**Université de Montréal**

# Pattern-Based Refactoring in Model-Driven Engineering

par

**Chihab eddine Mohamed Omar MOKADDEM**

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en Informatique - Génie Logiciel

juin 2021

© Chihab eddine Mohamed Omar MOKADDEM, 2021

# Université de Montréal

Faculté des études supérieures et postdoctorales

Cette thèse intitulée

# Pattern-Based Refactoring in Model-Driven Engineering

présentée par

## Chihab eddine Mohamed Omar MOKADDEM

a été évaluée par un jury composé des personnes suivantes :

*Sylvie Hamel*
(président-rapporteur)

*Houari Sahraoui*
(directeur de recherche)

*Eugene Syriani*
(co-directeur)

*Michalis Famelis*
(membre du jury)

*Ferhat Khendek*
(examinateur externe)

*Richard MacKenzie*
(représentant du doyen de la FESP)

Thèse acceptée le :
*15 Mars 2021*

# Sommaire

*L'ingénierie dirigée par les modèles (IDM) est un paradigme du génie logiciel qui utilise les modèles comme concepts de premier ordre à partir desquels la validation, le code, les tests et la documentation sont dérivés. Ce paradigme met en jeu divers artefacts tels que les modèles, les méta-modèles ou les programmes de transformation des modèles. Dans un contexte industriel, ces artefacts sont de plus en plus complexes. En particulier, leur maintenance demande beaucoup de temps et de ressources. Afin de réduire la complexité des artefacts et le coût de leur maintenance, de nombreux chercheurs se sont intéressés au refactoring de ces artefacts pour améliorer leur qualité.*

*Dans cette thèse, nous proposons d'étudier le refactoring dans l'IDM dans sa globalité, par son application à ces différents artefacts. Dans un premier temps, nous utilisons des patrons de conception spécifiques, comme une connaissance a priori, appliqués aux transformations de modèles comme un véhicule pour le refactoring. Nous procédons d'abord par une phase de détection des patrons de conception avec différentes formes et différents niveaux de complétude. Les occurrences détectées forment ainsi des opportunités de refactoring qui seront exploitées pour aboutir à des formes plus souhaitables et/ou plus complètes de ces patrons de conceptions.*

*Dans le cas d'absence de connaissance a priori, comme les patrons de conception, nous proposons une approche basée sur la programmation génétique, pour apprendre des règles de transformations, capables de détecter des opportunités de refactoring et de les corriger. Comme alternative à la connaissance disponible a priori, l'approche utilise des exemples de paires d'artefacts d'avant et d'après le refactoring, pour ainsi apprendre les règles de refactoring. Nous illustrons cette approche sur le refactoring de modèles.*

**Mots clefs** : Refactoring · Patrons de conception · Ingénierie dirigée par les modèles · Transformation de modèles · apprentissage de règles · Génie logiciel basé sur la recherche · Approche basée sur l'exemples · Programmation génétique.

# Summary

*Model-Driven Engineering (MDE) is a software engineering paradigm that uses models as first-class concepts from which validation, code, testing, and documentation are derived. This paradigm involves various artifacts such as models, meta-models, or model transformation programs. In an industrial context, these artifacts are increasingly complex. In particular, their maintenance is time and resources consuming. In order to reduce the complexity of artifacts and the cost of their maintenance, many researchers have been interested in refactoring these artifacts to improve their quality.*

*In this thesis, we propose to study refactoring in MDE holistically, by its application to these different artifacts. First, we use specific design patterns, as an example of prior knowledge, applied to model transformations to enable refactoring. We first proceed with a detecting phase of design patterns, with different forms and levels of completeness. The detected occurrences thus form refactoring opportunities that will be exploited to implement more desirable and/or more complete forms of these design patterns.*

*In the absence of prior knowledge, such as design patterns, we propose an approach based on genetic programming, to learn transformation rules, capable of detecting refactoring opportunities and correcting them. As an alternative to prior knowledge, our approach uses examples of pairs of artifacts before and after refactoring, in order to learn refactoring rules. We illustrate this approach on model refactoring.*

**Keywords**: Refactoring · Design pattern · Model-driven engineering · Model transformation · Rules learning · Search-based software engineering · By-example approach · Genetic programming

# Contents

# List of Tables

# List of Figures

# *Acknowledgments*

First and foremost, I thank **Allah** Azza wa Jalla the Almighty for giving me the courage and patience to carry out this thesis. وَمَا تَوْفِيقِي إِلَّا بِاللَّهِ

Still, the completion of this work would not have been possible without the support and invaluable assistance of several people, to whom I want to express my gratitude.

In the first place, I extend my sincere gratitude to Prof. Houari Sahraoui who supervised and supported me with his advice and very precious suggestions during all stages of my Ph.D., for his patience, motivation, enthusiasm, and immense knowledge. I would like to give him a special thanks for being a colleague and friend in the easy and difficult times.

I express my warmest thanks to my co-supervisor Dr. Eugene Syriani, who closely followed the progress of the thesis and its application. His wise advice, constructive criticism, permanent follow-up, and patience were certainly very favorable to the success of this project.

Most important of all, I am extremely grateful to **my parents** for their love, prayers, caring and sacrifices for educating and preparing me for my journey. I ask them to forgive me if I have failed to fulfill any of my duties to them by being occupied and absent. I hope they are satisfied with me. **I dedicate this work to them**. رَبِّي ارْحَمْهُمَا كَمَا رَبَّيَانِي صَغِيرًا

I am grateful to my friends inside and outside Montreal, as well as my family, in particular my brothers and sisters, for believing in me and encouraging me during this period. Thanks to everyone who prayed for me and my success in secret.

Finally, I warmly thank all academic colleagues and friends. Thanks to Geodes laboratory members for the wonderful times we spent together.

# Chapter 1

## Introduction

### 1. Research Context

*Model-Driven Engineering* (MDE) is becoming increasingly popular in the software industry (Mohagheghi et al, 2013). MDE is an approach to software engineering that uses first-rate artifacts from which validation, code, testing and documentation are derived. These artifacts are generally models, meta-models or transformation/generation rules. In an industrial context, these artifacts are increasingly complex, thus elevating the level of abstraction of these artifacts is of tremendous value for various activities in MDE (Mengerink et al, 2017). As a result, most of them are specific to particular fields. For example (Zarour et al, 2020) present a classification of 30 domains-specific BPMN extensions. Over time, the number of extensions is growing (Braun and Esswein, 2014). However, the diversity of MDE artifacts requires the availability of a prior knowledge specific to each artifact in order to be able to refactor them. In other words, the application of the refactoring of these artifacts requires prior expertise.

Existing work on refactoring (Kerievsky, 2005) express this knowledge in the form of catalogs of design patterns. These patterns reduce complexity and execution time, and improve the flexibility and modularity of an MDE artifact. Therefore, identifying the implemented design patterns in an existing artifact can greatly help the developer to better understand the design and better document the artifact. Unfortunately, we are deprived of reusing this valuable knowledge on existing MDE artifacts. Model transformation, for example, is still handled in a superficial manner as analyzes on this artifact are still not available. The absence of analyzes of design patterns on the artifact transformation creates a gap between the maintenance developer and design pattern knowledge. For the sake of reducing this gap,

approaches and tools are needed to analyze the current state of the artifacts and hence, open opportunities to the existing expertise to be reused.

Design pattern detection is a field with all characteristics to cover this gap. The absence of design pattern detection approaches is a strong argument for trying to adopt new techniques for detecting these patterns. The non-existence of the pattern becomes also an opportunity for implementing the pattern. To date, no approach has specifically worked on this case, due to its difficulty. Also, if a pattern has not been entirely implemented, its partial identification provides valuable information which could suggest its correct application or suggests its improvement. Our research work consists of proposing solutions in order to reinforce the absence of methods of pattern detection.

## 2. Problem Statement

The maintenance phase constitutes the most important phase in time (Lehman, 1980) and cost (Schach, 1993), in the software life cycle. Refactoring is a maintenance process that aims to improve the quality of a software by making modifications on its structure while preserving its behavior (Fowler, 1999a). The main question to be answered in this thesis is: ***"To what extent can we automate refactoring in MDE?"***. Refactoring in MDE means refactoring artifacts such as models, metamodels, model transformations, *etc.* The complexity of these large-scale artifacts complicates their refactoring. To address this problem, we have to face two scenarios depending on how prior knowledge, a.k.a. expertise, is available.

The $1^{st}$ **scenario is the *availability of explicit expertise***. In this scenario, we assume knowledge is represented in an explicit way, for example, but not limited to, in the form of algorithms, formal design patterns or anti-patterns. In this case, the refactoring approach can directly use the explicit knowledge in an automatic way.

The $2^{nd}$ **scenario is the *non-availability of explicit expertise***. In this scenario, the knowledge may exist, but only in an implicit way that makes it unusable directly through automation. Implicit knowledge can be found in the history of modifications that are made on an artifact. It can also be found in examples of the before and after refactoring versions of an artifact. Various other sources can contain implicit knowledge. However, the challenge

in this scenario is to extract this knowledge to make it available in an explicit manner so that the refactoring can be automated.

In this thesis, we deal with each scenario separately. Automatic supports, which facilitate the success of these two scenarios, constitute the main objective of this thesis. At this point, it is important to note that the interactions which result from the creation of these supports must be optimal.

### 2.1. Scenario 1: Explicit Expertise Issues

Consider the case where refactoring expertise is explicitly available. It is prior knowledge that expresses how to identify refactoring opportunities and how to apply the refactoring. It is usually the result for extensive research and analysis in various applications. For example, let us consider the refactoring of UML [1] Class diagram. The model has a class `X` that contains an attribute `X.a` which exists in its parent class `Y`. We notice that there is no need for this attribute in `X` as this class can directly use the attribute `a` from its parent class `Y` thanks to the inheritance relationship. An explicit knowledge of how to detect the refactoring opportunity in this model and how to refactor it, could be available as a refactoring rule. The rule dictates to search for classes, by its left-hand side, for the classes `X` and `Y`, the inheritance relation `X`$\longrightarrow$`Y`, and the attributes `X.a` and `Y.a`. Finding such a match on a model represents a refactoring opportunity. Then the rule then states to remove the attribute `a` from `X`. This is the refactoring application.

When languages, in which the artifacts are expressed, are in general use, such as UML, there is a critical mass of researchers/users whose work allowed to model, test, and validate refactoring rules. However, other languages may be specific to particular domains. In this case, refactoring techniques may exist but their knowledge remains specific to the modeling language.

A good practice that appeared recently is the use of design patterns to make successful refactoring of artifacts (Misbhauddin and Alshayeb, 2015). Overtime, catalogs of design patterns (Gamma et al, 1994; Lano and Kolahdouz-Rahimi, 2014) were proposed for different software artifacts. Several studies (Hegedűs et al, 2012; Lano et al, 2018) showed that these patterns reduce complexity and execution time, as well as improve the flexibility and

[1] http://www.uml.org/

modularity of the artifacts. Hence, design patterns constitute a prior knowledge of how an artifact could be refactored (Kerievsky, 2005). Namely, appropriate know-how expertise.

Unfortunately, we are deprived of reusing this valuable know-how expertise on certain MDE artifacts. The lack of necessary tools to inspect the status of design patterns in an artifact creates a gap between the artifact, the maintenance developer, and this know-how expertise. For source code and model many tools and approaches were proposed (Al-Obeidallah et al, 2016a) to solve this problem. For other artifacts in MDE, lack of tools and approaches is noticeable. With the existence of the urgent need for such material in MDE (Bucchiarone et al, 2020), this gap becomes an obstacle.

Although Model transformation is the heart and soul of MDE (Sendall and Kozaczynski, 2003), it also suffers from the ability to use the limited explicit expertise available. To our knowledge, no existing tool or approach targets the detection of design patterns for the case of model transformation. Manual attempts were recently made in (Lano et al, 2018). Furthermore, with the increasing scale and complexity of the used models in MDE, the developed transformations are also increasing in scale and complexity, and are constantly evolving in development projects. Hence, maintenance of the transformations architecture and design becomes a burden (Meyers and Vangheluwe, 2011). Complications in the transformations could paralyze the process of their development, and consequently, paralyze the entire MDE paradigm. Therefore, it is of uttermost importance to provide means of detecting design patterns for model transformations. The motivations cited above constitute a strong argument for choosing the model transformation artifact as a case study for the first scenario.

### 2.2. Scenario 2: Implicit Expertise Issues

Consider the case where refactoring expertise is implicit. This means that there is no explicit way of how to refactor a model. Let us consider the same refactoring example of the previous UML Class diagram. But this time, we do not have the refactoring rule that shows explicitly how to refactor this model. Instead, we only have a version of the model after the refactoring was applied manually. This version could be given by some domain-specific experts. For example, the after refactoring version only consists of the classes `X`, `Y`, the `X`$\longrightarrow$`Y` relationship, and attribute `Y.a` in `Y`.

The challenge here is to make the knowledge of how to identify refactoring opportunities and how to apply the refactoring, explicitly available. This knowledge preferably be represented as a set of refactoring rules.

A good practice in this second scenario, consists of applying automatic learning from examples in order to generate refactoring rules which detect patterns and correct them. In our case, we use a set of refactoring examples of an artifact and we use supervised learning to generate this knowledge (rules) (Varró, 2006). Examples are pairs of before and after refactoring versions of an artifact (source artifact/artifact). They are used to remedy the missing knowledge. A process of learning this knowledge will be carried out. The knowledge should be able to identify refactoring opportunities and apply appropriate refactorings.

At this point, refactoring an artifact could be seen as a particular transformation of the same artifact (Lúcio et al, 2014). It will, therefore, be a model transformation in the case of model and metamodel artifacts and a higher-order transformation in the case of transformation artifacts (Tisi et al, 2009). The idea has been successfully studied for learning transformation rules from examples (Faunes et al, 2013; Baki and Sahraoui, 2016). However, learning refactoring as transformation presents several difficulties to be overcome. First, refactoring is an in-place transformation, which means, modifications to the model are usually made directly on the source model without creating copies. Existing techniques to learn model transformation from examples are intended to generate a completely new artifact. Secondly, in the examples, not all components of the artifact source get affected by the transformation, but only specific ones among them that constitute refactoring opportunities. Thus, the learning process, while searching for patterns on the rules left-hand sides, concerns all components of this artifact, but it is only allowed to apply actions, on the rules right-hand sides, on very specific components of the artifacts. Because any action on the non-opportunity components is considered as wrong actions. This hinders the learning progress. Finally, the rules for refactoring transformation are highly complex, with many elements and a complex logic of interdependence, which makes learning difficult.

The approaches related to this type of learning have all used models, with certain limitations in terms of the types of rules generated or the types of examples used. Nowadays, none of these approaches works specifically on refactoring because of its difficulty, which creates a weak link in the state of the art of this type of learning.

## 3. Proposed Research Contributions

The automation of refactoring depends mainly on the type of expertise at hand, and how to approach it. We believe that ***"it is possible to automate refactoring in MDE when the refactoring expertise is explicitly available and when it is not"***. Hence, we have taken as a major objective in this thesis: ***"Improving refactoring automation in MDE"***. Problems, objectives and contributions are organized around a core element to describe more specifically what we want to achieve. This element is the way how the knowledge is present. In the previous section we presented the two scenarios: When knowledge is explicitly present and when it is implicit. The same structure is used to define two goals related to the scenarios sequentially:

- ***Automating refactoring when the knowledge is explicit***
- ***Automating refactoring when the knowledge is implicit***

In the following, we go with each scenario/goal separately. For each one, objectives, requirements and contributions are presented. The thesis summarize three primary contributions. The three contributions are published in separate papers.

### 3.1. Scenario 1: Refactoring when Knowledge is Explicit

Lano et al. are interested in the maintenance of patterns architecture and design in model transformations (Lano and Kolahdouz-Rahimi, 2014). They proposed to consider the identification of design patterns implemented in existing model transformations as a maintenance solution. This is due to the fact that this identification can considerably help the developer to better understand the design and to better document the transformation (Tsantalis et al, 2006). Moreover, the success of refactoring resides primarily in identifying the right refactoring opportunities (Fowler, 1999a; Mens and Tourwé, 2004). An opportunity can be seen in different ways (Mens and Tourwé, 2004; Al-Dallal, 2015). In this thesis, we consider the identification of refactoring opportunities as the identification of design patterns/anti-patterns (Gamma et al, 1994; Fowler, 1999a). An anti-pattern can be seen as a design flaw (Brown et al, 1998).

Identifying refactoring opportunities may result in three outcomes: the complete, partial or non-existence of an instance of the pattern. The non-existence of the pattern becomes an opportunity to implement the pattern to improve the artifact quality. If a pattern has not

been fully implemented, its partial identification provides valuable information to suggest correct application or propose a variant since any pattern can be implemented according to different variants (Prechelt and Krämer, 1998). After all, design pattern detection is a large, heavy and known problem in its self (Yarahmadi and Hasheminejad, 2020). Therefore the objective of the thesis for this scenario is to ***"propose an automatic approaches that target the detection of design patterns in model transformations"***. Solutions with different properties are desirable which also reinforces the work on the transformation artifact. In the following, we quote the main requirements targeted in this thesis for this objective. The proposed solutions must satisfy as many of these requirements as possible.

