, PC lines 9-11).

4.2.4. Declarative ATL

An implementation of the declarative ATL algorithm in our primitive language (PL) is included in Appendix A.4. The pseudo-code version (PC) was described in Algorithm 3.5 in Chapter 3.

The execution consists of three phases. First, we match the left-hand side of all rules to elements in the source model and save the matches using traces (PL lines 2, 6-15, 37-45; PC lines 7-10). Second, we create the right-hand side elements of the rule in the target model (PL lines 3, 17-25, 47-57; PC lines 12-19). Third, we initialize the properties of the target elements we just created (PL lines 4, 27-35, 59-94; PC lines 22-40).

4.2.5. Imperative ATL

We use the imperative ATL algorithm to validate that our primitive pseudo language can indeed express other kinds of transformation languages that were not considered while defining our language.

Imperative ATL uses *called rules* instead of matched rules. Called rules are not executed automatically, they must be called explicitly from another rule. Instead of defining the left-hand side and right-hand side of a rule, one defines a right-hand side and a '`do` section'. The `do` section can contain imperative statements, namely:

- A call to a another called rule or lazy (unique) rule. Parameters can be passed to the called rule.
- An assignment statement `target <- expression` that assigns the result of an `expression` to a transformation module property `target`.

- A conditonal statement: `if(` *condition* `) {` *statements*1 `} else {` *statements*2 `}`
- An iteration statement that iterates over a collection, executing statements one time for each element of the collection assigned to `iterator`:

  `for(`*iterator* `in` *collection*`) {` *statements* `}`

An implementation of the imperative ATL algorithm in our primitive language is included in Appendix A.5. An imperative ATL transformation starts at the *entry point rule* (line 2). When a called rule is executed, we first create the target elements and initialize their properties (lines 6-8). Second, all statements in the `do` section are executed (line 10-55). If another rule is a called, we execute this rule with the passed parameters (lines 17-18). For assignment statements we evaluate the expression and assign its result to the specified module property (lines 22-23).

For conditional statements we first evaluate the condition (lines 27-29). If the condition is true, we execute the first group of statements (line 31). If not, we execute the second group of statements (lines 32-34).

For iteration statements we iterate over the provided collection, adding the first element of the collection to the execution context of the enclosed statements each time (lines 37-55).

It should be noted that our implementation reuses some of the functions of our declarative ATL implementation. Additionally, the official ATL implementation does not create traces when executing imperative ATL. Our implementation does create traces, because it enables us to reuse part of the declarative algorithm. We consider this an implementation detail.

4.3. Related Work

Syriani and Vangheluwe [23] have conducted a similar deconstruction of transformation languages into primitives. Their primitives are objects (for instance a `Matcher`, `Rewriter`, `Iterator`) that perform parts of the transformation and exchange messages. A rough comparison between their and our primitives can be found in Table 4.2. Unfortunately, they do not motivate their choice of primitives.

Syriani and Vangheluwe do include support for parallel processing, an aspect of model transformation that we do not deal with in this thesis.

Our language includes primitives for creating and querying traces, while they do not provide support for tracing in their language.

4.4. Conclusions

We have defined our own model transformation language consisting of just a few primitive model transformation operations. We specified the transformation execution algorithms of five transformation languages in our own language, including a language

Table 4.2.: Comparison of transformation primitives

Syriani and Vangheluwe [23]	Our primitives
`Matcher`	Match rule
`Rewriter`	Create/delete element, set/delete property
`Iterator`	Conditional execution, functions
`Resolver`	Conditional execution, exception handling
`Rollbacker`	Continue on failure
`Selector`	Non-deterministic choice
`Synchronizer`	
`Composer`	Ordered sequence
	Create/query trace

with a transformation approach not initially considered. In the next chapter we work towards a transformation framework that can actually execute these specifications.

5 Transforming Model Transformations

Having defined our own primitive transformation language in Chapter 4, we can build a transformation framework that can actually transform models using transformation execution specifications written in this language.

5.1. Architecture

As mentioned before, model transformations can be described using the pattern shown in Figure 5.1. We have a source model M_a, a target model M_b and a transformation definition T_{ab}. The transformation definition is written in a transformation language *TL*. When executed, the transformation engine runs T_{ab}, which transforms M_a to M_b.

Our transformation engine architecture is shown in Figure 5.2. Instead of interpreting T_{ab} directly, we use it as input model for transformation T_{pl}. T_{pl} is a transformation written in our primitive language *PL*. It contains rules for interpreting *TL* transformations.

T_{pl} itself is interpreted by our primitive language interpreter I_{pl}.

Since the transformation language is itself implemented as a transformation, it is relatively easy to add support for new languages and extend existing languages.

5.2. Interpreter Implementation

The transformation engine is implemented as a plug-in for the Eclipse platform [24]. This gives us access to the Eclipse Modeling Framework [5] and related libraries.

Though the syntax of *PL* as described in Chapter 4 works fine for defining transformation languages, it became clear switching to a more established syntax would ease the implementation of the *PL* interpreter significantly. Implementing the Chapter

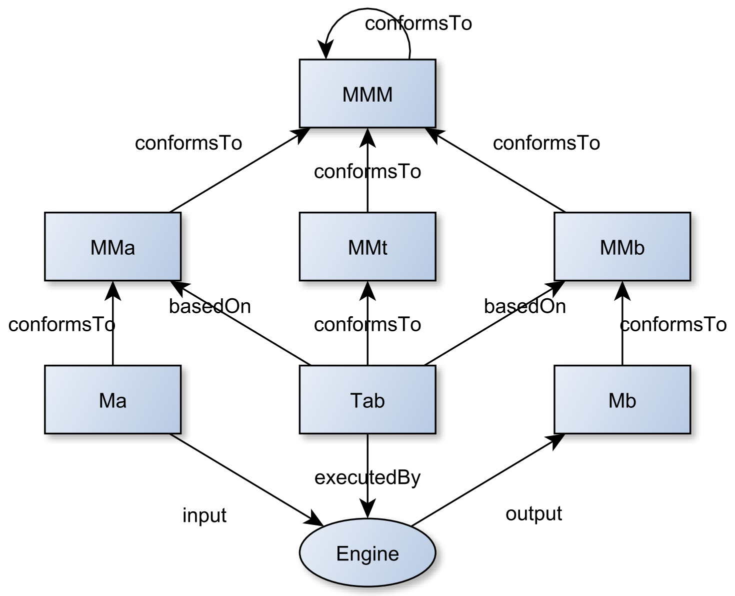


Figure 5.1.: Model Transformation Pattern

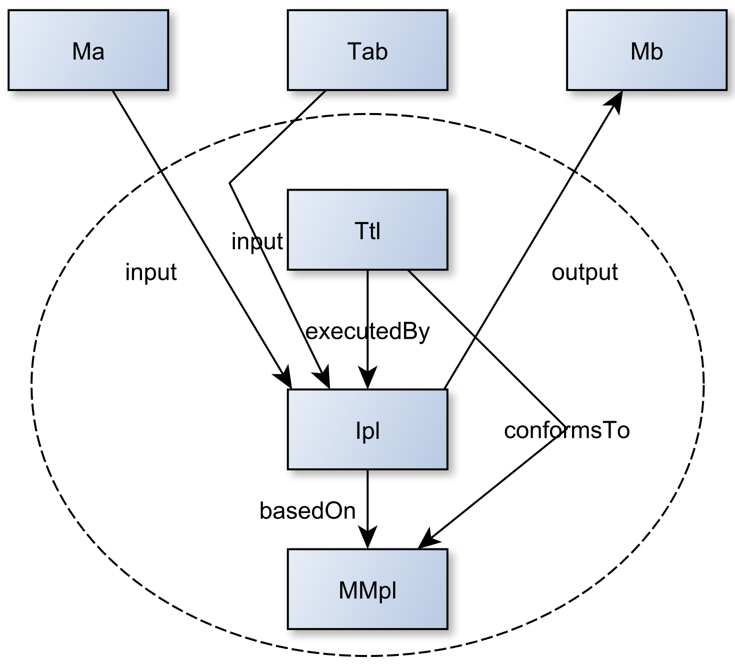


Figure 5.2.: Transformation engine architecture

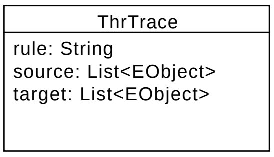


Figure 5.3.: Thrascias Trace structure

4 syntax would mean implementing our own OCL [8] engine, which is required for model navigation and collection operations. This is not a small task and out of scope for this thesis.

We decided to use a syntax based on *Mistral* [9], a general purpose model transformation language and to use its implementation of OCL. We use only a small subset of Mistral's syntax, which is much more elaborate. Our implementation is called *Thrascias*.

Thrascias' abstract syntax (MM_{pl}) is defined as an ECore model, shown in Figure 5.4.

The concrete syntax is described in Table 5.1. It comprises OCL for expressions, operations for defining which models are used, a function definition and the primitive operations described in Chapter 4. In addition to the input and output models, two other metamodels are always present: the EMF metamodel `Ecore` and `Thrascias`. The Thrascias metamodel contains the Trace class which the interpreter uses for managing traces. The structure of this class is shown in Figure 5.3.

For reflective purposes, the model of the transformation currently being executed is accessible as `_Transformation_`.

We map the concrete syntax to the abstract syntax using EMFText [25]. It also provides an editor and syntax checker for the language. The interpreter itself is implemented as a standard interpreter pattern with a stack.

5.3. Transformation Language Implementation

We implement two of the analysed transformation languages in our framework: graph transformation and ATL. The implementation of the other languages is left as further work. The implementations are included in Appendix B.1 and Appendix B.2.

When implementing graph transformation there are lots of different graph transformation languages to choose from, all with different syntax and slightly different semantics. We select SimpleGT [26], because it integrates well with the Eclipse Modelling Framework and its semantics are similar to the graph transformation algorithm we analysed.

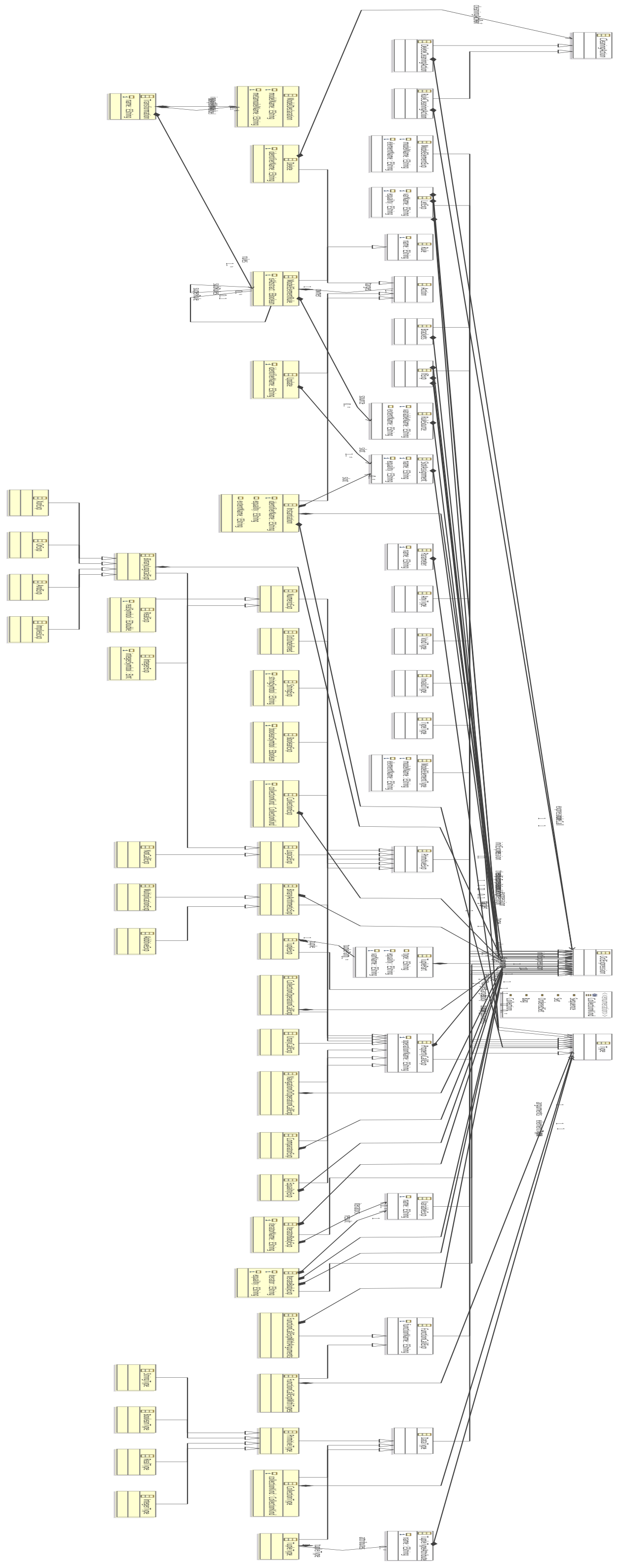


Figure 5.4.: Thrascias abstract syntax

Table 5.1.: Thrascias operations

Operation	Representation
Define transformation	`transform` *`transformationName`*
Specify input model	`inputModel` *`modelName`*`:` *`metamodelName`*
Specify output model	`outputModel` *`modelName`*`:` *`metamodelName`*

Function	*`functionName`* `ModelElementRule {`
	`source [`*`parameter1`*`:` *`TypeX`*`]`
	`target [`*`result1`*`:` *`TypeY`* `=` *`expression`*`]`
	`}`

Model navigation	OCL and Ecore
Collection operations	OCL
Find match for element in set	`match(element, elementSet)` returns `Sequence(Set(OclAny))`
Non-deterministic choice	*`collection`*`->any(true)`
Ordered sequence	*`expression`* `and` *`expression`*

Conditional execution	*`expression`* `or` *`expression`*
	`if` *`expr`* `then` *`expr`* `else` *`expr`* `endif`

Continue on failure	Using `ModelElementRule`
Create element	`create_element(`*`eClass`*`)`
Delete element	`delete` *`element`*
Set property	`update` *`element`* `{`*`property`*`=`*`value`*`}`
Delete property	`update` *`element`* `{delete` *`property`*`}`
Create trace	`add_trace(`*`rule`*`, [`*`sourceElements`*`[,` *`targetElements`*`]])`
Query trace	`match_trace_source(`*`sourceElement`*`)` returns `Trace`
Raise exception	`raise(`*`exception`*`)`
Catch exception	`catch(`*`exception`*`)`

5.4. Conclusions

We have explained the architecture and syntax of our transformation framework, and the rationale behind them. We wrote implementations for two transformation languages using our framework. In the next chapter we will validate our implementations with a number of common model transformation scenarios.