**Automation #1.1**. The first requirement that the proposed approach should satisfy, is to offer an automatic solution to detect model transformation design patterns. The authors in (Lano et al, 2018) tries to identify design patterns instances in ATL model transformation (Jouault et al, 2008). Besides that this work is among the few who tries to identify design patterns in model transformation, the pattern identification was done manually. This shows the weakness in the automation aspect. Hence, the urgent need for automatic solutions.

**Variability #1.2**. The approach should be able to detect different variants of the same design pattern. This is a significant prerequisite since any design pattern may be implemented with plenty of variations (Fowler, 1999a; Tsantalis et al, 2006).

**Incompleteness #1.3**. Many design pattern detection approaches filter out real pattern instances due to strict exact matches of generic pattern version (Dong et al, 2009). That is because design patterns may be present in various forms as mentioned formerly. Some design pattern candidates which might be real instances are incomplete forms compared to the pattern generic version. We refer to these last as approximations (Dong et al, 2009). Each design pattern has its approximations with different types. We require from the proposed approach to detect different approximations of the same design pattern.

**Adaptability #1.4**. Anti-patterns are structures that do not differ much from design patterns. Both are defined by static and dynamic aspects/elements (Stoianov and Şora, 2010). In the latter, differentiation may appear. As far as we know, no existing work presents a catalog of anti-patterns for model transformations. However, anti-patterns will surely be presented in the near future. Hence, the need for approaches to detect them. In (Stoianov

and Şora, 2010), the authors proposed a logic-based approach to detect design patterns and anti-patterns in source code. Their approach uses Prolog as an inference engine. Taking this as motivation, another targeting requirement in the thesis is the flexibility of the proposed approaches in terms of being able to detect anti-patterns.

**Wholeness #1.5**. Model transformations are sets of rules linked by control schemes. Their design patterns come on the same form where they are set of participants, that shall match the transformation rules, and their scheduling. Identification of pattern participants instances in the transformation is one step. Another important step is to preserve the correct scheduling between the detected participants instances according to the pattern. We call this the recovery phase. Well, the existence of these two steps is mandatory for the detection of the hole pattern. Hence, considering the hole patterns during detection is required of any proposed approach.

---

***Contribution #1*** In the first paper of the thesis, we satisfy the above requirements. Hence, we present, for first time, an approach to detect complete or partial instances of design patterns in concrete model transformation implementations. It is a model finding approach based on a rule engine, where we map model transformations to an abstract representation and design patterns to rules that these representations must satisfy. After identifying individual participants of a design pattern, we verify that the scheduling scheme described in the pattern is satisfied in the transformation. We compute an accuracy score at each detection step that is finally aggregated and reported. We implemented a prototype where we encode design patterns as rules and that automatically maps a complete model transformation implemented in a specific model transformation language to the abstract representation.

---

The first *Contribution* tackled the first five requirements. The coming requirements are considered as further steps that we would like to achieve in this thesis. Hence, the second *Contribution* is there to face the coming requirements.

**Genericity #1.6**. As mentioned above, design patterns come with plenty of variants and approximations. Hence, an important property of a detection strategy is the genericity with respect to the patterns to detect. Most of the existing work requires to write specific code for each pattern, being queries (Rasool et al, 2010), structural rules (Zanoni et al, 2015), or pattern matching rules (Mayvan and Rasoolzadegan, 2017). Our first contribution was not spared from this manual task. Presenting a generic solution that is fully independent of the input pattern is an additional requirement.

**Efficiency #1.7**. Offering an efficient algorithm to detect the patterns is also required. There are dozens of detection approaches in object-oriented programs (Al-Obeidallah et al, 2016b). These approaches suffer from performance problems, because detecting complete and incomplete occurrences is generally costly in time, due to the large search-space that includes all possible combinations of artifact elements (Guéhéneuc et al, 2010). The authors in (Kaczor et al, 2010) address these issues by using a string matching technique inspired by pattern matching algorithms in bioinformatics. We take this work as inspiration to offer an efficient algorithm that is based on string matching techniques.

---

***Contribution #2*** In the second paper of the thesis, We propose a generic technique to detect design pattern occurrences in model transformation implementations. By being generic, we do not need to re-write the detection code for each design pattern, its variants, and its approximations. We rely on a bit-vector algorithm that has proven to be efficient for string matching problems. We succeed to exceed the first challenge by offering a string representation for model transformation rules and another for design pattern participants. Thus, the detection consists of an automated step that matches the participant strings of a pattern with rule strings of transformation, and a manual step to complete the occurrences. In addition to the performance, an advantage of using this approach is the fact both complete and incomplete occurrences can be detected.

---

### 3.2. Scenario 2: Refactoring when Knowledge is Implicit

The best example of implicit knowledge is a set of refactoring examples that comes in form of pairs of artifacts. Each pair contains the before and after refactoring versions of the same artifact. We apply the learning approach to generate a set of rules that represent the explicit knowledge. The learning is in form of a generalization process that makes the implicit knowledge as explicit. The idea of learning from examples has been widely exposed (Kessentini et al, 2011). Yet, this field has always new challenges when we deal with domain-specific artifacts/languages. However, The set of refactoring rules are in form of an in-place transformation. In-place transformation is a transformation that reacts to the same artifact. This means that refactoring becomes an transformation which tends to improve the quality of the artifact by modifying its structure while preserving its behavior (Van Der Straeten and D'Hondt, 2006). The refactoring rules/transformation is going to detect refactoring opportunities. Then, it proposes the refactorings that better match to the examples of model refactoring.

The objective of the thesis for the second scenario is to ***"propose an automatic approach that learn refactoring rules from refactoring examples"***. As any software engineering artifact, model transformation has a certain quality requirements (Mens and Gorp, 2006). We define in the following seven quality requirements for this objective. The approach proposed within the frame of this thesis should maintain these requirements.

**Scalability #2.1**. A naive approaches, to learn from example, would be to generate all possible combinations of elements on the examples. This strategy fails whenever the examples reached a certain size. This is because the search space grows exponentially depending on the number of elements in the examples. Our approach should address this.

**Normality #2.2**. In model-driven engineering, refactoring is seen as an endogenous transformation. When the transformations create model elements in one model based on properties of another model, it is called out-place (Mens and Gorp, 2006). When the manipulation allowing to obtain the target model is carried out directly on the source model, the transformation is called in-place. Just as most of maintenance tasks, refactoring is more suitable to be an in-place transformation. Above the excess difficulties found in in-place transformation while learning, this last also require to learn default rules scheduling for executing the learned rules. This is because, in in-place transformation, it is important to have a scheduling. Changing the rules execution order may give different transformation results.

**Limited input** #2.3. In by example approaches, all the required knowledge settle in examples. For the sake of learning, many MTBE approaches use examples with explicit traces that show the operations of refactorings (Baki and Sahraoui, 2016; García-Magariño et al, 2009). In (Langer et al, 2010) the authors use demonstrative editing actions that are made by a user while learning. However, it is not easy to have such examples. Hence, we would like to have a tolerant solution. We require that learning should be made on examples without details of refactorings. As well, the approach must be able to learn even on a very small set of example models. Although, this affects the quality of the rules.

**Reusability #2.4**. Certain approaches produce non-executable and abstract transformation rules. Hence, a manual phase becomes necessary to complete and translate manually the rules into an executable language. This forbids the direct reusability of the learned rules. We aim to learn executable rules. This avoids any manual phases.

**Learning N-to-M rules #2.5**. MTBE approaches progress over time in learning complex rules. The complexity appears in the number of source and target elements by rule. The initiation of learning rules from examples was made by (Varró, 2006). His approach learned rules of 1-1 type. This stands for one element in the left-hand side of the rule and one element in the right-hand side of the rule (one-to-one). Today MTBE approaches, are able to learn m-n rules (many-to-many) (Faunes et al, 2013). In order to challenge existing MTBE approaches, we require to learn rules of n-m types. In addition, we are dealing with in-place transformation rather than out-place transformation. When the transformation manipulation comes directly on source model, the n and m numbers of the learned rules are going to be bigger in comparison to the out-place transformation rules numbers.

**Efficiency #2.6**. Our last requirement is about the quality of the learned rules. This appears on two things. The first thing is the ability to produce the correct target models. This means that all expected refactorings are going to be correctly applied. The second thing is the appearance of refactorings in the rules structure. This ensures that the resulting model is produced with the least number of modifications, thus mimicking an efficient manual refactoring application.

---

***Contribution #3*** The third paper of this thesis tackles the last contribution. We consider rule learning to be an optimization problem and we solve it by applying genetic programming (Langdon et al, 2008). Our algorithm searches for refactoring rules that best match the examples provided. We hypothesize that companies/experts in the field can provide concrete examples of refactoring artifacts. The model is considered a study artifact in the second scenario. Our third contribution, which covers the scenario where knowledge is implicit, is summarized in: Learning refactoring rules by using genetic programming on examples.

---

## 4. Thesis Structure

The rest of this document is structured around five chapters and a conclusion.

**Chapter II:** mainly concerns the state of the art of refactoring. It begins by giving the main definitions about the context. It then presents the state of the art of the approaches which participate in the application of the refactoring of MDE artifacts, mainly those which use the design patterns. Then, it discusses state of the art of approaches that learn refactoring through the use of examples.

**Chapter III:** outlines the scenarios and proposed approaches.

**Chapter IV:** presents the first article of design pattern detection. The approach here is based on rules that represent the design patterns.

**Chapter V:** presents the second article of design pattern detection. The approach uses string matching techniques while model transformations, as well as pattern participants, are both represented as sequences of strings.

**Chapter VI:** presents the third article of this thesis. The proposed approach covers the automatic learning of refactoring, in form of model transformations able to produce the refactoring of an artifact model.

**Chapter VII:** a **conclusion** broadly summarizes our work.

# Chapter 2

## State of the Art

The main objective of the thesis is to improve refactoring automation in MDE. We organize the state-of-the-art following the structure in Fig. 2.1. Each rectangle in the figure refers to a section in this chapter, indicated by the section number.

Before discussing the related work on refactoring in MDE, we introduce the basic concepts involved in our work in Section 1. Then, we separate this chapter according to the two scenarios mentioned in the previous chapter: whether refactoring knowledge is explicit or implicit. In the former case, we consider refactoring knowledge explicitly available through design patterns. Thus, we discuss related work on design pattern detection in software engineering in Section 2. Then, we cover the refactoring of two main artifacts in MDE namely, models (Section 3) and model transformation (Section 4). Most of existing studies focused mainly on model refactoring. We consider the studies not relying on any form of design pattern in Section 3.1 and the studies that are based on design patterns and anti-patterns for model refactoring in Section 3.2.

When refactoring knowledge is not explicitly available, we are interested in learning the refactoring from examples. Examples represent the implicit knowledge. Refactoring in MDE being seen as a model transformation (Balogh and Varró, 2009), we target, in Section 5, approaches that derive model transformation from examples. These approaches can be categorized in two groups, Model Transformation By-Example (MTBE) ( Section 5.2) and Model Transformation By-Demonstration (MTBD) ( Section 5.3).

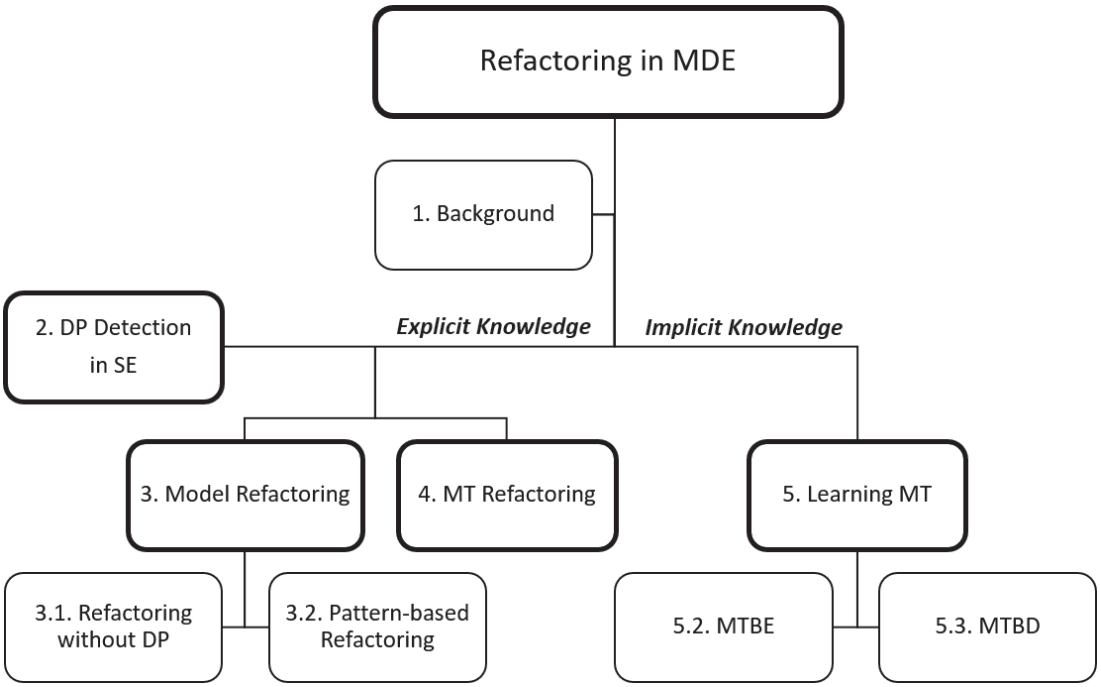



**Figure 2.1.** State of the Art Overview

# 1. Background and Definitions

This section introduces the main concepts in MDE, necessary to understand our contributions.

## 1.1. Model-Driven Engineering

MDE is a paradigm that promises to reduce the complexity of software by intensive use of models and their automatic transformations (France and Rumpe, 2007). The basic idea of MDE is that software development can be simplified by the use of high-level abstraction models (domain models) representing a system and by the application of automatic transformations producing low-level abstraction models (implementation of models) such as source code and test sets (Kelly and Tolvanen, 2008).

## 1.2. Model

A model is an abstraction of a system built for a specific purpose. We then say the model represents the system. A model is an abstraction in that it contains a limited set of information about a system. It is built for a specific purpose and the information it contains is chosen to be relevant to a specific use to which the model will be put (Muller, 2006).

### 1.3. Metamodel

According to Kühne, the metamodel defines the linguistic types and their relations (Kühne, 2006). A metamodel is defined using a metamodeling language, such as UML. In practice, we define metamodels using UML class diagrams, where classes represent the type concepts, attributes represent their properties, and associations represent their relations.

### 1.4. Modeling Languages

A modeling language is a formal language from which models can be instantiated. A modeling language comprises a syntax and its semantics (Harel and Rumpe, 2004). The metamodel defines the abstract syntax of the language. The concrete syntax can be represented graphically, textually, or a combination of both. A modeling language can be dedicated to a certain field or application domain, in which case it is called a domain-specific language (DSL) (Kelly and Tolvanen, 2008). In contrast, general-purpose modeling languages are applicable in any domain, such as UML and Petri nets.

### 1.5. Model Transformation

In MDE, model transformation is the automatic manipulation of a model according to a specification defined at the metamodel level, as shown in, Fig. 2.2 (borrowed from (Lúcio et al, 2014)). Model transformation is called *out-place* when it transforms a source model into a destination model different from the source model. An example of out-place transformation is when we transform a UML activity diagram into its semantically equivalent Petri nets (Syriani and Ergin, 2012). It is said to be *in-place* when it changes the source model itself without producing a different target model. An example of in-place transformation is when we refactor a domain-specific model (Toyoshima et al, 2015).

Model transformation is called *exogenous* when it is expressed between models conforming to different languages. *i.e.* source and target models conform to different metamodels. When the transformation affects models expressed in the same language, *i.e.* source and target models conform to the same metamodel, it is called *endogenous*.

Transformation rules have a preponderant place in the model transformation mechanism. According to Levendovski *et al.* (Levendovszky et al, 2002), these rules are assimilated to the

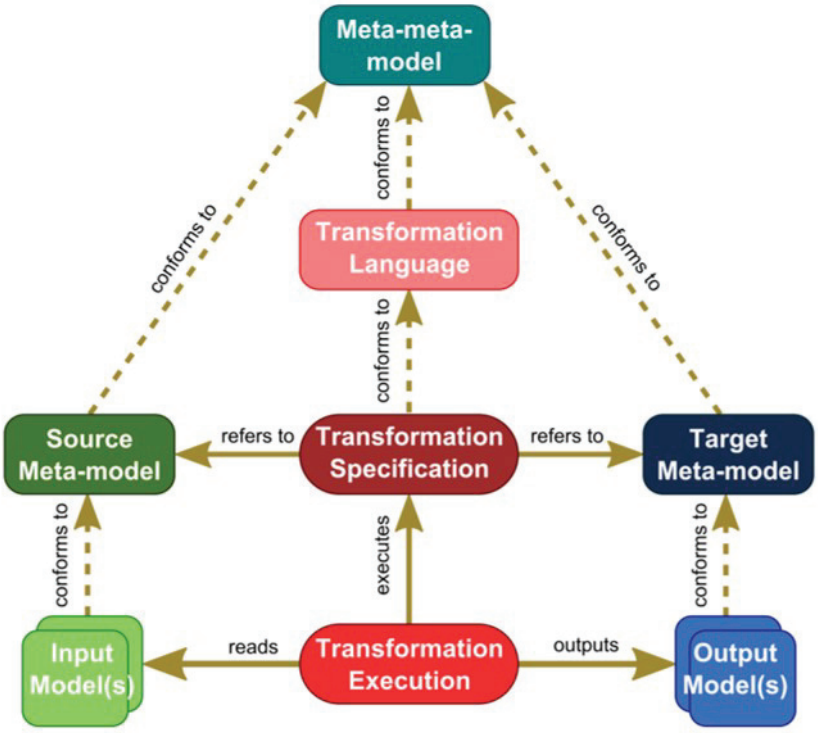



**Figure 2.2.** Model Transformation (Lúcio et al, 2014)

notion of *mapping*. In 2008, Deguil (Deguil, 2008) proposed four types of model mappings: "1-1" type mappings use relations which associate an element of a source model with one and only one element of a target model. "1-m" type mappings associate an element of the source model with several elements of the target model. Conversely, mappings of type "n-1" associate several elements of the source model with a single element of the target model. Finally, "n-m" type mappings associate several elements of the source model with several elements of the target model.

### 1.6. Design Pattern

Some of the research contributions presented in this thesis involve the notion of design patterns. A design pattern is a general reusable solution to a common problem in software design. Design patterns capture some of the best solutions to design problems, at different levels of abstraction, in forms designed to facilitate their reuse (Gamma et al, 1994). They were introduced in the mid-1990s as a catalog of common solutions to common design problems and are considered standards for 'good' software designs (Ramasamy et al, 2015).

Since their appearance, the patterns have aroused a lot of interest. Some studies have focused on the classification, comparison, and implementation of patterns. Others have attempted to formally specify patterns and/or their application.

### 1.7. Design Patterns for Model

The patterns presented by Gamma *et al.* represent a solution for source code as well as models. This is also because models are generally an abstraction of the source code. However, researchers acknowledge that there can be context specific practical pattern solutions to recurring real world design problems (Atwood, 2006).

van der Aalst and ter Hofstede collected a fairly complete set of workflow patterns (van der Aalst and ter Hofstede, 2005). Based on these patterns they evaluated several workflow products and detected considerable differences in their ability to capture control flows for non-trivial workflow processes. The workflow language YAWL (van der Aalst and ter Hofstede, 2005), was proposed by the same authors.

Brambilla *et al.* (Brambilla et al, 2011) presented a gallery of social BPM design patterns that represent reusable solutions to recurrent process socialization requirements, and a model-to-model and mode-to-code transformation technology. These patterns are part of a process design methodology, supported by a tool suite, for addressing the extension of business processes with social features.

### 1.8. Design Patterns for Model Transformation

There is a large body of research on design patterns for models, and especially for UML models. However, little attention has been paid to the design patterns for model transformations. In the mid-2000s, several works proposed design patterns for the transformation of models. Indeed, in 2005, Agrawal *et al.*. (Agrawal et al, 2005) have defined design patterns for the graph transformation described in a specific language of model transformations. Later in 2008, Iacob *et al.*. (Iacob et al, 2008) have defined other design patterns for out-place transformations. Simultaneously in 2009, Levendovszky *et al.*. (Levendovszky et al, 2009) have proposed domain design patterns for model transformation and for different DSLs.

More recently (2014), Lano and Kolahdouz-Rahimi (Lano and Kolahdouz-Rahimi, 2014) presented the most comprehensive study of design patterns of a model transformation and defined a catalog of 20 patterns classified into five categories. They have shown that these

patterns reduce complexity and execution time, and improve the flexibility and modularity of model transformations. Although the objective and the conditions of application of each pattern are rigorously described, they chose to define the solution part of the pattern using a formal specification without proving details on how to implement them.

In 2016, Ergin *et al.*. (Ergin et al, 2016) proposed a modeling language, called DelTa, with a graphical and textual representation (Ergin and Syriani, 2014), in order to facilitate the understanding of the design patterns of Lano and Kolahdouz-Rahimi. Furthermore, DelTa allows the automatic instantiation of design patterns in model transformation implementations. Ergin and Eugene also presented new design patterns (Ergin and Syriani, 2013), such as the iterative modification of a model until reaching a fixed point, or the execution of a modeling language by translating it to another modeling language for simulation.

A model transformation design pattern consists of participants and their scheduling scheme. Participants represent rule templates that shall be implemented in a concrete model transformation. The scheduling scheme defines the order in which the rules are applied when a transformation is executed. We provide further details on model transformation design patterns in the first two articles of this thesis.

## 2. Design Pattern Detection in Software Engineering

Design pattern detection has been mainly explored in object-oriented programs (Al-Obeidallah et al, 2016a). Different methodologies, such as database queries-based, UML structure, graph and matrix-based, metrics-based (Priya, 2014), string-matching-based and ontology-based are used for the detection.

One promising solution is the use of rules as part of the detection process. (Kramer and Prechelt, 1996) uses Prolog rules to recover structural design patterns. Design patterns are described using a variation of Prolog predicates and information about program elements and their roles are represented as Prolog facts. (Niere et al, 2002) present an approach that uses an algorithm that exploits an abstract syntax graph (ASG) that represent the program. The algorithm uses sets of graph transformation rules that represent Design patterns. (Arcelli et al, 2009) defined sets of rules implemented inform of queries to identify design patterns. Rules are characterized by weights indicating their importance in the detection of a specific design pattern. (Rasool et al, 2010) present an approach that is based on annotations, regular

expressions and database queries. They define the varying features of patterns and apply rules to match these features with the source code elements. (Alnusair et al, 2014) present a reverse-engineering approach that enhances program understanding through the automatic recovery of design patterns from source code. The authors conclude that effective and flexible detection of design patterns can be achieved without using hard-coded heuristics. (Martino and Esposito, 2016) use ontologies to cover the semantical and behavioral parts of the design patterns. Their approach extracts Prolog facts from both, formal pattern representation and UML documentation of software artifacts. The sets of facts are analyzed and the recognition rules are applied. (Al-Obeidallah et al, 2018) present a multiple phases approach for design patterns recovery based on structural and method signature features. The approach uses rules to match the method signature of the candidate design instances to that of the subject program. After all, rule-based approaches are a very suitable solution for detecting model transformation design patterns. Because rules can be useful not only for detecting pattern participants, but also for ensuring the validity of the chaining of participant instances with respect to the pattern scheduling.

In terms of performance, most existing design pattern detection techniques have performance issues when the size of the input program increases. (Antoniol et al, 1998) and (Guéhéneuc et al, 2004) tried to improve the efficiency of the detection by the use of metrics. Their idea is to reduce the search space by removing entities that do not participate in a design pattern according to expected metrics values. A similar problem in bioinformatics is to localize different genes in long anonymous DNA sequences. Efficient solutions exist, yet, they are designed for strings. Inspired by the bioinformatics solutions, (Kaczor et al, 2010) chose to represent the input programs, as well as design patterns, as strings. This allows the use of bioinformatics algorithms. Their approach identifies design patterns in source code. An efficient bit-vector algorithm that finds exact and approximate occurrences of design patterns in a program was used. The approach takes as input design patterns and generates their string encoding before starting the identification process. A new design pattern or pattern variants could be treated without any human interaction or any update on the approach. This makes it a completely generic approach.

# 3. Model Refactoring

We now present the state-of-the-art regarding model refactoring in MDE.

## 3.1. Refactoring Without Design Pattern

Starting from the early 2000s, UML models were targeted as candidates for refactoring according to Sunyé *et al.* (Astels, 2002; Boger et al, 2002; Sunyé et al, 2001). This was particularly the case for class diagrams because many existing refactoring rules in object-oriented programming were directly adapted to those diagrams.

In 2005, Van Kempen *et al.* (Van Kempen et al, 2005) used the CSP (Constraint Satisfaction Problem) formalism to describe the refactorings of state-transition diagrams, and showed how this formalism can be used to verify that refactoring effectively preserves behavior. In the same year, Gheyiet *et al.*. (Rohit et al, 2005) suggested to specify the refactoring of models using Alloy, a modeling language used for formal specification. It can be used to demonstrate the preservation of the semantic properties of model refactorings.

Different formalisms have been proposed to understand and explore the refactoring of models. Most of these approaches suggest declarative expression of these refactorings (Pretschner and Prenninger, 2007; Straeten and D'Hondt, 2006). In 2006-2007, Biermannet *et al.*. (Biermann et al, 2006) and Mens *et al.*. (Mens et al, 2007) used graph transformation theory to specify the refactoring of models. They used their formal properties to analyze and reason about these refactorings. In this period, Bouden (Saliha, 2006), studied the preservation of traceability between model refactoring and code refactorings. She proposed a catalog of 55 primitive and composite refactorings at the model level and defined for each refactoring preconditions, post-conditions, and actions, expressed in pseudo code and necessary to ensure the preservation of the model behavior.

## 3.2. Design Pattern-based Refactoring

In the following, we review several works that have been interested in design patterns as refactoring opportunities and those that have been interested in automating their application in model refactoring.

3.2.1. *Opportunities' Detection*

A model refactoring opportunity can be understood in several ways. In what follows, we consider that the opportunities are patterns, anti-patterns or models of smells. There are many known good and bad practices in software engineering that prove to benefit to or harm software quality. In the area of software design and the quality of the source code/design model, these best practices take generally the form of design patterns. On the other hand, bad practices are commonly referred as anti-patterns or bad (model) smells. To define model refactoring opportunities, many detection approaches have been proposed. We group them into two main categories: those based on design patterns and those that aim to detect anti-patterns (Brown et al, 1998) and models smells.

(1) Design Pattern-based Detection

One of the first works that motivated the idea of detecting design patterns was proposed in 1996 by Brown (Brown, 1996). He took the first step towards automating the detection of design patterns by applying reverse-engineering to code written in Smalltalk to facilitate the detection of four patterns from the Gamma *et al.* catalog (Gamma et al, 1994).

Later in 2009, Bouhours *et al.*. (Bouhours et al, 2009) defined the concept of 'Spoiled Pattern' to specify the problems of refactoring patterns. It is an abstraction of an alternative solution that is less optimal than the design pattern itself (optimal solution).

Ballis and Baruzzo (Ballis et al, 2008) have proposed a rule-based algorithm and a general text language to identify all instances of a pattern in the graph of UML models. This text language makes it possible to express patterns, anti-patterns, smells and other design heuristics at the same time. Esposito and Martino (Martino and Esposito, 2016) have seen that this work has certain limitations. They claimed that the proposed textual language is incapable of describing the intents of patterns, the scopes and the contexts of an application, and that the extension of this language also is difficult. Thus, in 2015, they proposed an approach (Martino and Esposito, 2016) that covers these limitations.

Most of the existing design pattern detection approaches use an intermediate formalism to represent their artifacts and patterns (Al-Obeidallah et al, 2016b; Misbhauddin

and Alshayeb, 2015). Software code and design models are both transformed into this intermediate representation. Hence, the detection of design patterns at the model level can reuse the detection of design patterns at the code level. Indeed, most of the existing design pattern detection approaches for software code consider that their approaches are adequate solutions for models. For this, Section 2 covers the detection approach for both code and model artifacts.

(2) Detection based on Anti-patterns and Smells

In 1998, Brown *et al.*. (Brown et al, 1998) introduced the concept of anti-patterns for the first time by describing 40 design smells. 18 of them are development and architecture anti-patterns. Among the latter, five treat the refactoring of models. A year after, Flower *et al.* proposed 22 code smells (Fowler, 1999b). 10 of them have received an extension and were discussed with regards to model refactoring, although some of these smells could not be exclusively applied to UML models (such as Long Method, Switch Statements, Comments) (Misbhauddin and Alshayeb, 2015). In (Singh and Kaur, 2018), Singh and Kaur present a systematic literature review of refactoring for code smells. They showed the current status, approaches, and tools of refactoring with respect to code smells and anti-patterns.

Misbhauddin and Alshayeb in 2015 (Misbhauddin and Alshayeb, 2015) found that most rule-based approaches define and detect smells in UML class diagrams (Akiyama et al, 2011; Dao et al, 2006; Dobrzanski and Kuzniarz, 2006; Llano and Pooley, 2009; Štolc and Polášek, 2010). Other approaches apply to sequence diagrams, component diagrams, and use-case diagrams. Dobrzański and Kuźniarz claim that the majority of work on the refactoring of UML models (Boger et al, 2002; Porres, 2003; Sunyé et al, 2001; Zhang et al, 2005) concerns non-executable design models. In their work (Dobrzanski and Kuzniarz, 2006), they present a systematic approach to specifying the refactoring of executable UML models and their associated bad smells. Their approach describes the smell of an intermediate model in UML sequence diagrams.

El-Attar and Miller worked on use-case models for which they identified 26 anti-patterns (El-Attar and Miller, 2010). In addition, their work provided a query-based approach capable of detecting these anti-patterns. El-Attar and Ali Khan provided another approach (Khan and El-Attar, 2016) based on a model transformation that

allows detecting instances of anti-patterns in a given use-case model and apply its refactoring in a suitable way.

For sequence diagrams, one of the few works is one of Liu *et al.*. (Liu et al, 2006). They proposed an approach to detect duplication by converting a two-dimensional sequence diagram into a one-dimensional array, then the array into a suffix tree. Then, an algorithm makes it possible to identify the common prefixes, from the suffix tree, in the form of diagrams of the reusable sequences that will be, subsequently, considered as refactoring candidates. Ren *et al.*. (Ren et al, 2003) also proposed an approach for the detection of clones in sequence diagrams and their refactoring.

### 3.2.2. *Refactoring Application*

In 2003 France *et al.*. (France et al, 2003) proposed a pattern-based approach as a tool to support the refactoring of UML models. In their approach, the specification of patterns should include the specifications of the problem solution, as well as the specification of the problem-to-solution transformations. However, their approach requires that a specification of a transformation be previously written using a specific domain language and that it does not allow the user to freely experience a transformation that has not the specific objective of restructuring into a design pattern (Verebi, 2015).

In 2008, in order to improve the quality of design, Kim (Kim, 2008) proposed an approach that uses design patterns to refactor software models. As in (France et al, 2003), the design pattern is defined by three components: the specifications of the problem, the solution, and the transformation. In this work, Kim followed the work of Schulz *et al.*. (Schulz et al, 1998) and Cinneide (Cinnéide, 2001) to specify the problem targeted by the design patterns. This is used to check the applicability of a pattern. Kim had also developed a prototype that supports class diagram refactoring.

In 2009, Shahir *et al.*. (Shahir et al, 2009) proposed an approach based on design patterns to transform/refactor models. The approach was demonstrated through a case study, where five previously defined design patterns were applied to a model to ultimately lead to an improved model of the software system. In 2015, Ben Ammar and Bhiri (Ammar and Bhiri, 2015) proposed a pattern-based refactoring approach to introduce the association relationships of UML models. In their work, they detail a refactoring pattern that allows

the introduction of an association relationship between two existing classes. The pattern is applied to class diagrams and state machine diagrams to obtain high-quality UML models.

In 2017, Arcelli and Di Pompeo (Arcelli and Pompeo, 2017) introduced a preliminary refactoring approach based on anti-patterns of software systems. The approach is driven by the application of design patterns and focuses on the refactoring of software artifacts (i.e., models, code). It aims to eliminate possible performance anti-patterns by applying design patterns. The authors describe a procedure that selects a design pattern to be applied according to its rank (degree of classification) with respect to a score assigned to each design pattern, quantifying the probability that the pattern removes the corresponding anti-pattern. They also provide a preliminary validation of the approach, showing how the selection procedure works on three design patterns to remove empty sequences from the performance anti-pattern.

### 3.3. Discussion

As shown at the beginning of this Section 3.1, very few approaches target model refactoring without considering design patterns. Researchers see that detecting refactoring opportunities is suitable by detecting design patterns/anti-patterns and bad smells.

Most detection approaches target the patterns of Gamma *et al.*. in object-oriented programs (Gamma et al, 1994). These approaches consider mainly structural patterns because they can be detected by matching the structure of the code, expressed in a high level of abstraction, to one of the patterns (Dong et al, 2008). To improve detection, some projects combine several strategies as in (Rasool et al, 2010). The detection of behavior patterns has also attracted interest from the research community. In De Lucia *et al.*. (De Lucia et al, 2010), the authors use model verification to improve the detection of behavior patterns. A work similar to this one is that of (Guéhéneuc et al, 2004), The authors first identify the key participants of the pattern using a machine learning technique. Then they check the other participants of the model and the relationships between them.

Pattern-based approaches (Section 3.2.1 (1)) are very popular. They detect refactoring opportunities by looking for design problems in the model and they suggest their corrections in the form of design patterns.

The other approaches (Section 3.2.1 (2)) identify both model smells and anti-patterns using a declarative rule definition. These rules are defined manually to identify the symptoms that characterize the smell. It should be noted that some approaches (Ballis et al, 2008) are also able to identify the design patterns. As mentioned in (Misbhauddin and Alshayeb, 2015), most of these studies define and detect smells in UML class diagrams (Akiyama et al, 2011; Dao et al, 2006; Dobrzanski and Kuzniarz, 2006; Llano and Pooley, 2009; Štolc and Polášek, 2010). Other approaches apply to sequence diagrams, component diagrams and use case diagrams.

However, none of the mentioned approaches can refactor other artifacts than models. We have studied these approaches with the hope of finding an approach that could be generalized and adapted to refactor other MDE artifacts. Unfortunately, our observation did not reveal any clear proposal for an approach that can be generalized to refactor other MDE artifacts.