6 Model Transformation Scenarios

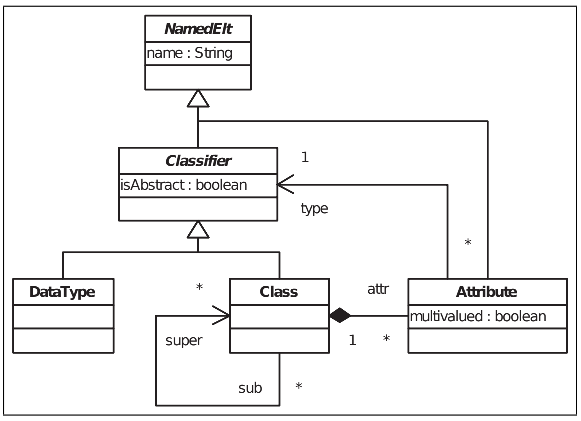
We validate our implementation by executing two different model transformation scenarios, namely transforming object-oriented classes to relation tables and pulling up subclass attributes to a superclass. First, we implement the model transformations using SimpleGT and declarative ATL. Second, we execute these implementations with both their original interpreter and our own transformation framework. Third, we compare the results of the executions.

6.1. Common Use Cases

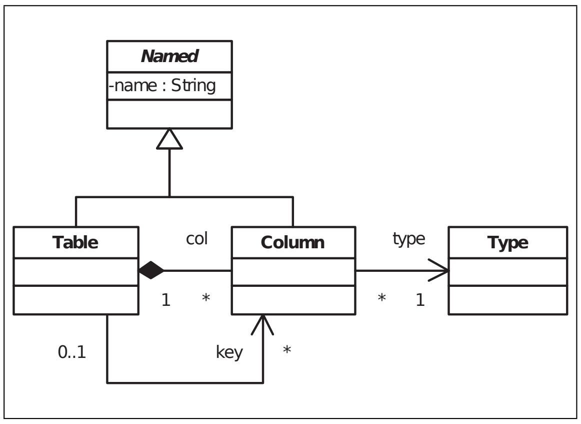
The use cases in which model transformations are most commonly applied can be categorized into five scenarios [11]:

- Generating lower-level models and source code from higher-level models
- Reverse engineering higher-level models from lower-level models or source code
- Generating views from a model using a query
- Mapping and synchronizing models
- Model evolution or refactoring

The first three scenarios usually involve generating and parsing source code files or other documents without an explicit metamodel. Because our research focuses on model-to-model transformations, we validate our implementation using the last two scenarios.



(a) Object-oriented class



(b) Relational table

Figure 6.1.: Metamodels for mapping scenario [4]

6.2. Object-oriented Class to Relational Table

For the model mapping use case, we choose one of the classic examples of model transformation: the mapping of an object-oriented class diagram to a relational database table model.

6.2.1. Implementation

Both models are shown in Figure 6.1. In the class metamodel (Figure 6.1a) a `Class` has a name (inherited from the abstract `NamedElt`) and a number of `Attributes`. These `Attributes` may be `multivalued`. `Class` inherits from `Classifier`, which allows us to declare the `type` of `Attributes`.

In the table metamodel (Figure 6.1b) a `Table` has a name (inherited from the abstract `Named`) and a number of `Columns` and `keys`. A `Column` has a `Type` and belongs to a `Table` (its `owner`) and might be a `key of` another `Table`.

The implementations of this transformation in SimpleGT and declarative ATL are included in Appendix C.

6.2.2. Results

We apply the transformations to an example model of a book library [5] and verify the results manually. The structure of the model is shown in Figure 6.2. Unfortunately it is hard to visualize the model before and after the transformation, because its structure essentially stays the same (as it should). In our manual verification we found no differences in transformation result between the original interpreters of SimpleGT or ATL and our own implementations.

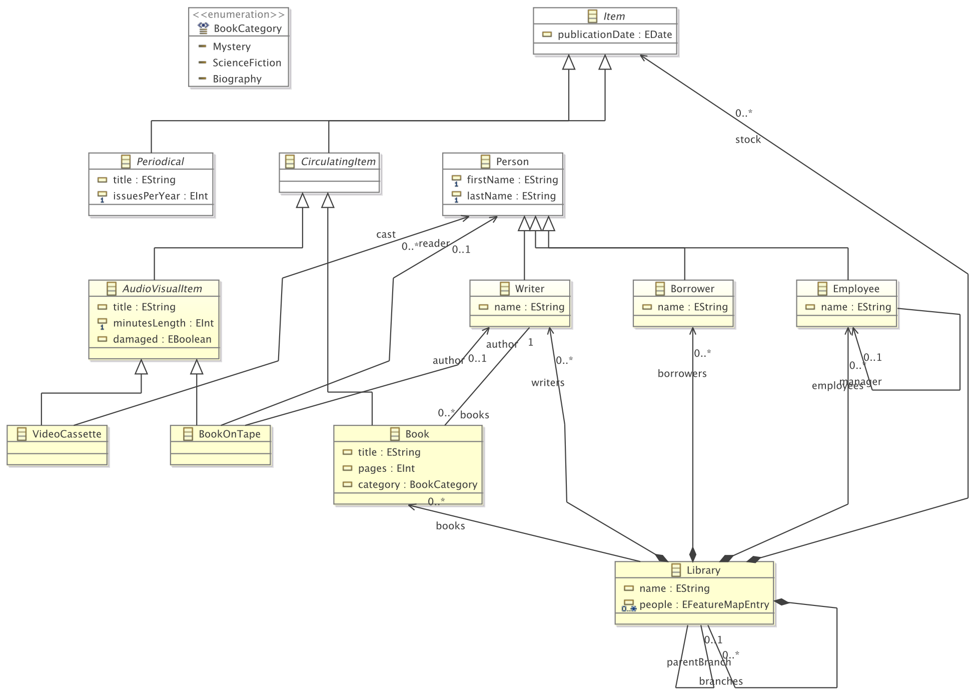


Figure 6.2.: EMF 'extlibrary' metamodel (adapted from [5])