# 4. Model Transformation Refactoring

In this section, we discuss the existing work on refactoring of model transformations. The two main families of contributions are approaches based on heuristic search and those driven by smells.

## 4.1. Search-based Refactoring

In 2012, Wimmer *et al.*. (Wimmer et al, 2012) proposed a dedicated catalog of refactorings for model transformations. The objective of this catalog is to improve the quality of model transformations. The refactorings were explored while analyzing examples of existing transformations defined in ATL (Jouault et al, 2008). The application of a model transformation refactoring based on this catalog is a semi-automatic process that requires the presence of a user. In order to automate this process, in 2016 Alkhazi *et al.* (Alkhazi et al, 2016) provided an approach to find the refactoring space for a model transformation. The approach is based on a multi-objective optimization algorithm that recommends the best refactoring sequence (ex: extraction rule, merge rules, etc.) optimizing a set of quality metrics based on ATL (ex: number of rules, coupling, etc.). In (Alkhazi et al, 2020), they proposed a novel set of quality attributes to evaluate refactored ATL programs.

### 4.2. Smells-based Refactoring

In 2012–2013, Taentzer *et al.*. (Taentzer et al, 2012) and Tichy *et al.*. (Tichy et al, 2013) presented their approaches in order to improve the quality of model transformations based on graphs. Taentzer *et al.*. presented a set of refactoring patterns to reduce the effect of bad performance smells that Tichy *et al.* were able to detect. Both approaches are based on the transformation language Henshin.

In 2011, Tairas and Cabot (Tairas and Cabot, 2011) provided an evaluation of cloning in OCL expressions as an initial step toward a broader understanding of cloning in DSLs. Their evaluation covered ATL transformations because it contains OCL-like expressions. In 2016, Struber *et al.*. (Strüber et al, 2016a) presented an approach for the identification of clone detection for graph-based model transformation languages. They consider Type I and Type II clones (Koschke, 2006), which are regularly produced when the rules are produced by copy and paste. It also contains an identification of five main needs for a clone detection technique for graph-based model transformations. In the same year, Struber *et al.*. (Strüber et al, 2016b) proposed a refactoring approach based on the fusion of rules for this same type of transformation. In order to select groups of rules to unify, their approach uses clone detection to identify overlapping parts of the rules and their clustering.

### 4.3. Discussion

A limited number of approaches were tailored to the refactoring of model transformations. None was found that use design patterns for refactoring, excepted for approaches that correct smells. As explained previously, detecting design patterns can be seen as detecting refactoring opportunities. To the best of our knowledge, our contributions (Mokaddem et al, 2021, 2016) represent the first solutions of the detection of design patterns in model transformations.

We believe that detecting refactoring opportunities through the detection of design patterns can open new possibilities to improve the state of the art. This is why we explore this avenue in the following chapters.

# 5. Derivation of Model Manipulation Artifacts

Most of the existing work on the derivation of model refactoring solutions is based on search-based techniques (Mens and Gorp, 2006). We present the significant contributions in the following subsection. However, there are alternative techniques that were used to derive other kinds of model manipulation artifacts. We discuss, in particular, in this section, techniques that learn model transformations from examples and those that target model transformation by demonstration.

## 5.1. Search-based Refactoring Derivation

In 2007, Bodhuin *et al.* (Bodhuin et al, 2007) present SORMASA, a tool that assists the user by suggesting a set of model refactorings. It relies on a mono-objective evolutionary algorithm aiming at increasing cohesion and reducing coupling of class models. Later, Jensen and Cheng (Jensen and Cheng, 2010) present an approach that attempts to introduce design patterns in class diagrams by optimizing specific software design metrics. To ensure the behavior preservation, Mansoor *et al.* (Mansoor et al, 2017) developed a multi-objective evolutionary algorithm that searches for the best trade-off between quality attributes for class and activity diagrams.

## 5.2. Learning Model Transformation from Example

Model transformation by example (MTBE) aim at inferring a model transformation from a set of examples of this transformation. The examples are given in the form of pairs where each has an input model and its corresponding transformed model. They are often accompanied by the transformation traces indicating the fine-grained correspondence between fragments of the input and output models.

The transformation rules are automatically or semi-automatically extracted from the provided examples (Saada et al, 2012) instead of writing them manually. Fig. 2.3 describes this process.

### 5.2.1. *Extraction of Transformation Rules*

In 2006, Varró (Varró, 2006) proposed a first approach to MTBE. In his work, he derives transformation rules from a set of prototypical examples of transformations with interrelated

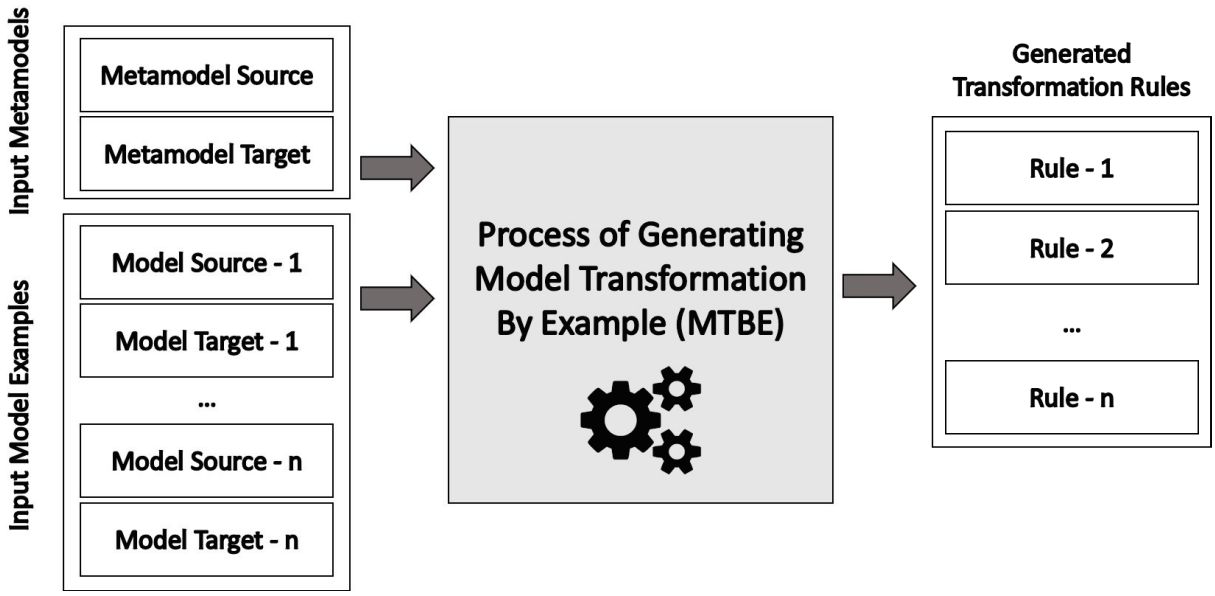



**Figure 2.3.** Description of the MTBE Process (updated from (García-Magariño et al, 2009))

models. Examples are provided by the user. This semi-automatic and iterative process begins by analyzing the mappings between the source and target models of the examples with their respective metamodels. The transformation rules are finally produced using an ad-hoc algorithm. In 2007, Wimmer *et al.* proposed in (Wimmer et al, 2007) an approach similar to that proposed by Varró in (Varró, 2006), with the difference that Wimmer produces executable transformation rules written in ATL. The limitation of the two approaches is that they can only derive "1-1" transformation rules. These kinds of rules can test the presence or absence of elements in source models. However, this level of expressiveness is insufficient for many common transformation problems (*e.g.*, transformation from UML class diagram to relational diagram).

In 2010, Kessentini *et al.* (Kessentini et al, 2010) proposed an approach to derive "1-m" transformation rules from examples of transformations. The process of deriving transformation rules is established by producing "1-m" mappings between the elements of the source metamodel and the elements of the target metamodel. The approach is based on a meta-heuristic and hybrid research using the particle swarm optimization (PSO) combined with the simulated annealing technique. In 2012, Saada *et al.* (Saada et al, 2012) succeeded in having the same type of rule ("1-m") by extending the work of Dolques *et al.* (Dolques et al, 2010). Indeed, a transformation rule derivation process works in two stages. In the first step, the approach by Dolques is used to learn the patterns of transformations from

examples and traces of transformations. In the second step, the learned patterns are analyzed and those considered relevant are selected. The selected patterns are then translated into transformation rules implemented in Jess (Java Expert System Shell) (Hill, 2003). Although these approaches advanced the state of the art by inferring "1-m" rules, the absence of transformation rules "n-m" for both approaches remains a limitation.

A group of approaches have overcome this limitation by successfully generating "n-m" transformation rules. To this end, these approaches adopted different strategies. In 2008, Strommer and Wimmer (Strommer and Wimmer, 2008) describe a model mapping language and a metamodel mapping language, as well as reasoning algorithms to infer metamodel mappings from model mappings. This work is considered as an extension to Wimmer's *et al.* approach (Wimmer et al, 2007). In 2009, Balogh *et al.* (Balogh and Varró, 2009) improved Varró's original work by using inductive logic programming (ILP) instead of the original ad-hoc heuristic. As for Varró's approach, it uses transformation mappings and produces "n-m" transformation rules from Prolog clauses which are obtained by a semi-automatic process in which the user must add logical assertions until the ILP inference engine can produce the desired transformation rule. A limitation of this approach is that the user must interact with the ILP inference engine.

In 2009, Garcia-Magariño *et al.* (García-Magariño et al, 2009) proposed an algorithm capable of creating "n-m" transformation rules from examples of interconnected models and their metamodels. Transformation rules are created in a generic model transformation language and are later converted to ATL for evaluation purposes.

In 2013, Faunes *et al.* (Faunes et al, 2013) proposed an approach based on genetic programming to automatically learn the rules from prior transformations of pairs of source-target models used as examples. The approach can produce "n-m" transformation rules, which is a remarkable improvement over most other contributions. These transformation rules are executable, which is not the case with the approach of Balogh *et al.* Derived transformation rules, written in Jess, look for non-trivial patterns in the source models and instantiate non-trivial patterns in the target models. Unlike previous approaches, this one does not need fine traces of transformations. It is therefore applicable to a wide spectrum of transformation problems. This approach was evaluated quantitatively and qualitatively on transformations

of structural models. The results showed that the transformation rules produced are operational and correct. Unfortunately, this approach only deals with exogenous transformations, falling short on endogenous transformation used, for example, for refactoring models.

In 2014, Baki *et al.* (Baki et al, 2014) adopted the same approach as Faunes *et al.* to automatically learn the rules of model transformations through the use of genetic programming. Their approaches had many similarities, except that Baki *et al.* was also able to learn implicit and explicit control in these transformations. In a second work (Baki and Sahraoui, 2016) (in 2016), Baki and Sahraoui developed a new approach, where the evaluation process was based on two strategies to reduce the size of the research space and to better explore it, namely multi-step learning and adaptive search.

In 2017, Al-Jamimi and Ahmed (Al-Jamimi and Ahmed, 2017) use ILP to derive transformation rules from given examples of analysis-design pairs. Their approach aims to use the minimal inputs, the source and target models only, to derive "n-m" rules. But it does require a manual task to refine the generated rules. The authors claim that the approach could be used for refactoring. However, their paper does not shows how the approach reacts to any form of refactoring.

#### 5.2.2. *Mappings and Other Forms of Transformations*

In 2008, Kessentini *et al.* (Kessentini et al, 2008) propose another approach to model transformations which, using examples of transformations and their traces, transforms a source model into a target model. It uses an adapted version of the PSO algorithm, and it differs from previous approaches because it does not produce transformation rules. Instead, it directly derives the target model required by analogy with the existing examples.

In 2010, Dolques *et al.* (Dolques et al, 2010) proposed an MTBE approach that produced sets of recurring "n-m" mappings organized in a Welsh lattice. It is based on relational concept analysis (RCA) where the derivation process analyzes an example of a unique transformation and its mappings, as well as the source and target metamodels. In addition to the limitations, as mentioned by the authors, the two do not produce RTs. Thus, the need for traces of transformations also constitutes a limit.

**Table 2.1.** Model Transformation By-Example Approaches

| Approaches | Input | | | Rule | Output |
|---|---|---|---|---|---|
| | Example | Example & Traces | User | | |
| | Fauness *et al.*. [2012; 2013]<br>Al-Jamimi and Ahmed [2017] | Strommer *et al.*. [2007; 2008]<br>Balogh *et al.*. [2009]<br>García-Magariño *et al.*.[2009]<br>Baki *et al.*. [2014; 2016] | Langer et al. [2010]<br>Kehrer *et al.*. [2017] | n-m | |
| | Kassentini *et al.*. [2010] | Saada *et al.*. [2012a; 2012b] | | 1-m | |
| | | Varró *et al.*. [2006]<br>Wimmer *et al.*. [2007] | | 1-1 | |
| | | Kassentini *et al.*. [2008; 2012]<br>Deloques *et al.*. [2010] | Brosch *et al.*. [2009a; 2009b]<br>Sun *et al.*. [2009; 2011] | - | |
| | MTBE | | MTBD | | |

### 5.3. Model Transformation By Demonstration

Instead of inferring MTBE rules, Model Transformation By Demonstration (MTBD) asks users to demonstrate how a model transformation should behave by directly editing (adding, deleting, updating, etc.) a given input model to simulate the model transformation process step-by-step. An inference and recording engine captures all user operations in a transformation task to infer a user's intention. A transformation pattern is generated from this inference to associate a precondition with the sequence of operations necessary for carrying out the transformation. This pattern can be reused to automatically match a precondition to a new instance of the metamodel and repeat the same operations to simulate the transformation process. Unfortunately, MTBD requires a large number of patterns to produce consistent results. In addition, it is not expressive enough to transform a complete source model (Kessentini et al, 2010).

In 2009, Sun *et al.* (Sun and Gray, 2009) proposed a new perspective for the derivation of transformations, namely, the MTBD. In this work, the objective is to generalize the cases of model transformations. However, instead of using the examples, users are asked to demonstrate how model transformation should behave. However, the fact that this approach aims only to carry out an endogenous transformation represents an important limitation. For this, in 2010, Langer *et al.* (Langer et al, 2010) proposed an approach very similar to that of Sun *et al.* with improved management of exogenous transformations. The main

limitation of the two approaches is that they do not produce transformation rules, but rather transformation patterns. As with many approaches, the absence of rigorous validation makes it difficult to assess the performance of this approach in realistic scenarios.

Recently (2017), Kehrer *et al.* (Kehrer et al, 2017) proposed an automatic specification inference based on in-place transformation rules for complex models. Their approach uses the inference of complex rules characterized by conditions of non-applicability of rules, multi-object patterns and global invariants. They illustrate how their approach works by inferring the refactoring operation on UML class diagrams. Their work has many limitations. As an example, they considered a metamodel and a unique transformation rule based on this metamodel. They also defined their own examples of transformation. In addition, some internal details, such as how they identify the corresponding model elements, are generally not known to experts in the field who will be called upon to use their approach.

### 5.4. Discussion

Existing MTBE approaches only partially solve the transformation rules derivation problem (see Table 2.1). Most of them require detailed mappings (traces of transformations) between the source and target models of the examples, which are difficult to provide in certain situations. Others can hardly derive transformation rules that test many constructions in the source model and/or produce many constructions in the target model. However, "n-m" transformation rules, with their complexity, are a necessity in complex transformation problems. Finally, certain approaches produce, non-executable and abstract transformation rules that must be completed and translated manually into an executable language.

By proposing an approach based on genetic programming (Mokaddem et al, 2018), we have succeeded in overcoming these limitations. Faunes *et al.* (Faunes et al, 2013) was the forerunner to this solution. Unlike current approaches, the two approaches does not need traces of fine transformations to produce "n-m" transformation rules. They are therefore applicable to a wide range of transformation problems. Since the learned transformation rules are produced directly in a transformation language, they can be easily tested, improved, and reused. The approaches have been successfully evaluated on well-known transformation problems, which is not the case for many approaches.

Faunes *et al.* (Faunes et al, 2013) focus only on exogenous transformation. The endogenous transformation seems to be deliberately neglected which could create a limitation in their work. According to the previous state of the art, apart from the contributions of Sun (Sun and Gray, 2009), Langer (Langer et al, 2010) and Kehrer (Kehrer et al, 2017), no other contribution claims to support the endogenous transformation. However, our third contribution provide a solution to refactoring models by learning endogenous transformations from examples.

# Chapter 3

## A Vision on Refactoring in MDE

This thesis aims at applying automatic refactoring to MDE artifacts. The majority of studies to date offer specific solutions for the refactoring problem and only for certain MDE artifacts without having a holistic view. In this thesis, we consider the problem as a whole, and we propose refactoring solutions, depending on the context. In the remainder of this chapter, we explain this idea for scenarios 1 and 2 described in Chapter 1.

We start with an overview of the general framework, and then relative to each scenario, we present the proposed contributions within this framework. The details about each contribution will be given in the subsequent chapters as articles.

### Overview

The general framework of this thesis is illustrated in Fig. 3.1. It is structured according to two scenarios of the refactoring problem: **When the refactoring knowledge is explicitly available** (left side of Fig. 3.1) and **when it is only implicit** (right side of Fig. 3.1). In the first scenario, we exploit this knowledge to automate (partially) the refactoring process. In this thesis, we consider, as an example of available knowledge, design patterns. We specifically use design patterns to detect refactoring opportunities. Hence, our specific research contribution for the first scenario is the definition of detection strategies of design patterns in MDE artifacts. For the second scenario, i.e., when the knowledge is not explicitly available, the goal is to recover such knowledge from refactoring examples. Hence, our research contribution for the second scenario is to learn refactoring rules from refactoring examples.