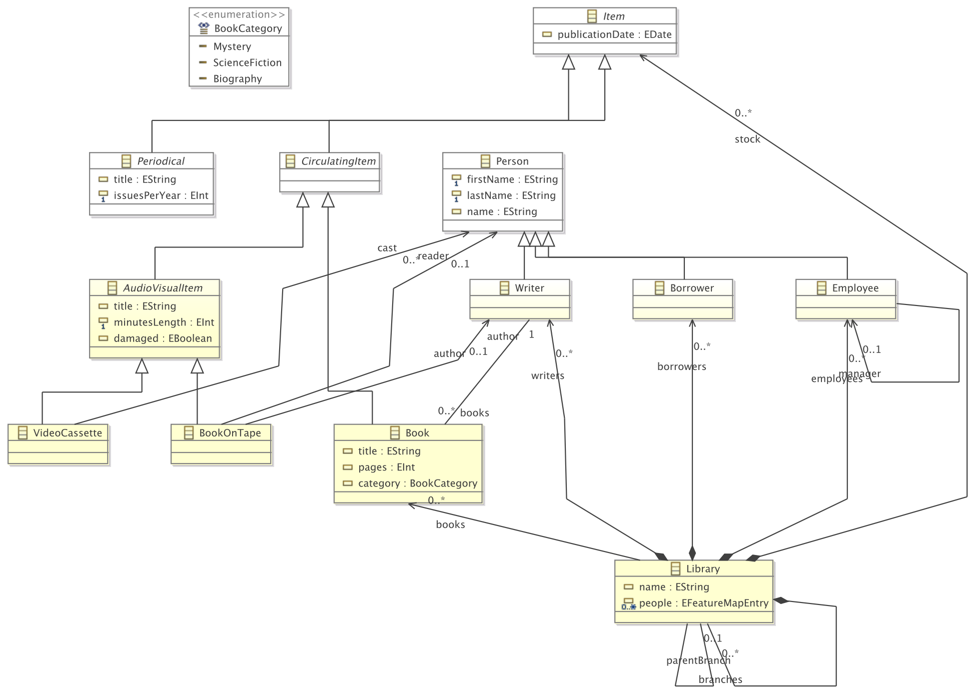
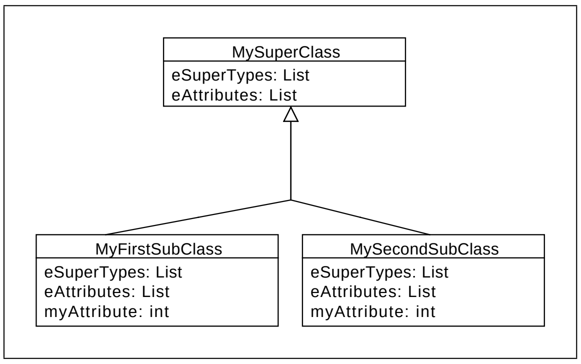
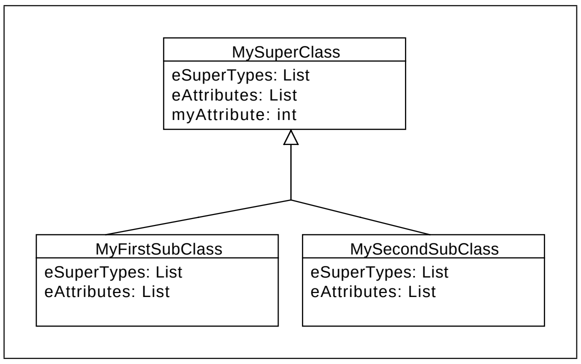


Figure 6.3.: 'extlibrary' model after Pull Up Class Attribute transformation



(a) Before transformation



(b) After transformation

Figure 6.4.: Model for refactoring scenario

6.3. Pull Up Class Attribute to Superclass

For the second use case, model refactoring, we choose to refactor an object oriented class hierarchy. We take an attribute that is present on all subclasses of a certain class and move the attribute to the superclass. This is also known as 'pulling up' a class attribute.

6.3.1. Implementation

We move an attribute called `myAttribute` from a class to its superclass (Figure 6.4). The model conforms to the Ecore metamodel. Prerequisite for this refactoring is an attribute which is present in all subclasses of a certain superclass, and is not present in the superclass itself.

The implementations of this transformation in SimpleGT and declarative ATL are included in Appendix D.

6.3.2. Results

Again we apply the transformations to our example book library model (Figure 6.2). We move the attributes `name` of all the subclasses of `Person` to their superclass. The result of the transformation is shown in Figure 6.3. Verifying the results manually, we find no differences in transformation result between the original interpreters of SimpleGT or ATL and our own implementations.

6.4. Conclusions

We have introduced a number of common model transformation scenarios. We implemented two of the scenarios in SimpleGT and ATL. We then ran the transformations on an example model using both the original interpreters and our own implementations. In our manual verification we found no difference in transformation result.

In the next chapter we summarize the results of our research and look at possible future directions of research.

7 Conclusions and Future Work

In Chapter 1 we introduced the problem of having many different model transformation languages, all with their own unique approach. We recognized that, despite their differences, they have some features in common. We asked the question: “how can we execute different model transformation languages in a common environment, while retaining the variability between the languages?”. We can now answer this question based on our research.

7.1. What should a common execution environment for model transformation languages look like?

In Chapter 2 we explored models and model transformations. We categorized transformation languages based on properties like transformation approach and rule application strategy. Using this categorization, we selected five significantly different transformation languages for further investigation: graph transformation, GROOVE, Henshin, Stratego and declarative ATL.

In Chapter 3 we took a closer look at the execution algorithms of the selected languages. We analysed the commonality and variability between them, both at the rule level and the transformation level.

In Chapter 4 we defined our own primitive model transformation language based on the analysis of commonality and variability. We expressed the execution algorithms for all analysed languages in our primitive language.

Subsequently, we took a look at imperative ATL. It has a completely different transformation approach, one not considered during the design of our primitive language. Nevertheless we were able to express its execution algorithm, demonstrating that our language can be used to describe the execution algorithm of any transformation language.

7.2. How do we build support for executing different transformation languages in this common environment?

In Chapter 5 we discussed the implementation of a model transformation framework based on our primitive transformation language. We described the architecture, syntax and interpreter design. We implemented two model transformation languages in our framework: SimpleGT and declarative ATL.

In Chapter 6 we tested our transformation framework by executing two common model transformations in both our own interpreter and the original interpreter for the transformation language. For our humble test scenarios, the result turned out to be the same regardless of the interpreter used.

7.3. What are the limitations of executing transformation languages in a common environment?

We have shown that, despite the variability between them, transformation languages have lots of commonalities. Their execution algorithms can be described using less than fifteen primitive operations. Based on these findings we have implemented our own transformation framework for executing model transformation written in these languages. However, our implementation has some limitations.

Model representation To ease the implementation of our framework, we require all models to be represented as EMF Ecore models. In some of the discussed transformation languages models are usually represented differently. Future work could include creating an abstraction layer which maps these model representations to Ecore.

OCL for navigation and collection manipulation Another generalization is the use of OCL for model navigation in all languages. Other navigation languages can be used, but one would have to add an interpreter for such a language either to the Thrascias interpreter or to the transformation language implementation.

Multiple source and target models To prevent the example implementations from becoming overly complex we did not consider transforming from multiple source models, or transforming to multiple target models. This however does not mean there is any theoretical or practical impediment to implement this.

Limited number of implemented languages We implemented just two of the five languages analysed. Furthermore there are many more transformation languages available. Future work could include the implementation of the other three languages mentioned in this thesis, or any other available transformation language.

Limited number of test scenarios We performed tests using two relatively simple transformation scenarios. Testing larger and more advanced transformations can be considered future work.

7.4. Looking ahead

Besides addressing some of the limitations of our implementation, we take a moment to look at possible directions for future research.

We have shown that different transformation languages can be interpreted with a single interpreter. A logical next step would be the composition of transformations written in different transformation languages into a single transformation. This would enable one to create libraries of transformations for common transformation tasks that can be used in no matter which language. Moreover a separate transformation language could be chosen for each transformation subproblem. Combining transformation languages introduces a host of new problems, like different composition strategies and conflict resolution between rules in different languages.

A second direction for future research could be parallel processing. Our framework does not take advantage of multiple processing units at the moment. Additional language constructs might be necessary for operations like synchronization of parallel threads.

Performance improvement is a possible third direction of future research. Although we have not conducted any testing for processing speed or memory usage, we expect our prototype implementation to be slower than the original interpreters for most languages.