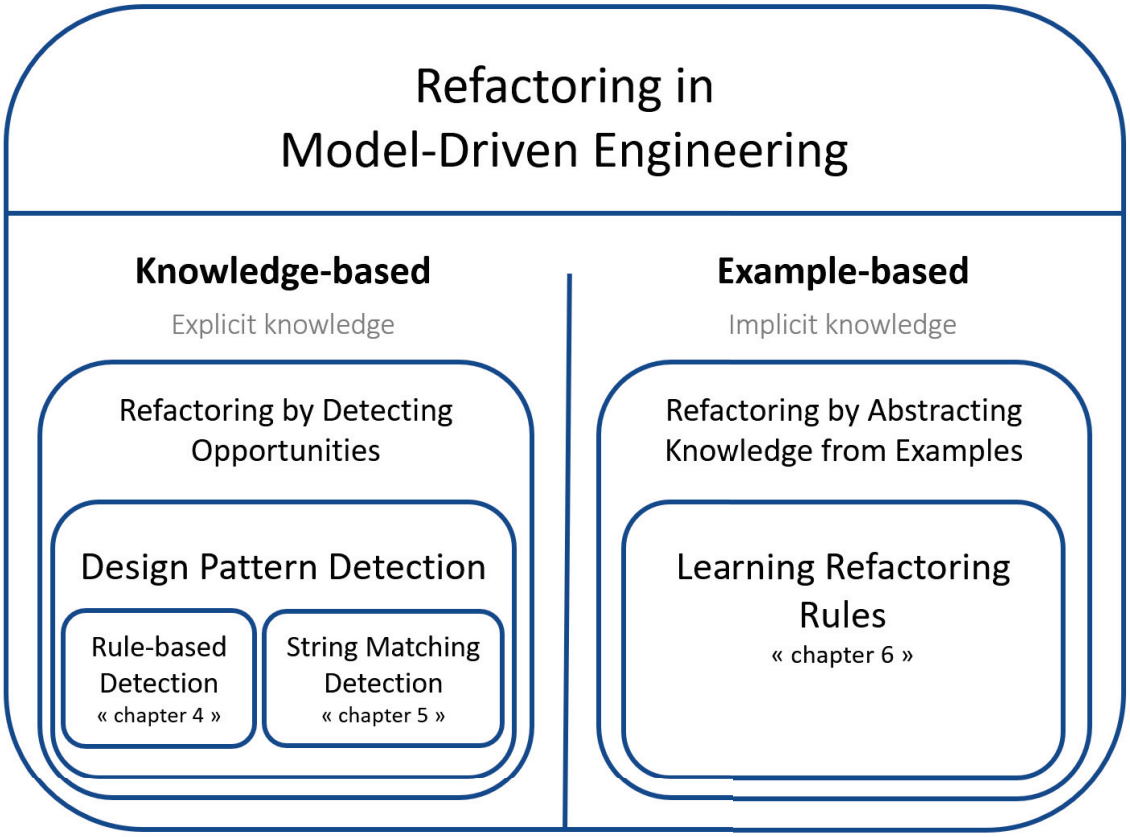



**Figure 3.1.** Research General Overview

## 1. Design Pattern Detection (Scenario 1)

In the first scenario we are considering design patterns as an available explicit knowledge of how to refactor an artifact. Design patterns detection generates refactoring opportunities. Hence, the purpose of this scenario. The detection of design patterns in models, and especially in UML class diagrams, is a well-covered topic in research. However, this is not the case for model transformations. Hence, to explore our idea with scenario 1, we decided to focus on the detection of design patterns in model transformations. This detection aims at determining to which extent instances of a design pattern are implemented in a transformation. The implementation can be at three levels: completely implemented instances, partially implemented instances, and non-implemented instances. Detecting these levels of implementation allows determining refactoring opportunities in different ways. Firstly, design pattern instances come with different variants. Some variants are preferable to others

with respect to some quality characteristics. Thus, detecting a variant of a complete instance may suggest to refactor the corresponding code fragment to use another variant. Secondly and more interestingly, the detection of a partial instance may reveal a design problem. Therefore, it is desirable to refactor the corresponding code fragment to complete the implementation of the detected instance. Finally, for the cases where instances are missing, we consider them as opportunities to apply design patterns. Indeed, the refactoring can be performed to introduce design pattern instances in the transformation. In our work, we consider the detection of the two first implementation levels. The third level is out off the scope of this thesis and will be addressed in a future work. We present in the following, two contributions that target the detection of design patterns for complete and partial instances.

### 1.1. Rule based Detection

The first contribution of design patterns detection in model transformations, described in Chapter 4, is based on rules. The detection of instances of a given design pattern is specified as a set of rules. In the context of model transformations, a design pattern is defined as set of rule templates (call them participants) and a scheduling scheme that specifies how the instances of rule templates should be executed. Therefore, the detection of a design pattern is performed in two steps, each implemented by detection rules, rules that detect the pattern participants and rules that detect the scheduling scheme of the pattern. The pattern participant rules detect all possible instances of the pattern participants. The scheduling scheme rules, then, determine if the individual pattern participants form correct pattern instances. The correct sets of instances are the one that respect the scheduling of the pattern. In other words, the proposed approach detects first the pattern participants through the rules. Then, it ensures that the control flow over these rules corresponds to the scheduling scheme in the design pattern. This approach allows to detect partial and complete design pattern instances together with their variants.

### 1.2. String Matching Detection

The second contribution, described in Chapter 5, uses a string matching technique to search for partial and complete instances of design patterns in model transformations. Both model transformations and design pattern participants are mapped of sequences of elements represented as strings. The proposed approach uses a generic technique that is based on a

bit-vector algorithm. The technique searches for the pattern participants and their variants in the input transformations.

## 2. Refactoring by Learning (Scenario 2)

In the second scenario the explicit refactoring knowledge is not available. This happens often in domain-specific languages for which a critical mass of researchers and/or practitioners is not available to gather a well-established knowledge. Hence, we are limited to refactoring examples that can be seen as an implicit refactoring knowledge. The goal here is to abstract the refactoring knowledge from examples to make it explicitly available. The knowledge abstraction is performed through a learning process. We illustrate this idea on the case of model refactoring. In this case, examples that can be provided by domain experts, for instance, are pairs of models. Each pair contains the model version before refactoring and the version after a refactoring performed manually. The research challenge here is how to infer the refactoring knowledge from these examples. Knowledge is inferred as a set of rules that represent an in-place transformation. To this end, in Chapter 6, we define a genetic-programming-based approach to conduct the learning. The approach is agnostic with respect to the modeling language as it learns rules for different types of models. The learned rules are executable and do not violate the conformance with the modeling language represented by a metamodel.

# Chapter 4

## Towards Rule-Based Detection of Design Patterns in Model Transformations [1]

Chihab eddine Mokaddem,
Houari Sahraoui and Eugene Syriani
DIRO, Université de Montréal

This article represents the work I did for the first contribution of my thesis.
The co-authors are my supervisors who contributed to the writing.

[1]The content of this chapter has been published in System Analysis and Modeling. Technology-Specific Aspects of Models: 9th International Conference, SAM 2016, Saint-Malo, France, October 3-4, 2016. Reference (Mokaddem et al, 2016)

**Résumé.** La transformation de modèles est au cœur même du paradigme de l'Ingénierie Dirigée par les Modèles. En tant que programmes modernes, ils sont complexes, difficiles à écrire et à tester, et dans l'ensemble, difficiles à comprendre, à maintenir et à réutiliser. Dans d'autres paradigmes, tels que la programmation orientée objet, les patrons de conception jouent un rôle important pour comprendre et réutiliser le code. De nombreux travaux ont été proposés pour détecter des instances de patrons de conception complets à des fins de compréhension et de documentation, mais également des instances de patrons de conception partiels à des fins d'évaluation de la qualité et de refactoring. Récemment, un catalogue de patrons de conception a été proposé pour les transformations de modèles. Dans cet article, nous proposons de détecter ces patrons de conception dans des programmes de transformation de modèles déclaratifs. Notre approche détecte d'abord les règles qui peuvent jouer un rôle dans un modèle de conception. Ensuite, il garantit que le flux de contrôle sur ces règles correspond au schéma d'ordonnancement dans le patron de conception. Notre évaluation préliminaire montre que notre mécanisme de détection est efficace pour les instances complètes et partielles de patrons de conception.

# Towards Rule-Based Detection of Design Patterns in Model Transformations

Chihab eddine Mokaddem, Houari Sahraoui, and Eugene Syriani(✉)

University of Montreal, Montreal, Canada
{mokaddec,sahraoui,syriani}@iro.umontreal.ca

**Abstract.** Model transformations are at the very heart of the Model-Driven Engineering paradigm. As modern programs, they are complex, difficult to write and test, and overall, difficult to understand, maintain, and reuse. In other paradigms, such as object-oriented programming, design patterns play an important role for understanding and reusing code. Many works have been proposed to detect complete design pattern instances for understanding and documentation purposes, but also partial design pattern instances for quality assessment and refactoring purposes. Recently, a catalog of design patterns has been proposed for model transformations. In this paper, we propose to detect these design patterns in declarative model transformation programs. Our approach first detects the rules that may play a role in a design pattern. Then, it ensures that the control flow over these rules corresponds to the scheduling scheme in the design pattern. Our preliminary evaluation shows that our detection mechanism is effective for both complete and partial instances of design patterns.

## 1 Introduction

Model-driven engineering (MDE) is a recent software development approach that is rapidly growing in popularity [14]. At its core, it makes intensive use of models as a means for automation and reuse. MDE developers use model transformations to perform operations on models, such as: evolving, refactoring, and simulating them [16]. Model transformations, which uses generally a rule-based declarative paradigm [9], are still manually developed. Therefore, like any hand-written software programs, model transformations must be well-designed and implemented in order to be understandable by other developers, be re-used in other projects, and reduce maintenance efforts.

In other paradigms, such as object-oriented programming (OOP), design patterns play an important role in software design [13]. They are proven solutions to recurring design problems that complement practices of developers. Design patterns are described at a higher level of abstraction than the implementation language to ease communication and comprehension. They are considered as micro-architecture building blocks from which more complex designs can be built, thus promoting modularity and reuse. Recently, Lano et al. proposed a thorough catalog of over 20 design patterns for model transformations [17]. They

© Springer International Publishing AG 2016
J. Grabowski and S. Herbold (Eds.): SAM 2016, LNCS 9959, pp. 1–15, 2016.
DOI: 10.1007/978-3-319-46613-2_14

showed that these design patterns reduce complexity and execution time, as well as improve the flexibility and modularity of model transformations. Although the intent and application conditions of each pattern are described rigorously, they chose to define the solution part of the design pattern using a formal notation. To facilitate their understanding for model transformation engineers and to enable the automatic instantiation of design patterns in model transformation implementations, Ergin et al. [10] proposed a dedicated modeling language DelTa with both a graphical and a textual [12] notation.

With the increasing scale and complexity of utilizing models in MDE, the model transformations developed are also increasing in scale and complexity. Furthermore, as with any software product, model transformations are evolving constantly in development projects. This tends to deteriorate their architecture and design, which is a burden of maintenance tasks. Nevertheless, design patterns expressed in DelTa impose structure thanks to the abstraction they use. Therefore, the identification of design patterns implemented in an existing model transformation can tremendously help the developer in understanding the design, as well as document the transformation [22]. Even if a design pattern was not implemented in its integrity in the model transformation, identifying some of its participants provides valuable feedback to the developer: (1) a missed opportunity to implement it in order to improve the quality, (2) a suggestion to correctly implement it through refactoring, or (3) the presence of a modified version of the design pattern, since any design pattern may be implemented with endless variations [20]. Various design pattern detection mechanisms have proven to be very efficient [2,4,7,22]. However, these techniques have been applied to imperative OOP code. Detecting design patterns on model transformations comes with many challenges because they are described (1) declaratively, (2) at the level of meta-models dealing with types and relations rather than instances, and (3) with non-deterministic execution of rules.

In this paper, we present an approach to detect complete or partial instances of design patterns in concrete model transformation implementations. It is a model finding approach based on a rule engine, where we map model transformations to an abstract representation and design patterns to rules that these representations must satisfy. After identifying individual participants of a design pattern, we verify that the scheduling scheme described in the pattern is satisfied in the transformation. We compute an accuracy score at each detection step that is finally aggregated and reported. We implemented a prototype where we encode design patterns defined with the DelTa language as rules and that automatically maps a complete model transformation implemented in a specific model transformation language to the abstract representation. We report preliminary results that show our detection mechanism is effective for both complete and partial instances of design patterns.

In Sect. 2 we provide the necessary background on model transformation and their design patterns. In Sect. 3 we describe our approach on an example. We report the results on the effectiveness of our approach in Sect. 4. Finally, we conclude in Sect. 5.

## 2 Background

We first review background on model transformations and their design patterns, and then discuss different techniques for detecting design patterns in programs.

### 2.1 Model Transformation

In MDE, a model transformation is the automatic manipulation of a model following a specification defined at the level of metamodels [16]. A model transformation can be outplace, when it produces a target model from a source model, such as in a translation, or it can be inplace when it modifies a model and the result is an updated version of the source model, such as in a simulation. Typically, a model transformation is defined by a set of declarative rules to be executed. A rule consists of a pre-condition and a post-condition pattern. The pre-condition pattern determines the applicability of a rule: it is the pattern that must be found in the input model to apply the rule. Optionally negative patterns may be specified in the pre-condition to inhibit the application of the rule if present. The post-condition imposes the pattern to be found after the rule is applied. Patterns are made up of structural elements (i.e., model fragments) and of constraints on their attributes. Rules follow a scheduling scheme that defines the order in which they are applied when a transformation is executed. The scheduling can be made explicit by the language with a control flow structure partially ordering rules, such as in Henshin. In some languages, such as ATL, rules are scheduled implicitly, depending on the causal dependence between the post-condition of a rule and the pre-condition of another. Features that model transformation languages support are listed in [9]. A comparison of existing model transformation tools can be found in [18]. Possible scheduling schema of model transformations are described in [21].

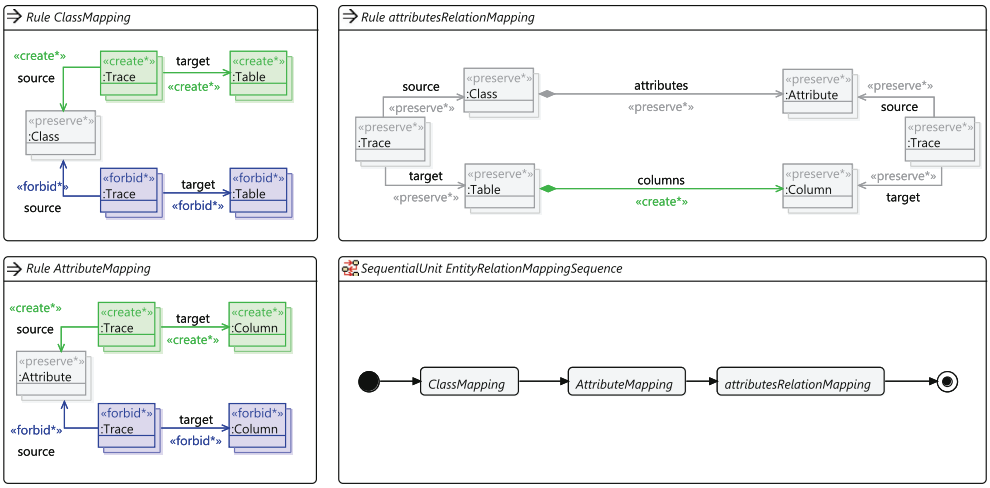



**Fig. 1.** Model transformation of entity relation in Henshin

For example, consider the model transformation defined in the Henshin language in Fig. 1. It contains three rules that are scheduled to execute in sequence, as depicted on the bottom right. This is an excerpt of transformation that creates database tables and columns from a class diagram. The first rule in the top left states that if a class is present then create a table and link it with a trace element unless such a trace already exists for the class.

### 2.2 Model Transformation Design Patterns

A design pattern expresses a means of solving a common model transformation design problem: it describes the transformation structure (rules, condition patterns, and scheduling) that constitute the solution idea. A design pattern includes also a description of the problem which motivated the pattern, how such problems can be detected, and the benefits and negative consequences to consider when using the pattern.

In the mid-2000s, several works proposed design patterns for model transformation. Agrawal et al. [8] defined design patterns for graph transformation described in a specific model transformation language. Iacob et al. [15] defined other design patterns for outplace transformations. Levendovszky et al. [19] proposed domain-specific design patterns for model transformation and different domain-specific languages.

More recently, Lano et al. [17] presented the most comprehensive model transformation design pattern study and defined a catalog of 29 patterns classified into five categories. For example, these include a design pattern to map objects before links, to decompose a transformation into phases based on the target model, the criteria to separate rules so they can be executed in parallel, to ensure that elements created by a rule are unique, or to individually process all nodes of a model recursively.

At the same time, Ergin and Syriani [11] presented similar design patterns, as well as new ones, such as modifying a model iteratively until a fixed point is reached, or the execution of a modeling language by translating it into another modeling language that can be simulated.