A Primitive Representations of the Algorithms

A.1. Single-Pushout Graph Transformation

```
function executeTransformation(Transformation transformation):
    // repeat as often as possible
    try {
        transformGraph(transformation.getRules()) ;
        executeTransformation(transformation)
    }

function transformGraph(Set rules):
    try {
        // choose random rule and find match
        Rule rule = nd_choice(rules) ;
        Set matches = match_rule(rule) ;
        // execute this rule
        executeRule(rule, nd_choice(matches))
    } else {    // no matches
        // remove rule from list
        rules = rules.remove(rule) ;
        // try again
        transformGraph(rules)
    }

function executeRule(Rule rule, Match match):
    // make temporary copy of graph
    Model D = rule.getTransformation().getSourceModel().copy() ;
    // remove elements not in target
    removeOldElements(D, rule, match.copy()) ;
    // add elements not in source
    addNewElements(D, match, rule.getTarget()) ;
    // replace graph by changed copy
    rule.getTransformation().setSourceModel(D)

function removeOldElements(Model D, Rule rule, Match match):
    try {
        ModelElement element = nd_choice(match.getSource()) ;
        try {
            rule.getTarget().contains(element)
        } else {
            D.remove(element)
        } ;
        removeOldElements(D, rule, match.remove(element))
    }

function addNewElements(D, match, rhs):
    try {
        ModelElement element = nd_choice(rhs) ;
        try {
            match.getSource().contains(element)
        } else {
            D.add(element)
        } ;
        addNewElements(D, rhs.remove(element))
    }
```

A.2. Henshin

```
function executeTransformation(Transformation transformation):
    processAllUnits(transformation.getTransformationUnits())

function processAllUnits(Set units):
    try {
        TransformationUnit unit = nd_choice(unit) ;
        processUnit(unit) ;
        processAllUnits(units.remove(unit))
    }

function processUnit(TransformationUnit unit):
    try {
        unit.isRule() ;
        Set rule_matches = match_rule(unit) ;
        executeRule(unit, nd_choice(matches))
    } else {
        try {
            unit.isIndependentUnit() ;
            processIndependentUnit(unit)
        } else {
            try {
                unit.isPriorityUnit() ;
                processPriorityUnit(unit, null)
            } else {
                try {
                    unit.isSequentialUnit() ;
                    processSequentialUnit(unit)
                } else {
                    try {
                        unit.isCountedUnit() ;
                        processCountedUnit(unit, unit.getCount())
                    } else {
                        try {
                            unit.isAmalgamationUnit() ;
                            processAmalgamationUnit(unit)
                        } else {
                            unit.isConditionalUnit() ;
                            processConditionalUnit(unit)
                        }
                    }
                }
            }
        }
    }

function processIndependentUnit(IndependentUnit unit):
    // repeat as often as possible
    try {
        processIndependentSubUnits(unit.getSubUnits()) ;
        processIndependentUnit(unit)
    }
```

```

function processIndependentSubUnits(Set units):
    try {
        // process random unit
        TransformationUnit unit = nd_choice(units) ;
        processUnit(unit)
    } else {
        // remove unit from list
        units = units.remove(unit) ;
        // try again
        processIndependentSubUnits(units)
    }

function processPriorityUnit(PriorityUnit unit, TransformationUnit currentSubUnit):
    try {
        TransformationUnit nextSubUnit = unit.getNextSubUnit(currentSubUnit) ;
        try {
            processUnit(nextSubUnit)
        } else {
            processPriorityUnit(unit, nextSubUnit)
        }
    }

function processSequentialUnit(SequentialUnit unit, TransformationUnit currentSubUnit):
    try {
        TransformationUnit nextSubUnit = unit.getNextSubUnit(currentSubUnit) ;
        processUnit(nextSubUnit) ;
        processSequentialUnit(unit, nextSubUnit)
    } else {
        nextSubUnit == null
    }

function processCountedUnit(CountedUnit unit, Integer counter):
    try {
        counter > 0 ;
        processUnit(unit.getSubUnit()) ;
        processCountedUnit(unit, counter - 1)
    } else {
        counter == 0
    }

function processAmalgamationUnit(AmalgamationUnit unit):
    processUnit(unit.getKernelRule()) ;
    processMultiRules(unit.getMultiRules())

function processMultiRules(Set rules):
    try {
        processIndependentSubUnits(rules) ;
        processMultiRules(rules)
    }
```

```

function processConditionalUnit(ConditionalUnit unit):
    try {
        processUnit(unit.getIf()) ;
        processUnit(unit.getThen())
    } else {
        processUnit(unit.getElse())
    }
```

A.3. Stratego

```
function executeTransformation(Transformation transformation):
    cascade(transformation.getRules(), transformation.getSourceModel())

function cascade(List rules, AST tree):
    try {
        cascade(rules, nd_choice(tree.getChildren()))
    } ;
    applyAndRestart(rules, tree, rules)

function applyAndRestart(List rules, AST tree, List allRules):
    Rule rule = rules.shift() ;
    try {
        applyRule(rule, tree) ;
        cascade(allRules, tree) ;
        raise(BreakException)
    } else {
        try {
            catch(BreakException)
        } else {
            applyAndRestart(rules, tree, allRules)
        }
    }

function applyRule(Rule rule, AST tree):
    Match m = nd_choice(match_rule(rule, tree)) ;
    Set bindings = m.getBindings() ;
    try {
        Match s = match_rule(rule.getPreconditions(), tree) ;
        bindings.append(s.getBindings())
    } else {
        rule.getPreconditions() == null
    } ;
    tree = rule.getTarget().replaceVars(bindings)
```

A.4. Declarative ATL

```
function executeTransformation(Transformation transformation):
    matchAllRules(transformation.getRules()) ;
    createElements(transformation.getAllTraces()) ;
    initElements(transformation.getAllTraces())

function matchAllRules(Set rules):
    try {
        // choose rule randomly
        Rule rule = nd_choice(rules) ;
        // find matches and save them
        Set rule_matches = match_rule(rule) ;
        findMatches(rule, rule_matches) ;
        // match next rule
        matchAllRules(rules.remove(rule))
    }

function createElements(Set traces):
    try {
        // choose trace (= match) randomly
        Trace trace = nd_choice(traces) ;
        // create targets for trace
        createTargets(trace, trace.getRule().getTarget().copy()) ;
        // proceed with next trace
        createElements(traces.remove(trace))
    }

function initElements(Set traces):
    try {
        // choose trace randomly
        Trace trace = nd_choice(traces) ;
        // init its targets
        initTargets(trace.getTarget().copy()) ;
        // proceed with next trace
        initElements(targets.remove(targetElement))
    }

function findMatches(Rule rule, Set matches):
    try {
        // get match
        Match match = nd_choice(matches)
        // add trace with empty target
        add_trace(rule, match.getSource(), null) ;
        // process next match
        findMatches(rule, matches.remove(match))
    }

function createTargets(Trace trace, Set targets):
    try {
        // choose target element randomly
        ModelElement targetElement = nd_choice(targets) ;
        // create empty target element
        Object t = trace.getRule().getTransformation().getTargetModel()
            .add(create_element(targetElement.class)) ;
        // set target element for trace
        trace.getTarget().add(t) ;
        createTargets(trace, targets.remove(targetElement))
    }

function initTargets(Set targets):
    try {
        ModelElement targetElement = nd_choice(targets) ;
        initProperties(targetElement, targetElement.getProperties().copy()) ;
        initTargets(targets.remove(targetElement))
    }

function initProperties(ModelElement target, Set properties):
    try {
        // choose property randomly
        ModelElementProperty property = nd_choice(properties) ;
        try {
            // if primitive, assign directly
            property.getContent().isPrimitive() ;
            target.setProperty(property.getName(), property.getContent())
        } else {
            try {
                // if source element, resolve, then assign
                Trace trace = match_trace_source(property.getContent()) ;
                target.setProperty(property.getName(),
                    trace.getTarget().get(0))
            } else {
                try {
                    // if trace query, resolve, then assign
                    property.getContent().isQuery() ;
                    Trace trace = match_trace_source(property.getContent()
                        .getSource()) ;
                    target.setProperty(property.getName(), trace.getTarget()
                        .get(property.getContent().getPattern()))
                } else {
                    // if target element, assign directly
                    target.setProperty(property.getName(),
                        property.getContent())
                }
            }
        } ;
        initProperties(target, properties.remove(property))
    }
```