### 2.3 DelTa to Describe the Structure of Design Patterns

Lano et al. [17] presented the structure design patterns using a formal language TSPEC in the form of contracts with pre- and post-conditions that a concrete model transformation implementing the pattern should satisfy. However, Ergin and Syriani [12] engineered a domain-specific language, *DelTa*, dedicated to represent the structure of model transformation design patterns. Because an implementation is already available in EMF, we opted to use the DelTa implementations of Lano et al.'s design patterns.

DelTa is a language to define model transformation design patterns with its own syntax and semantics. It is independent from existing model transformation languages. In terms of abstraction, DelTa borrows concepts from various MTLs

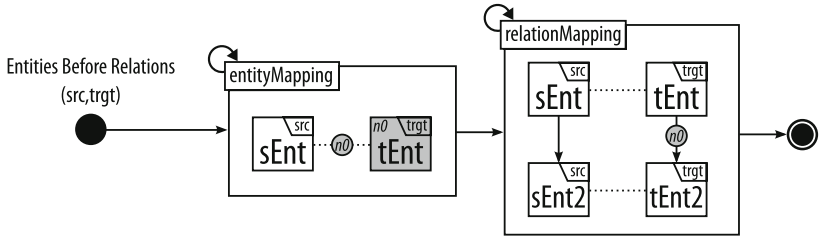



**Fig. 2.** Entities before relations design pattern

to create a more understandable and common language. Figure 2 represents a model transformation design pattern in its graphical syntax as described in [10].

A DelTa model specifies the minimal rules (the large rectangles) and necessary rule scheduling (the connections between them) that a concrete model transformation implementing it should have. Rules consist of the minimal constraints and actions on elements of the metamodel that concrete transformation rules implementing them should specify. Constraints and actions refer to variables that are typed as entities (rectangles like `sEnt`) or relations (arrows between entities) of a metamodel, or traces (dotted lines). For example, in rule *entityMapping*, there is a constraint stating that there must be an entity (`sEnt`). Furthermore, the *n0* symbol on rule elements indicates that trace and the entity `tEnt` are part of a negative constraint. These two entities come from different metamodels (`src` and `trgt`). In DelTa, we only reason about entities and relations, independently from specific metamodel types and relations. Entities are represented using a UML class notation and their metamodel appears on the top right. An "x" symbol on an element inside a rule means that this element should not appear in the concrete transformation rule implementing the DelTa rule.

Color coding of entities and relations inside the rules indicates whether they are part of the constraint or a type of action of the rule. White elements form the minimal application pre-condition that a concrete transformation rule implementing it should have. Gray elements are the minimal elements to be created in the concrete transformation rule. For example, the `tEnt` and the trace between it and `sEnt` must be created. Therefore, the rule *entityMapping* dictates that the concrete transformation rule implementing it should look for an entity from one metamodel and create a new entity from another metamodel, as well as a trace between them. Elements in black are the minimal elements to be deleted in the concrete transformation rule.

When a self loop symbol appears on the top left (as it is the case with both rules in Fig. 2), the DelTa rule is exhaustive: the concrete transformation rule implementing it should be applied on all of its matches. This may require to have more than one rule implementing this DelTa rule, for example to match different metamodel types.

In DelTa, the scheduling is depicted using a control flow notation. The start node (filled ball) indicates the initial rule of the design pattern. Arrows between rule blocks indicate a predence order: the concrete transformation rule

implementing the *entityMapping* rule should be performed before the one implementing the *relationMapping* rule. A dashed box containing rules specifies that the order of execution of the rules it contains is irrelevant to the design pattern. Entities, rules, and scheduling represent the *participants* of a model transfromation design pattern. In this paper, we use model transformation design patterns expressed in DelTa from [10].

### 2.4 Design Patterns Detection in Software Engineering

To the best of our knowledge, there is no previous work that tackles the detection of design patterns in model transformation. Most of the detection approaches target the patterns of Gamma et al. in object-oriented programs [13]. These approaches target primarily the structural patterns as these can be detected by matching the structure of code to one of the pattern [3,22]. To improve the detection, some projects combine multiple strategies as in [7]. The detection of behavioral patterns also attracted the interest of the research community. In De Lucia et al. [2], the authors use model checking to improve the detection of behavioral patterns. A work similar to our is one in [5]. In this paper, the authors first identify pattern key participants using a machine learning technique. Then, they check for the other participants of the pattern and the relations between them.

## 3 Design Pattern Detection for Model Transformation

We propose an approach to detect complete and partial instances of design patterns in concrete model transformations. We consider design pattern detection as a constraint satisfaction problem where a design pattern imposes a specific structure that a concrete model transformation should contain, and we solve it using a declarative strategy based on an inference rule engine.

### 3.1 Overview

As shown in Fig. 3, the detection of a design pattern is encoded as a set of *rules*. These rules apply to a set of *facts* representing the model transformation. The facts conform to *fact templates*: a generic abstract representation of transformation components relevant to design pattern detection. This abstract representation makes our approach independent from a specific model transformation language. The mapping to of a concrete model transformation is performed by a model-to-text transformation.

The detection process is performed in three automated steps. First, the transformation is mapped to an abstract representation (i.e., facts) using a higher-order transformation. Second, we identify which rules of the model transformation can play the role of the participants of the design pattern. Third, once the participant candidates are identified, we verify that their execution satisfies the scheduling scheme specified in the design pattern.

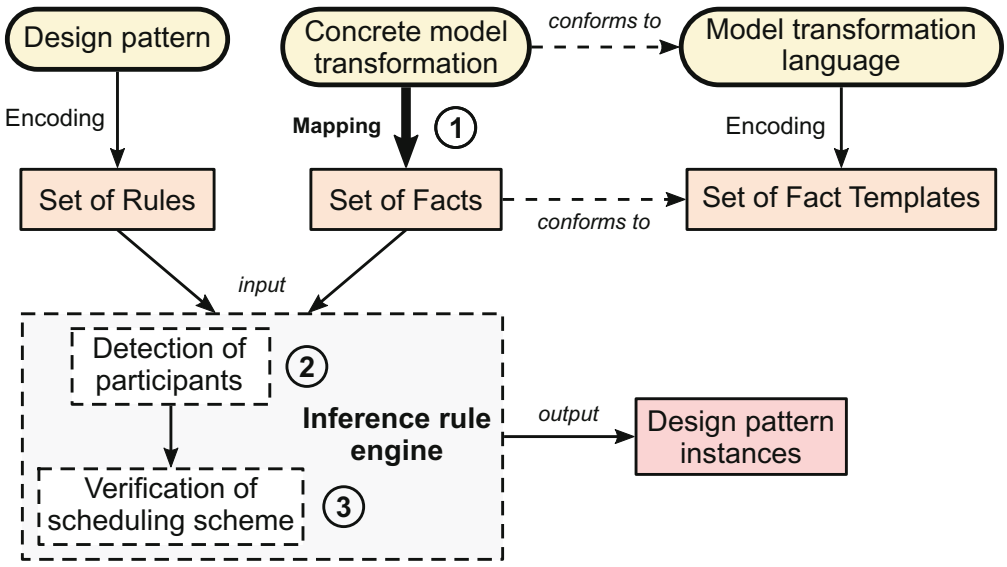



**Fig. 3.** Architecture overview of design pattern detection

In the remainder of this section, we describe how concrete model transformations are mapped to generic facts and then explain the two steps of the detection process.

### 3.2 Mapping Model Transformations to Generic Facts

**Fact Template.** To describe model transformations, we defined a fact-based language inspired by the Henshin transformation language [1]. The motivation behind this decision is that design patterns, as defined in [17], deal mainly with the manipulation (creation/modification/deletion) of model elements by rules as well as with the rule execution scheduling. All these constructs can be described by the Henshin concepts.

The main fact template to describe a transformation is `Rule`. A `Rule` is composed of nodes, each corresponding to an action on a model element present in the pre- or post-condition of a model transformation rule. Nodes are described by the fact template `Node`. Nodes have several attributes to define the element name and type they represent, a reference to the rule in which they appear, and also an action. If the action slot is assigned "create", "update" or "delete", then the node is part of the post-condition of the transformation rule. If it is assigned "preserve" or "forbid", then the node is part of the pre-condition (positive or negative constraint, respectively) of the transformation rule. Nodes may also be related with the `Edge` fact template when the action in one node depends on another node, e.g., an element is created and its attributes are set according to those of another element. For rule execution scheduling, we define the fact template `Sequence` that specifies the precedence between two rules.

The precedence relationship may also involve control events such as the beginning and the end of a loop.

Listing 1.1 shows fact templates for `Rule`, `Node`, and `Sequence` expressed in the Jess language [6]. In Jess, each template has a name and a set of slot definitions. When asserting a fact, the slots must be set with values. Some slots are used to describe the fact properties such as Name in Rule and Action in Node. Others are used to connect facts. For example, the slot `RuleId` in Node is set with the `Id` of the `Rule` which the node belongs to. Similarly, `SourceId` and `TargetId` in Sequence refer respectively to the `Id`s of the preceding and following rules.

**Listing 1.1.** Fact Templates representing a model transformation language

```
(deftemplate Rule (slot Id)(slot Name))

(deftemplate Node (slot Id)(slot RuleId)(slot Action)
(slot Occurrences)(slot Name)(slot Type))

(deftemplate Sequence(slot SourceId)(slot TargetId))
```

**Fact.** Listing 1.2 shows the Jess facts of a rule having two nodes.

**Listing 1.2.** Fact representing a concrete model transformation

```
(Rule (Id"R1")(Name"Class2TableMapping"))

(Node (Id"N1")(RuleId"R1")
(Action"preserve")(Occurrences"n")(Name"")(Type"Class"))

(Node (Id"N2")(RuleId"R1")
(Action"create")(Occurrences"n")(Name"")(Type"Table"))
```

To be effective for large transformations, we automate the mapping of a given concrete model transformation to a set of facts. Therefore, we need to write a fact generator for each model transformation language considered. To this end, we use Acceleo[1], a template-based model-to-text transformation tool in EMF. These code generation templates encode the semantic equivalence between the transformation language constructs and our fact templates. Listing 1.3 illustrates an example for generating of a fact `Rule` from a Henshin rule. Although our implementation currently supports Henshin, adapting to another model transformation language simply requires to create a new Acceleo template for it.

**Listing 1.3.** Fact representing a concrete model transformation

```
[template public generateRule(rule:Rule, position:Integer)]
    (Rule (Id \"["R" + position]\") (Name \"[rule.name/]\"))
[/template]
```

[1] https://eclipse.org/acceleo/.

### 3.3 Encoding Design Patterns as Detection Rules

As mentioned in Sect. 2.2, the participants of a model transformation design pattern are DelTa rules, the elements they contain in their constraint and actions, and their scheduling scheme. The pattern *Entities Before Relations* in Fig. 2, for instance, consists of two DelTa rules: *entityMapping* and *relationMapping*. It also mandates that the former must be executed before the latter. Consequently, our detection strategy starts by finding concrete model transformation rules that match the ones in the DelTa model, and then verify if the scheduling specified in the patterns holds for the concrete matched rules.

**Listing 1.4.** Rule encoding the complete entityMapping rule of Entities before Relations design pattern

```
(defrule CreateEntityMapping_Rule
  (Rule (Id ?r_1)(Name ?r_2))
  (Node (Id ?sEnt_1)(RuleId ?r_1)(Action"preserve")
    (Occurrence ?sEnt_4)(Name ?sEnt_5)(Type ?sEnt_6))
  (Node (Id ?tEnt_1)(RuleId ?r_1)(Action"forbid")
    (Occurrence ?tEnt_4)(Name ?tEnt_5)(Type ?tEnt_6))
  (Node (Id ?tEnt_2)(RuleId ?r_1)(Action"create")
    (Occurrence ?tEnt_4)(Name ?tEnt_5)(Type ?tEnt_6))
  (Edge (Id ?ed_1)(RuleId ?r_1)(SourceId ?sEnt_1)
    (TargetId ?tEnt_1))
  (Edge (Id ?ed_2)(RuleId ?r_1)(SourceId ?sEnt_1)
    (TargetId ?tEnt_2))
=>
  (assert
   (EbR_entityMapping
    (Id (str-cat ?r_1 ?sEnt_1 ?tEnt_1 ?tEnt_2 ?Ed_1 ?Ed_2))
    (RuleId ?r_1)
    (sEnt_1Id ?sEnt_1)
    (tEnt_1Id ?tEnt_1)
    (tEd_1Id ?ed_1)
    (tEnt_2Id ?tEnt_2)
    (tEd_2Id ?ed_2)
    (accuracy 1))
  )
)
```

The detection of instances of a DelTa rule is encoded as a *rule* in Jess. For example, Listing 1.4 rule detects complete instances of *entityMapping*. The Jess rule first filters all transformation rule facts that have a "preserve" node connected to a "forbid" node and to a "create" node. For each rule satisfying these conditions, it asserts a fact *EbR_entityMapping*. Another Jess rule will filter the concrete rules that can play the role of *relationMapping* and asserts for each match a fact *EbR_relationMapping*. The encoding of DelTa rules into Jess rules can be implemented with Acceleo templates.

Once the potential participants are detected, the next step is to ensure if the execution schedule of the concrete rules corresponds to the one of the pattern. In the case of the pattern *Entities Before Relations*, a Jess rule filters facts

*EbR_entityMapping* and *EbR_relationMapping*, and a `Sequence` fact relating the rules respectively involved in the participant facts.

### 3.4 Accuracy for Complete and Partial Instances

In the case of complete instance detection, all the conditions (participants and scheduling) should be fully satisfied, i.e., accuracy equals 1.

When detecting partial instances, rules variants are defined for participants and scheduling detection. These rules may omit one of the conditions and adjust the value of fact accuracy accordingly. For example, in the detection of *entityMapping* participants, a variant rule can consider rules with "preserve" and "create" nodes, but without a "forbid" node. This is depicted in Listing 1.5. The accuracy is then adjusted to 0.66 for example. The scheduling verification rule, calculate the global accuracy of the pattern instance from the accuracy values of the participants facts and one of the scheduling itself.

**Listing 1.5.** Rule encoding a partial entityMapping rule of Entities before Relations design pattern

```
(defrule CreateEntityMapping_Rule
  (Rule (Id ?r_1)(Name ?r_2))
  (Node (Id ?sEnt_1)(RuleId ?r_1)(Action"preserve")
    (Occurrence ?sEnt_4)(Name ?sEnt_5)(Type ?sEnt_6))
  (not (Node (Id ?tEnt_1)(RuleId ?r_1)(Action"forbid")
    (Occurrence ?tEnt_4)(Name ?tEnt_5)(Type ?tEnt_6))
  ...
=>
  (assert
   (EbR_entityMapping
    (Id (str-cat ?r_1 ?sEnt_1 ?tEnt_1 ?tEnt_2 ?Ed_1 ?Ed_2))
    (RuleId ?r_1)
    (sEnt_1Id ?sEnt_1)
    (tEnt_1Id"")
    (tEd_1Id"")
    (tEnt_2Id ?tEnt_2)
    (tEd_2Id ?ed_2)
    (accuracy 0.66))
  )
)
```

## 4 Preliminary Evaluation

### 4.1 Setup

A preliminary evaluation of this work consists in selecting a subset of design patterns and detect their instances on a sample of model transformations. The goal here is to analyze qualitatively how our detection approach applies to concrete transformations.

We selected 13 Henshin transformations[2] with different characteristics (see Table 1). As we had to analyze manually the results, we opted for small-medium transformations having 1 to 13 rules. We also paid attention to the control complexity as most of the transformation design patterns deal with the rule execution control. Indeed, some of the selected transformations use default implicit control (no control specified), and others have up to 13 rule scheduling units with loops and calls between the units. Additionally, we varied the complexity of the rules with respect to the number of involved model elements, with an average number of nodes per rule between 3 and 11.

**Table 1.** Selected transformations.

| Model transformations | # rules | # sch-unit | # nodes | # relations | # calls | # loop |
|---|---|---|---|---|---|---|
| bank | 3 | 0 | 12 | 12 | 0 | 0 |
| bankmap | 1 | 0 | 5 | 4 | 0 | 0 |
| comb | 2 | 1 | 22 | 38 | 1 | 1 |
| diningphils | 4 | 0 | 22 | 34 | 0 | 0 |
| ecore2genmodel | 8 | 6 | 55 | 59 | 12 | 2 |
| gossipingGirls | 2 | 0 | 7 | 9 | 0 | 0 |
| grid-full | 4 | 5 | 18 | 27 | 8 | 3 |
| grid-sparse | 3 | 4 | 11 | 16 | 6 | 2 |
| java2statemachine | 13 | 13 | 77 | 59 | 27 | 5 |
| petriM | 2 | 0 | 15 | 27 | 0 | 0 |
| sierpinski | 1 | 0 | 6 | 12 | 0 | 0 |
| sort | 1 | 1 | 3 | 2 | 1 | 1 |
| entityRelationMapping | 3 | 1 | 16 | 14 | 3 | 0 |

In this preliminary evaluation we experimented with the detection of three patterns, selected from the catalog of [17]. Two of them deal with the rule modularization (*Entities Before Relations* and *Construction and cleanup*), and one with optimization (*Unique Instantiation*).

**Entities Before Relations.** The goal of this pattern (Fig. 2), also called *Map Objects Before Links*, is to create the entities and then their relations. As mentioned in Sect. 3.3, three rules are defined for the detection of this pattern: (1) detection of entities creation, (2) detection of relations creation, and (3) precedence checking between the two creations. In addition to the detection of complete instances, we implemented the detection of one kind of partial instance, i.e., the situation in which the transformation program have rules for creating the entities before the creation of their relations, but does not check if an entity exists before it creates a new one (see Sect. 3.4).

[2] https://www.eclipse.org/henshin/examples.php.

**Construction and Cleanup.** As shown in Fig. 4, this pattern consists in separating rules which create model elements from those which delete elements [17]. Like for the previous patterns, the detection is done in three phases: (1) finding element creation rules, (2) finding element deletion rules, and (3) precedence checking between the two.

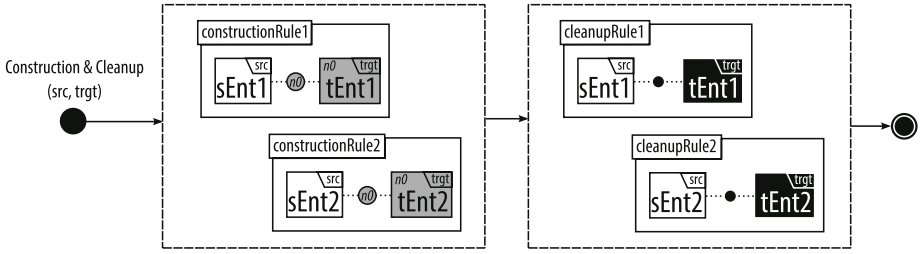



**Fig. 4.** Construction & cleanup - Structure in DelTa

**Unique Instantiation.** This pattern, sketched in Fig. 5, aims at avoiding multiple creations of the same model element. This may happen in two situations: (1) two rules creating the same model element or (2) a rule creating a model element, and that appears in a loop inside a rule execution schedule. We defined detection rules for each situation, i.e., identifying element-creation rules, and checking duplications and loops.

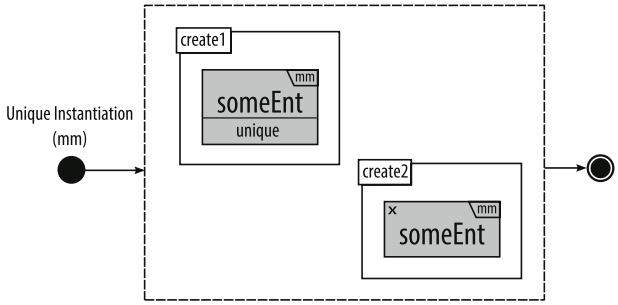



**Fig. 5.** Unique instantiation - structure in DelTa

### 4.2 Qualitative Analysis

**Entities Before Relations.** Surprisingly, our prototype did not find complete instances of the pattern *Entities Before Relations*. To understand this, we manually inspected the automatically detected partial instances. We noticed that, in many cases, the *EntityMapping* participants were identified with an accuracy of 1. However, the *relationMapping* participants did not satisfy the condition of the non-existence of a relation before its creation. All the detected partial instances satisfied the execution schedule conditions with perfect accuracy. Figure 1 illustrates two examples of partial instances found in the *entityRelationMapping* e rules transformation. The rules *ClassMapping* and *AttributeMapping*

are both complete instances of *entityMapping*. Conversely, in rule *attributeRelationMapping*, the relation between "Class" and "Attribute" is mapped to a relation between "Table" and "Column" without ensuring that such a relation does not already exist (not a "forbid" action). Although the scheduling is perfectly accurate, i.e., both *ClassMapping* and *AttributeMapping* rules precede *attributeRelationMapping*, the aggregated accuracy is lower than 1.

**Construction and Cleanup.** The prototype found many instances of the design pattern *Construction and cleanup*. An interesting instance is one found in the *Java2StateMachine*. In this transformation, only one rule has a "delete" action (*updateAction* on the right of Fig. 6). All the other rules create elements. This rule appears at the last step of the execution schedule (on the left of Fig. 6). This is a non trivial instance to detect because of the modularization of the execution schedule. In our detection program, we implemented a function that reconstructs a flat schedule by resolving the schedule step references.

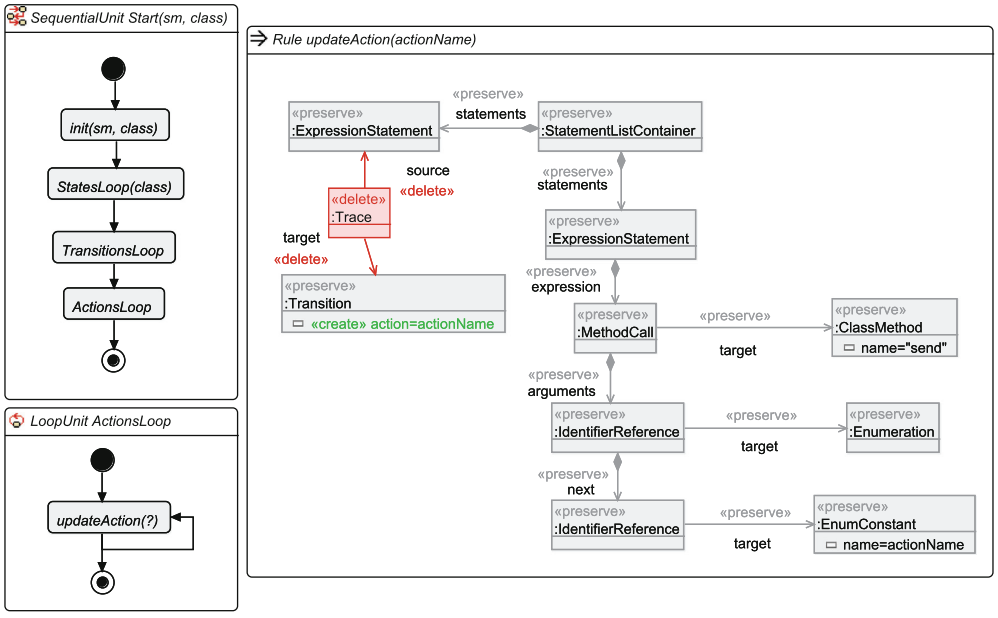
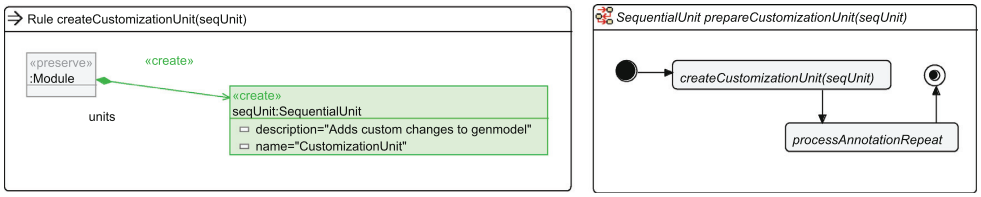



**Fig. 6.** An example of instance of the pattern *Const*ruction and Cleanup

**Unique Instantiation.** This is by far the most frequent pattern and many of its instances were found in almost all the considered transformations. Some of them have a high accuracy. An example of a complete instance was found in the *Ecore2GenModel* high-order transformation. The *createCustomizationUnit* rule creates an element, which is not created by other rules (Fig. 7a). Moreover, this rule does not appear in a loop in the execution schedule (Fig. 7b).




(a) Creation rule (b) Cleaning instance

**Fig. 7.** Unique instantiation instance detected in Ecore2Genmode transformation

## 5 Conclusion

In this paper, we propose an approach and a preliminary implementation for the detection of complete and partial instances of design patterns in model transformations. Our approach follows a declarative strategy which consists in identifying transformation rules that play the roles of design pattern participants and then check if their execution sequence conforms to the schedule specified in the pattern.

We conducted a preliminary evaluation which consisted in applying our detection rules on a set of transformations and in qualitatively analyzing the detection results. Although the obtained results are encouraging, our evaluation revealed some limitations. First, we define explicitly rules for detecting pattern variants [20]. The advantage of this strategy is that we identify acceptable variants of a design pattern. The drawback is that our detection code is very verbose with very similar rules. We plan in the future to have a generic detection of variants by allowing weights to the pattern participants.

Another limitation of our approach resides in the limited number of control structures we handle. In our current implementation, we do not consider alternatives structures. Thus for the pattern *Unique Instantiation*, if two rules respectively in the two branches of the alternative create the same element, we do not detect a valid instance. Handling more control structures is a part of our future work.

## References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: Model Driven Engineering Languages and Systems, pp. 121–135 (2010)
2. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Improving behavioral design pattern detection through model checking. In: European Conference on Software Maintenance and Reengineering, pp. 176–185 (2010)
3. Dong, J., Zhao, Y., Peng, T.: A review of design pattern mining techniques. Int. J. Softw. Eng. Knowl. Eng. **19**(06), 823–855 (2009)
4. Guéhéneuc, Y.G., Guyomarc'h, J.Y., Sahraoui, H.: Improving design-pattern identification: a new approach and an exploratory study. Softw. Qual. J. **18**(1), 145–174 (2010)

5. Gueheneuc, Y.G., Sahraoui, H., Zaidi, F.: Fingerprinting design patterns. In: Working Conference on Reverse Engineering, pp. 172–181. IEEE (2004)
6. Hill, E.F.: Jess in Action: Java Rule-Based Systems. Manning Greenwich, Greenwich (2003)
7. Rasool, G., Mäder, P.: Flexible design pattern detection based on feature types. In: International Conference on Automated Software Engineering, pp. 243–252 (2011)
8. Agrawal, A.: Reusable idioms and patterns in graph transformation languages. In: International Workshop on Graph-Based Tools, ENTCS, vol. 127, pp. 181–192. Elsevier (2005)
9. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. Spec. Issue Model-Driven Softw. Dev. **45**(3), 621–645 (2006)
10. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Comput. Lang. Syst. Struct. **46**, 106–139 (2016). doi:10.1016/j.cl.2016.07.004
11. Ergin, H., Syriani, E.: Identification and application of a model transformation design pattern. In: ACM Southeast Conference, ACMSE 2013. ACM (2013)
12. Ergin, Hüseyin, Syriani, Eugene: Towards a Language for Graph-Based Model Transformation Design Patterns. In: Ruscio, Davide, Varró, Dániel (eds.) ICMT 2014. LNCS, vol. 8568, pp. 91–105. Springer, Heidelberg (2014). doi:10.1007/978-3-319-08789-4_7
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional, Boston (1994)
14. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: International Conference on Software engineering, pp. 471–480. ACM (2011)
15. Iacob, M.E., Steen, M.W.A., Heerink, L.: Reusable model transformation patterns. In: Enterprise Distributed Object Computing Conference Workshops, pp. 1–10. IEEE Computer Society (2008)
16. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G.M., Syriani, E., Wimmer, M.: Model transformation intents and their properties. Softw. Syst. Model. **15**(3), 647–684 (2014)
17. Lano, K., Rahimi, S.K.: Model-transformation design patterns. IEEE Trans. Softw. Eng. **40**(12), 1224–1259 (2014)
18. Lano, K., Rahimi, S.K., Poernomo, I.: Comparative evaluation of model transformation specification approaches. Int. J. Softw. Inf. **6**(2), 233–269 (2012)
19. Levendovszky, T., Lengyel, L., Mészáros, T.: Supporting domain-specific model patterns with metamodeling. Softw. Syst. Model. **8**(4), 501–520 (2009)
20. Prechelt, L., Krämer, C.: Functionality versus practicality: employing existing tools for recovering structural design patterns. J. Univ. Comput. Sci. **4**(11), 866–882 (1998)
21. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. Softw. Syst. Model. **12**(2), 387–414 (2013)
22. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.: Design pattern detection using similarity scoring. Trans. Softw. Eng. **32**(11), 896–909 (2006)

# Chapter 5

## A Generic Approach to Detect Design Patterns in Model Transformations Using a String-Matching Algorithm [1]

Chihab eddine Mokaddem,
Houari Sahraoui and Eugene Syriani
DIRO, Université de Montréal

This paper represents the work I did for the second contribution of my thesis.
The co-authors are my supervisors who contributed to the writing.

[1]The content of this chapter has been submitted to the International Journal on Software and Systems Modeling (SoSyM). Reference (Mokaddem et al, 2021)

**Résumé.** La maintenance des artefacts logiciels fait partie des tâches les plus difficiles auxquelles un ingénieur est confronté. Comme tout autre morceau de code, les transformations de modèles développées par les ingénieurs sont également soumises à une maintenance. Pour faciliter la compréhension des programmes, les ingénieurs logiciels s'appuient sur de nombreuses techniques, telles que la détection des patrons de conception. Par conséquent, la détection des patrons de conception dans les implémentations de transformation de modèles est d'une grande valeur pour les développeurs. Dans cet article, nous proposons une technique générique pour détecter automatiquement les patrons de conception et leurs variations dans les implémentations de transformation de modèles. Il faut en entrée un ensemble de règles de transformation de modèles et les participants d'un patron de conception de transformation de modèles pour trouver les occurrences de ce dernier dans le premier. La technique détecte également certains types de formes dégénérées du patron, indiquant ainsi les opportunités potentielles d'améliorer l'implémentation de la transformation de modèles.

# A Generic Approach to Detect Design Patterns in Model Transformations Using a String-Matching Algorithm

**Chihab eddine Mokaddem · Houari Sahraoui · Eugene Syriani**

Received: date / Accepted: date

**Abstract** Maintaining software artifacts is a complex and time-consuming task. Like any other program, model transformations are subject to maintenance. In a maintenance process, much effort is dedicated to the comprehension of programs. To this end, several techniques are used, such as feature location and design pattern detection. In the particular case of model transformations, detecting design patterns contributes to a better comprehension as they carry valuable information on the transformation structure. In this paper, we propose a generic approach to detect, semi-automatically, design patterns and their variations in model transformations. Our approach encodes both design patterns and transformations as strings and use a string-matching algorithm for the detection.

**Keywords** Design pattern · Model transformation · Pattern detection · String matching · Bit-vector · Model-driven engineering

## 1 Introduction

Model transformation is now the mainstream paradigm to manipulate models in model-driven software engineering (MDE) [6]. Designing model transformations is a tedious task. Moreover, like any other code artifact, model transformations evolve and should be maintained. To assist developers in writing and maintaining model transformations, several design patterns have been proposed [11, 22]. In general, design patterns facilitate the comprehension and manipulation of software programs [1]. In the special case of model transformations, they help improving the quality of model transformation specifications and designs, as stated by Lano et al. [22].

Detecting instances of a pattern in a transformation provides valuable information to the developer, such as understanding high-level concepts used, and iden-

C. Mokaddem (✉) · H. Sahraoui · E. Syriani
Université de Montréal, Montréal, Canada
E-mail: {cemo.mokaddem,houari.sahraoui,eugene.syriani}@umontreal.ca

tifying refactoring and reuse opportunities. However, as for general programs, developers do not always implement perfectly a pattern in model transformations. Hence, design pattern detection should identify both complete and incomplete occurrences. Detecting various forms of a design pattern, including incomplete forms, offers refactoring opportunities to improve transformations by completing a form or by replacing one form by a more appropriate one.

Detecting design patterns in model transformations did not get much attention so far from the modeling community. To the best of our knowledge, only our previous work in [28] has attempted to automatically detect design patterns in model transformations using manually-written detection rules. Preliminary results showed that this is an effective technique to find complete and approximate design pattern occurrences. However, this technique has performance limitations as it relies on a rule inference engine that is time and memory consuming. Another limitation of this technique is the need to specify a set of detection rules for each pattern.

To find inspiration on how to detect patterns in transformations, we looked at the active community of design pattern detection in object-oriented programs. As reported in [1], there are dozens of detection approaches for this family of programs. However, as mentioned in [13], these approaches also suffer from performance problems, because detecting complete and incomplete occurrences is generally costly in time, due to the large search-space that includes all possible combinations of classes. These approaches are also prone to return many false positives, impeding program comprehension, and cluttering the maintainers' cognitive capabilities. To address the performance issues, the work by Kaczor et al. [20] uses a string matching technique inspired by pattern matching algorithms in bioinformatics to identify pattern occurrences in object-oriented programs. These algorithms allow to efficiently process a large amount of data if the problem to solve can be encoded as a string matching one.

In this paper, we propose a generic technique to detect design pattern occurrences in declarative model transformation implementations, without writing detection code for each design pattern, its variants, and approximations. Like in Kaczor et al., we rely on a bit-vector algorithm that has proven to be efficient for string matching problems [30]. The challenge we faced is how to encode model transformations, which are sets of rules linked by control schemes, as strings. The same challenge arises also in the encoding of the patterns as strings. We succeed to encode the participants of a patterns as strings, but had to complete our approach by a manual step to combine the identified participant instances to form pattern occurrences. Thus, the detection consists in an automated step that matches the participant strings of a pattern with rule strings of transformation, and a manual step to complete the occurrences. In addition to the performance, an advantage of using this approach is the fact both complete and incomplete occurrences can be detected.