A.5. Imperative ATL

```
function executeTransformation(Transformation transformation):
    executeRule(transformation.getEntryPointRule(), null)

function executeRule(Rule rule, List parameters):
    // Execute 'to' block
    Trace trace = add_trace(rule, null, null) ;
    createTargets(trace, rule.getTarget().copy()) ;
    initTargets(trace.getTarget().copy()) ;
    // Execute 'do' block
    executeStatements(rule, parameters, rule.getStatements())

function executeStatements(Rule rule, List parameters, List statements):
    try {
        Statement s = statements.get(0) ;
        try {
            // Call another rule
            s.isCallStatement() ;
            executeRule(s.getRule(), s.getParameters())
        } else {
            try {
                // Assign module property
                s.isAssignmentStatement() ;
                rule.getModule().setProperty(s.getTarget(),
                    s.getExpression())
            } else {
                try {
                    // Execute 'if' statement
                    s.isIfStatement() ;
                    try {
                        s.getCondition() ;
                        // Condition = true
                        executeStatements(rule, parameters,
                            s.getStatements1())
                    } else {
                        // Condition = false
                        executeStatements(rule, parameters,
                            s.getStatements2())
                    }
                } else {
                    try {
                        // Execute 'for' statement
                        s.isForStatement() ;
                        executeForStatement(rule, parameters,
                            s.getCollection(),
                            s.getStatements())
                    }
                }
            }
        } ;
        executeStatements(rule, parameters, statements.unshift())
    }
```

```

function executeForStatement(Rule rule, List parameters, Collection
    collection,
        List statements):
    try {
        List newParameters = parameters.copy().add(collection.get(0)) ;
        executeStatements(rule, newParameters, statements) ;
        executeForStatement(rule, parameters, collection.unshift(),
            statements)
    }
```

B

Thrascias Implementations of the Algorithms

B.1. SimpleGT

```
transform simplegt
inputModel SimpleGT : simplegt
outputModel Out : simplegt

executeTransformationRule ModelElementRule {
    source [transformation: SimpleGT!Module]
    target [
        -- run transformation, resulting in a new model
        transformStep: Tuple() =
            transformGraphRule(OrderedSet{transformation.models}
            ->first(), transformation.elements),
        -- replace old source model by new model
        update transformation {models = OrderedSet{transformStep.newModel}},
        -- if a rule was applied, we run the transformation again
        result: SimpleGT!OclModel =
            if not transformStep.newModel.oclIsUndefined() then
                executeTransformationRule(transformation)
            else
                transformation.models ->first()
            endif
    ]
}

transformGraphRule ModelElementRule {
    source [sourceModel: SimpleGT!OclModel, rules: OrderedSet(SimpleGT!Rule)]
    target [
        -- select a random rule and try to find a match
        rule: SimpleGT!Rule = rules ->asSet()->any(true),
        matches: Set(Set(OclAny)) =
            if not rule.oclIsUndefined() then
                match(rule.input.elements, sourceModel)
            else
                -- no rules left
                Set{}
            endif,
        newModel: SimpleGT!OclModel =
            if matches ->notEmpty() and not
                hasApplicableNacRule(sourceModel, rule).result
            then
                -- match found, execute the rule for this match
                executeRule(sourceModel, rule, matches ->any(true)).result
            else
                -- no match, remove this rule from the set and try again
                transformGraphRule(rules ->excluding(rule)).newModel
            endif
    ]
}

hasApplicableNacRule ModelElementRule {
    source [sourceModel: SimpleGT!OclModel, rule: SimpleGT!Rule]
    target [
        result: Boolean = rule.nac ->exists(n | match(n.elements,
            sourceModel) ->notEmpty())
    ]
}

executeRule ModelElementRule {
    source [sourceModel: SimpleGT!OclModel, rule: SimpleGT!Rule, match:
        Set(OclAny)]
    target [
        -- remove elements that are not in the target graph
        D: OrderedSet(ECore!EObject) = removeOldElementsRule(sourceModel,
            rule, match)
            .result,
        -- check for dangling edges
        updatedObjectSet: OrderedSet(ECore!EObject) =
            if hasDanglingEdgeRule(D ->select(o|
                o.oclIsKindOf(ECore!EReference)))
                .result then
                -- abort rule execution
                sourceModel.elements
            else
                -- add elements that are not in the source graph
                D ->union(rule.output.elements ->collect(e|
                    create_element(e.eClass()))
                ->asOrderedSet())
            endif,
        -- update model
        update sourceModel {elements = updatedObjectSet},
        result: SimpleGT!OclModel = sourceModel
    ]
}

```

```
removeOldElementsRule ModelElementRule {
    source [sourceModel: SimpleGT!OclModel, rule: SimpleGT!Rule, match:
        Set(OclAny)]
    target [
        oldElements: Set(OclAny) = match - rule.output.elements,
        result: OrderedSet(ECore!EObject) = sourceModel.elements -
            oldElements
    ]
}

hasDanglingEdgeRule ModelElementRule {
    source [edges: Collection(ECore!EReference)]
    target [
        result: Boolean = edges ->exists(e| e.eContainer.isOclUndefined()
            or e.getEType().isOclUndefined())
    ]
}
```

B.2. Declarative ATL

```
transform atl
inputModel ATL: atl
outputModel Out: atl

executeTransformationRule ModelElementRule {
    source [transformation: ATL!Module]
    target [
        matches: Tuple() =
            matchAllRulesRule(OrderedSet{transformation.inModels}->first(),
            OrderedSet{transformation.elements->select(e|
                e.oclIsKindOf(ATL!MatchedRule))},
        elements: Tuple() = createElementsRule(OrderedSet{matches.traces}),
        result: Tuple() = initElementsRule(OrderedSet{matches.traces})
    ]
}

-- Find all matches for rules and save them using traces
matchAllRulesRule ModelElementRule {
    source [sourceModel: ATL!OclModel, rules: OrderedSet(ATL!MatchedRule}]
    target [
        rule: ATL!MatchedRule = rules ->asSet()->any(true),
        matches: Sequence(Set(OclAny)) =
            if not rule.oclIsUndefined() then
                -- match rule in source model
                match(rule.inPattern.elements, sourceModel)
            else
                -- no rules left to match
                Sequence{}
            endif,
        -- create trace with empty target
        traces: Bag(Thrascias!Trace) = matches ->collect(m| add_trace(rule,
            m)),
        result: Bag(Thrascias!Trace) =
            if rules ->size() > 1 then
                -- match next rule
                matchAllRulesRule(sourceModel, rules
                    ->excluding(rule))->including(matches)
            else
                Bag{}
            endif
    ]
}

-- Create all new elements using the traces
createElementsRule ModelElementRule {
    source [traces: OrderedSet(Thrascias!Trace)]
    target [
        result: Collection(Tuple()) = traces ->collect(tr|
            tr.target ->collect(ta| createTargetsRule(tr, ta,
                OrderedSet{tr.rule.module.outModels}->first()))
        )
    ]
}

-- Create new element for trace in output model
createTargetsRule ModelElementRule {
    source [trace: Thrascias!Trace, targetElement: Ecore!EObject, outModel:
        ATL!OclModel]
    target [
        -- create empty target element
        newElement: OclAny = create_element(targetElement.eClass()),
        -- set target element for trace
        update trace {target = trace.target ->including(newElement)},
        -- add element to output model
        update outModel {elements = outModel.elements
            ->including(newElement)}
    ]
}

-- Initialize properties for all new elements
initElementsRule ModelElementRule {
    source [traces: OrderedSet(Thrascias!Trace)]
    target [
        result: OclAny = traces ->collect(tr|
            tr.source ->collect(s|
                -- find input pattern element for trace
                let ipElement: ATL!InPatternElement = match(s,
                    rule.inPattern.elements)
                    ->any(m| m.notEmpty()) in
                -- find target elements and initialize their properties
                ipElement.mapsTo -> collect(opElement|
                    initTargets(opElement, match(opElement, tr.target)
                        ->any(m| m.notEmpty()))
                )
            )
        )
    ]
}

-- Initialize all properties of a target element
initTargets ModelElementRule {
    source [outputElement: ATL!OutPatternElement, targetElement:
        Ecore!EObject]
    target [
        result: Bag(Tuple()) = outputElement.bindings ->collect(b|
            resolvePropertyRule(targetElement, b))
    ]
}

-- Resolve the value of a property
resolvePropertyRule ModelElementRule {
    source [targetElement: Ecore!EObject, property: ATL!Binding]
    target [
        result: OclAny = if property.value.type.oclIsKindOf(ATL!Primitive)
            then
```

```
                -- property has primitive value, initialize directly
                initPropertyRule(targetElement, property, property.value)
            else
                -- if property has source element value, resolve, then
                    initialize
                let trace: Thrascias!Trace =
                    match_trace_source(property.value) in
                    if not trace.oclIsUndefined() then
                        initPropertyRule(targetElement, property,
                            trace.target)
                    else
                        -- if property has trace query value, resolve, then
                            initialize
                        -- example: resolveTemp(source_element,
                            target_pattern)
                        if property.value.oclIsKindOf(ATL!OperationCallExp)
                            and
                            property.value.operationName = 'resolveTemp' then
                            let trace: Thrascias!Trace =
                                match_trace_source(property.value.arguments
                                    ->at(1)) in
                            let sourceObject: Ecore!Object =
                                OrderedSet{trace.target}->first() in
                                initPropertyRule(targetElement, property,
                                    sourceObject
                                    .eGet(sourceObject.eClass().getEAllStructuralFeatures
                                        ->any(f| f.getName() =
                                            property.value.arguments
                                        ->at(2))))
                        else
                            -- if property has target element value,
                                initialize directly
                            initPropertyRule(property, property.value)
                        endif
                    endif
            endif
    ]
}

-- Initialize the value of a property
initPropertyRule ModelElementRule {
    source [targetElement: Ecore!EObject, property: ATL!Binding, value:
        OclAny]
    target [
        -- find structural feature for property
        feature: ECore!EStructuralFeature = targetElement.eClass()
            .getEAllStructuralFeatures() ->any(f| f.getName() =
                property.propertyName),
        -- set value of feature
        result: Boolean = targetElement.eSet(feature, value)
    ]
}
```

C Implementations of the Mapping Scenario

C.1. SimpleGT

```
module Class2Relational;

metamodel Class;
metamodel Relational;
transform C: Class, R: Relational;

rule Class2Table {
    from c : Class!Class
    to t : Relational!Table (
            name =~ c.name,
            col =~ Sequence {key}->union(c.attr->select(e | not
                e.multivalued)),
            key =~ Set{key}
        ),
        key : Relational!Column (
            name =~ 'objectId',
            type =~ Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        )
}

rule DataType2Type {
    from dt : Class!DataType
    to t : Relational!Type (name =~ dt.name)
}

rule DataTypeAttribute2Column {
    from a : Class!Attribute (
            type =~ dt,
            multivalued =~ false
        ),
        dt : Class!DataType
}

rule MultivaluedDataTypeAttribute2Column {
    from a : Class!Attribute (
            type =~ dt,
            multivalued =~ true
        ),
        dt : Class!DataType
    to t : Relational!Table (
            name =~ a.owner.name + '_' + a.name,
            col =~ Sequence{id, value}
        ),
        id : Relational!Column (
            name =~ a.owner.name + 'Id',
            type =~ Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        ),
        value : Relational!Column (
            name =~ a.name,
            type =~ a.type
        )
}

rule ClassAttribute2Column {
    from a : Class!Attribute (
            type =~ c,
            multivalued =~ false
        ),
        c : Class!Class
    to foreignKey : Relational!Column (
            name =~ a.name + 'Id',
            type =~ Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        )
}

rule MultivaluedClassAttribute2Column {
    from a : Class!Attribute (
            type =~ c,
            multivalued =~ true
        ),
        c : Class!Class
    to t : Relational!Table (
            name =~ a.owner.name + '_' + a.name,
            col =~ Sequence{id, foreignKey}
        ),
        id : Relational!Column (
            name =~ a.owner.name + 'Id',
            type =~ Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        ),
        foreignKey : Relational!Column (
            name =~ a.name + 'Id',
            type =~ Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        )

}
```

C.2. Declarative ATL

```
-- Adapted from ATL Modeling Zoo [27]

module Class2Relational;
create OUT : Relational from IN : Class;

rule Class2Table {
    from
        c : Class!Class
    to
        out : Relational!Table (
            name <- c.name,
            col <- Sequence {key}->union(c.attr->select(e | not
                e.multivalued)),
            key <- Set {key}
        ),
        key : Relational!Column (
            name <- 'objectId',
            type <- Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        )
}

rule DataType2Type {
    from
        dt : Class!DataType
    to
        out : Relational!Type (
            name <- dt.name
        )
}

rule DataTypeAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!DataType) and not a.multivalued
        )
    to
        out : Relational!Column (
            name <- a.name,
            type <- a.type,
            owner <- thisModule.resolveTemp(a.owner, 'key')
        )
}

rule MultivaluedDataTypeAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!DataType) and a.multivalued
        )
    to
        out : Relational!Table (
            name <- a.owner.name + '_' + a.name,
            col <- Sequence {id, value}
        ),
        id : Relational!Column (
            name <- a.owner.name + 'Id',
            type <- Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        ),
        value : Relational!Column (
            name <- a.name,
            type <- a.type
        )
}

rule ClassAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!Class) and not a.multivalued
        )
    to
        foreignKey : Relational!Column (
            name <- a.name + 'Id',
            type <- Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        )
}

rule MultivaluedClassAttribute2Column {
    from
        a : Class!Attribute (
            a.type.oclIsKindOf(Class!Class) and a.multivalued
        )
    to
        t : Relational!Table (
            name <- a.owner.name + '_' + a.name,
            col <- Sequence {id, foreignKey}
        ),
        id : Relational!Column (
            name <- a.owner.name + 'Id',
            type <- Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        ),
        foreignKey : Relational!Column (
            name <- a.name + 'Id',
            type <- Class!DataType.allInstances()->select(e | e.name =
                'Integer')->first()
        )
}
```

D Implementations of the Refactoring Scenario

D.1. SimpleGT

```
module PullUpAttribute;

metamodel Ecore;
transform IN: Ecore, OUT: Ecore;

rule AddAttributeToSuperclass {
    from sc : Ecore!EClass,
        c : Ecore!EClass (
            eSuperTypes =~ sc,
            eAttributes =~ a
        ),
        a : Ecore!EAttribute (
            name =~ 'myAttribute'
        )
    not sc : Ecore!EClass (
            eAttributes =~ a
        ),
        c2 : Ecore!EClass (
            eSuperTypes =~ sc,
            eAttributes =~ c2.eAttributes->excluding(a)
        )
    to out : Ecore!EClass (
            name =~ sc.name,
            eSuperTypes =~ sc.eSuperTypes,
            eAttributes =~ sc.eAttributes->including(a)
        )
}

rule RemoveAttributeFromSubclass {
    from sc : Ecore!EClass,
        c : Ecore!EClass (
            eSuperTypes =~ sc,
            eAttributes =~ a
        ),
        a : Ecore!EAttribute (
            name =~ 'myAttribute'
        )
    not sc : Ecore!EClass (
            eAttributes =~ a
        ),
        c2 : Ecore!EClass (
            eSuperTypes =~ sc,
            eAttributes =~ c2.eAttributes->excluding(a)
        )
    to out : Ecore!EClass (
            name =~ c.name,
            eSuperTypes =~ c.eSuperTypes,
            eAttributes =~ c.eAttributes->excluding(a)
        )
}
```

D.2. Declarative ATL

```
module PullUpAttribute;
create OUT : Ecore from IN : Ecore

rule AddAttributeToSuperclass {
    from
        c : Ecore!EClass (
            Ecore!EClass.allInstances()->forAll(e|
                c.isSuperTypeOf(e) implies e.getEAttributes()->exists(a|
                    a.name = ca.name))
            and not c.getEAttributes()->exists(a| a.name = ca.name)
        ),
        ca : ECore!EAttribute (
            ca.name = 'myAttribute'
        )
    to
        out : Ecore!EClass (
            name <- c.name,
            eSuperTypes <- c.eSuperTypes,
            eAttributes <- Sequence{c.getEAttributes()}->including(ca)
        )
}

rule RemoveAttributeFromSubclass {
    from
        c : Ecore!EClass (
            c.eAttributes->exists(a| a.name = ca.name) and
            c.eSuperTypes->exists(s|
                Ecore!EClass.allInstances()->forAll(e| s.isSuperTypeOf(e)
                    implies (
                    e.getEAttributes()->exists(a| a.name = ca.name)
                and not s.getEAttributes()->exists(a| a.name = ca.name)
            )))
        ),
        ca : Ecore!EAttribute (
            ca.name = 'myAttribute' and
            ca.eContainer() = c
        )
    to
        out : ECore!EClass (
            name <- c.name,
            eSuperTypes <- c.eSuperTypes,
            eAttributes <- Sequence{c.getEAttributes()}->excluding(ca)
        )
}
```

Bibliography

[1] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss, "Graphical definition of in-place transformations in the Eclipse Modeling Framework," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds. Springer Berlin / Heidelberg, 2006, vol. 4199, pp. 425–439. [Online]. Available: http://dx.doi.org/10.1007/11880240_30

[2] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1–2, pp. 31 – 39, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642308000439

[3] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place EMF model transformations," *Model Driven Engineering Languages and Systems*, vol. 6394, pp. 121–135, 2010.

[4] M. Lawley, K. Duddy, A. Gerber, and K. Raymond, "Language features for re-use and maintainability of mda transformations," in *Workshop on Best Practices for Model-Driven Software Development*, 2004.

[5] The Eclipse Foundation. (2014, May) Eclipse Modeling Framework project. [Online]. Available: http://www.eclipse.org/modeling/emf/

[6] J. Miller and J. Mukerji, "MDA guide," Object Management Group, Tech. Rep., 2003.

[7] S. Kent, "Model driven engineering," in *IFM*, 2002, pp. 286–298.

[8] *Object Constraint Language*, Object Management Group Std., Rev. 2.0, Jun. 2005.

[9] I. Kurtev, "Adaptability of model transformations," Ph.D. dissertation, University of Twente, Enschede, 2005. [Online]. Available: http://doc.utwente.nl/50761/

[10] T. Mens and P. van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, no. 0, pp. 125 – 142, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066106001435

[11] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[12] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, Object Management Group Std., Rev. 1.1, Jan. 2011.

[13] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 128–138.

[14] E. Visser, "Program transformation with Stratego/XT," in *Domain-Specific Program Generation*, ser. Lecture Notes in Computer Science, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. Springer Berlin / Heidelberg, 2004, vol. 3016, pp. 315–349. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-25935-0_013

[15] L. Baresi and R. Heckel, "Tutorial introduction to graph transformation: A software engineering perspective," in *Proceedings of the First International Conference on Graph Transformation*, ser. ICGT '02. London, UK: Springer-Verlag, 2002, pp. 402–429. [Online]. Available: http://dl.acm.org/citation.cfm?id=647562.730670

[16] F. Budinsky, D. Steinberg, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed., ser. Eclipse series. Addison-Wesley, 2009.

[17] A. Rensink, "The edge of graph transformation – graphs for behavioural specification," in *Graph Transformations and Model-Driven Engineering*, ser. Lecture Notes in Computer Science, G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, Eds. Springer Berlin / Heidelberg, 2010, vol. 5765, pp. 6–32. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17322-6_2

[18] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe, *Algebraic approaches to graph transformation. Part I: basic concepts and double pushout approach*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997, pp. 163–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=278918.278928

[19] T. Staijen, "Graph-based specification and verification for aspect-oriented languages," Ph.D. dissertation, University of Twente, Enschede, June 2010, iPA Dissertation Series ; 2010-04. [Online]. Available: http://doc.utwente.nl/71550/

[20] A. Rensink, "The GROOVE simulator: A tool for state space generation," in *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, ser. Lecture Notes in Computer Science, J. Pfaltz, M. Nagl, and B. Böhlen, Eds., vol. 3062. Berlin: Springer Verlag, 2004, pp. 479–485. [Online]. Available: http://doc.utwente.nl/66357/

[21] (2014, May) The Henshin project. Eclipse Foundation. [Online]. Available: http://www.eclipse.org/modeling/emft/henshin/

[22] C. Ermel, E. Biermann, J. Schmidt, and A. Warning, "Visual modeling of controlled EMF model transformation using HENSHIN," *ECEASST*, vol. 32, 2010.

[23] E. Syriani and H. Vangheluwe, "De-/re-constructing model transformation languages," *ECEASST*, vol. 29, pp. 1–14, 2010.

[24] The Eclipse Foundation. (2014, May) Eclipse platform. [Online]. Available: http://www.eclipse.org/

[25] (2014, May) EMFText concrete syntax mapper. [Online]. Available: http://www.emftext.org

[26] D. Wagelaar. (2014, May) SimpleGT graph transformation language for EMFTVM. [Online]. Available: http://code.google.com/a/eclipselabs.org/p/simplegt/

[27] Eclipse Foundation. (2014, May) ATL modeling zoo. [Online]. Available: http://www.eclipse.org/atl/atlTransformations/