We evaluated our approach for the detection of various forms of 10 design patterns in 18 transformations, collected from open source repositories. We first manually checked to which extent design patterns are used in these transformations. Then, we evaluated the ability of our approach to detect the various instances of the patterns. We also studied the co-occurrences of the different patterns. Our results show that patterns are effectively used in transformations and that our approach is able to detect them, in less time than the rule-based approach. Moreover,

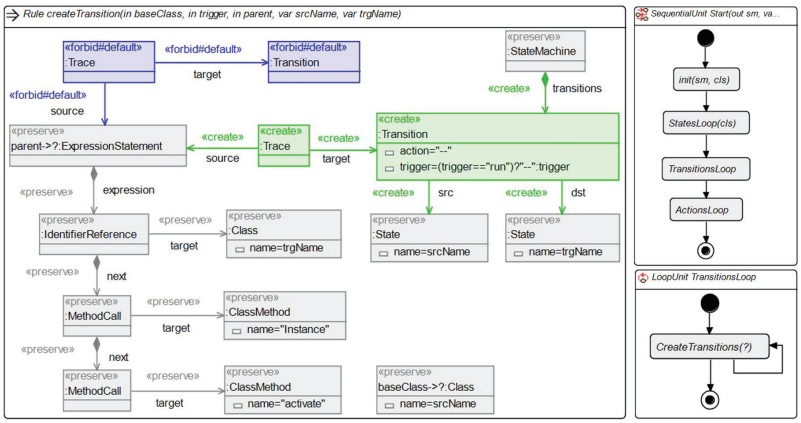



Fig. 1: Excerpt of the `Java2StateMachine` transformation implemented in Henshin, showing the `createTransition` rule

we found that patterns are not always used independently, but in combination with other patterns.

The rest of the paper is structured as follows. In Section 2, we first introduce the basic notions used in our work, and then, discuss the related work. Section 3 details the different steps of our approach, whereas Section 4 describes the different forms of patterns that can be detected by our approach. We provide an evaluation of the detection approach in Section 5. Finally, we discuss the limits of our approach in Section 6, and conclude in Section 7.

## 2 Background and Related Work

In this section, we briefly give some background on model transformations and corresponding design patterns and illustrate these concepts with a running example. This example also allows us to highlight the challenges of design pattern detection in model transformations. The rest of the section is dedicated to the discussion of the related work on design pattern detection.

### 2.1 Model Transformation: an Illustrative Example

To help understanding the main concepts involved in our research, we use as example the exogenous transformation `Java2StateMachine`, implemented in Henshin [5]. This transformation was written for the Transformation Tool Contest of 2011 [19]. An excerpt of this transformation is depicted in Fig. 1.

We are specifically interested in rule-based model transformations. This kind of transformation is typically defined with a set of declarative rules to be executed. A rule consists of a pre-condition and a post-condition pattern. The pre-condition

pattern determines the applicability of a rule and contains two types of conditions, positive and negative. The positive application condition (PAC) represents the pattern that must be found in the input model to apply the rule. Optionally, negative application conditions (NAC) may be specified to inhibit the application of the rule if these conditions are verified. The rule `createTransition` in Fig. 1-left illustrates the elements involved in a transformation in Henshin. A rule is represented as a graph representing both the pre and post-conditions. Gray elements, annotated with the *"preserve"* stereotype, indicate the PAC pattern whereas the blue elements, annotated with the *"forbid"* stereotype, form the NAC conditions. Henshin supports multiple NAC groups. In this rule, the `Trace` and `Transition` objects together with their adjacent associations are part of the same NAC group identified as *#default*. The post-condition imposes the pattern to be found after the rule is applied. In our rule, the green elements, annotated with the *"create"* stereotype, must be created by the rule. It is also possible to delete elements (represented by the color red and the stereotype *"delete"* (not present in our example). In summary, patterns are made up of structural elements (i.e., model fragments) and of constraints or actions on them. Therefore, roughly speaking, the rule `createTransition` states that if a method call exists between a source class and a target class, create a transition between the states corresponding to these classes if a transition was not created already for this method call. To check for the pre-existence of the transition (NAC), the transformation uses a traceability auxiliary metamodel to keep a trace of all created transitions (as highlighted by the `Trace` node). Rules in Henshin can be parameterized to bind elements across rules or pass values. This rule has three input (`in`) parameters. `baseClass` is bound to a class type that must be preserved, `parent` is bound to an expression statement type that must be preserved and `trigger` is a string value. The latter is used in attribute constraints or assignments. For example, if the value of the `trigger` parameter is the string `"run"`, then the trigger value of the transition is `--`, otherwise it will be assigned the parameter value. A rule may also declare local variables, like `srcName` and `trgName` to refer to objects within the rule. For example, the source of the created transition must be the state whose name is the same as the base class name.

The rule `createTransition` is fired according to a scheduling scheme that defines the order in which the rules are applied when a transformation is executed. In Henshin, the scheduling can be specified explicitly through one or more control flow structures to partially order the rule executions. In our example, a first control structure specifies that state rules must be executed before transition rules (Fig. 1-top-right. The second control structure, in the bottom-right of the figure, states that the transition rule must be fired iteratively until all the transitions are created. As it can be seen in our approach, the detection of design patterns is done semi-automatically: (1) automated detection of patterns in the rules and (2) manual verification of the compliance with the control scheme and attribute constraints.

### 2.2 Design Patterns for Model Transformations

A model transformation design pattern expresses a means of solving a common model transformation design problem [22]. It describes the transformation structure (rules, condition patterns, and scheduling) that constitute the solution idea.

A design pattern includes also a description of the problem which motivated the pattern, how such problems can be detected, and the benefits and negative consequences to consider when using the pattern.

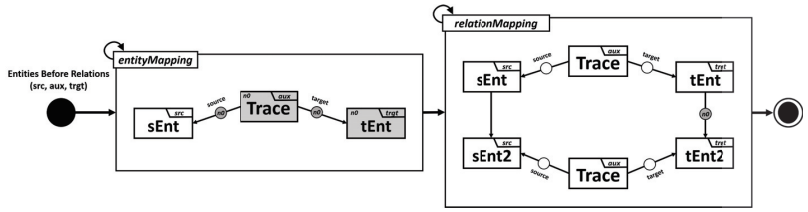



Fig. 2: The *Entity-before-Relation* design pattern with explicit trace elements

The idea of proposing design patterns for model transformation gained popularity by the late 2000s. Small sets of patterns were proposed by different research teams such as ones in [16] and [17]. Later in [22], Lano et al. presented a larger catalog of 24 design and specification patterns. To allow the representation of patterns in a way that ease their usage in development processes, Ergin et al. [11] defined a domain-specific language, DelTa. DelTa represents design pattern solutions in a platform-independent model: independent from the model transformation language. Since we are interested in detecting design patterns in concrete model transformations, we must transform the design pattern into a format that will correspond to its implementation in a specific model transformation language. Fig. 2 shows an example pattern from the catalog in [22] that was described in DelTa, and adapted for Henshin. For example, in this representation specific to Henshin, we explicitly represent a trace object with source and target links. In DelTa, this is originally represented by a trace link, which is not possible to define in Henshin. This pattern states that, when transforming a model into another model, rules mapping entities should be executed before ones mapping relations.

In DelTa, color coding is used to indicate the elements to be preserved (in white), elements to be created (in gray), and elements to be deleted (in black) in a transformation. A model transformation design pattern consists of participants and their scheduling. **Participants** represent rule templates that shall be implemented in a concrete model transformation. In the pattern *Entities before Relations* (E-R), there are two participants: *entityMapping* and *relationMapping*. In a participant, entities and relations play a **role**. *entityMapping* indicates that an entity `sEnt` has to be mapped if an entity `tEnt` was not previously created from it. Negative application conditions (NAC) can exist in any participant. It indicates the pattern that should not be found by the concrete transformation rule implementing it. Elements considered by the NAC are labeled by *n0* in DelTa. The second participant *relationMapping* states that if two source entities `sEnt` and `sEnt2` were mapped into two target entities `tEnt` and `tEnt2`, a relation between `sEnt` and `sEnt2` can be mapped into a relation between `tEnt` and `tEnt2`, if this was not done before. Note that both rule templates use the auxiliary type to keep track of the previous mapping by means of the `Trace` element. As it can be seen in Fig. 2, DelTa also allows to express the rule **scheduling** scheme in the form of edges

between the rule templates. For the E-R pattern, the *entityMapping* participant must preceed the *relationMapping* participant. We refer the reader to [11] for the complete description of design patterns in DelTa.

### 2.3 Design Pattern Detection

Up to date, design patterns were primarily used for model transformation writing [11, 24]. To the best of our knowledge, few research contributions targeted their detection. In a previous work, we proposed an approach to detect patterns in transformations [28]. Design patterns are encoded into rules and transformations into facts on which the rules are applied. Although the detection results were very encouraging, this approach is not generic enough to be applicable to any pattern. Indeed, while the mapping of transformations into fact sets is automated, the detection rules for each pattern have to be written manually. The detection of design pattern was also mentioned in [24]. In this study, the authors defined detection criteria for each pattern to be applicable manually by the authors to check which patterns are used in transformations. The criteria are given in natural language at a high-level of abstraction.

Outside the model transformation community, there is a large body of work on design pattern detection, especially in object-oriented programs [1]. The detection strategies differ in many ways. One difference is the intermediate representation of the code used to perform the detection. The most-used representation is the Abstract Syntax Tree (AST) [25,40,47] or the Abstract Syntax Graph (ASG) [12, 32]. Some strategies map the code into high-level graph representation such as UML [36,46]. Other representations include, among others, matrices [10,44], Prolog assertions [21,34], and strings [20].

Another difference between the strategies is the detection technique. Query-based detection with mainly SQL is used, for example, in [4,38]. The authors use SQL queries to represent and detect the design patterns. This methodology is limited to the structural patterns. Some techniques use a combination of metrics and structural relations to find patterns [9,45]. Another alternative is to use ontologies as in [2,27].

One of the most important limitations of the above-mentioned techniques is the lack performance when the input programs are very large. Attempts have been made in [3] and [14] to improve the detection efficiency. Their idea is to reduce the search space by removing entities that obviously do not participate in an occurrence of design pattern according to expected metrics values. The detection performance issues are experienced in many domains. For example, in bioinformatics, pattern detection is also used to locate different genes in long DNA sequences. To this end, powerful solutions were adopted, such as vectorial algorithms [7,30], automata simulation [15], and dynamic programming alignment [31,41]. Yet, these solutions cannot be applied directly to our problem since we are dealing with complex structures and not strings. Inspired by these algorithms, Kaczor et al. [20] chose to represent object-oriented programs, as well as design patterns, as strings. This allowed the use of string-matching algorithms to detect pattern occurrences with a good accuracy.

Finally, an important property of a detection strategy is the genericity with respect to the patterns to detect. Most of the existing work requires to write a

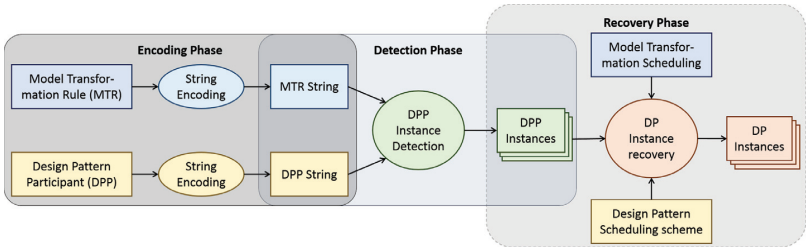



Fig. 3: Overall process of the approach

specific code for each pattern, being queries [38], structural rules [47], or pattern-matching rules [26]. There are few strategies that provide the description of the patterns as an input of the detection algorithm such as in [20, 44].

Overall, the aim of the approach proposed in this paper is to detect design patterns in model transformations efficiently and without the need of writing specific detection code for each pattern. To address time and memory consumption issues, we rely on a string-matching algorithm proven to be efficient for large genomic databases. Additionally, the fact that the patterns to detect are specified as strings to match in these algorithms eliminates the need to write new code for each pattern.

## 3 Design pattern detection by string matching

String matching techniques and algorithms are popular in bioinformatics, given they are highly scalable in terms of input size and performance [30, 35]. Kaczor et al. [20] showed the benefits of this approach by applying them to detect design patterns in source code. In this paper, we adapt this technique to detect design patterns in model transformation.

Fig. 3 shows an overview of our approach separated into three phases. A key aspect of the string matching technique is to represent input artifacts as string sequences. Given a model transformation implementation and a set of design patterns, our goal is to find instances of the design patterns in the transformation. To automate the process, we consider a rule of the transformation (e.g., `createTransition` in Fig. 1) and a participant of a given design pattern (e.g., `entityMapping` in Fig. 2) as inputs. The first phase encodes both inputs as strings following an identical encoding strategy. The choice of string representation will have a direct impact on the performance of the detection algorithm and is, therefore, a crucial step.

The second phase detects the occurrence of the participant string in the rule string using a string matching algorithm. A naive detection algorithm would traverse all elements in both structures. Each element in one would be compared with all elements in the other to find the matching element structurally and the matching properties. To resolve this combinatorial problem, previous approaches using a bit-vector algorithm (like in [20]) have shown to be an efficient solution by bounding the number of vector operations independently from the length of

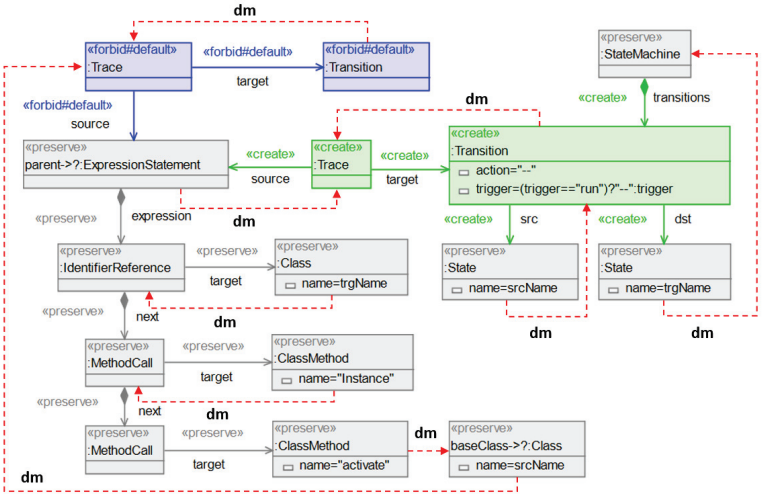



Fig. 4: The `createTransition` rule as directed graph

the structures. We, therefore, employ a bit-vector algorithm to match the encoded strings. This outputs all the instances of the participant string in the rule string.

After finding all instances of the participants of the design pattern in all the rules of the model transformation separately, the last phase recovers the complete design pattern instances inside the model transformation. While the first two phases are automated in our approach, the recovery phase is performed manually to consider the scheduling aspect of the transformation and the design pattern.

## 3.1 Encoding phase

The goal of the encoding phase is to represent a rule and a participant each in a unique circular string to employ a bit-vector algorithm to match them. We first transform the input artifacts into Eulerian directed graphs. Then, we derive the optimal Eulerian circuit of each graph. Finally, we encode the circuit as a unique circular string.

### *3.1.1 Rules and participants as Eulerian directed graphs*

It is natural to represent model transformation rules as graphs. For example, the graph transformation paradigm defines rules explicitly as graphs, such as in Henshin (see Section 2.1). Other model transformation paradigms, like ATL [18], can also be represented as graphs [39]. Therefore, without loss of generality, we consider model transformation rules as directed graphs for this work. Following the Henshin representation in Fig. 4, the *rule graph* consists of nodes and edges. Nodes and edges are typed by metamodel types, e.g., `ExpressionStatement` or `expression`, and

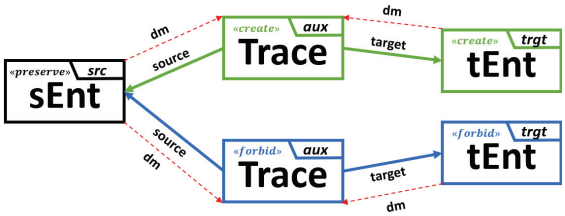



Fig. 5: The participant graph for the `entityMapping` participant of the E-R design pattern

attributed with an action, e.g., preserve, create, delete, forbid. Nodes and edges may be typed by special types, such as `Trace`, that are not found in the input/output metamodel of the model transformation, but used as auxiliary structures to facilitate the transformation.

Similarly, we represent design pattern participants as directed graphs. We follow the notation used by DelTa, with the difference of explicitly representing NACs like in Fig. 5. The nodes and edges in the *participant graph* are typed by the role name, e.g., `sEnt`, and attributed with an action. DelTa also employs built-in roles, such as `Trace`. We consider these nodes in our graph representation as *fixed nodes* for detection purposes.

It is important to convert the directed graph into a unique string representation of rules and participants. Furthermore, the strings must be circular so that a participant string may match any part of the rule string (and vice versa). We can achieve this by computing the minimum Eulerian circuit of each graph [20]. However, the directed graphs we obtain are not necessarily Eulerian, i.e., they do not contain a Eulerian circuit: a cycle that uses every edge of the graph exactly once. Following the method employed in [20] for class diagrams, we transform[1] the participant and rule graphs into unweighted directed Eulerian graphs. A graph is Eulerian if and only if every node is balanced: has equal in- and out-degrees. Therefore, we consistently add a *dummy edge* between unbalanced node. They are represented by dashed arrows in Fig. 4 and 5. Furthermore, in the case of an isolated node (like `baseClass` in Fig. 5), we add dummy edges to ensure its degree is at least two. There are many ways of making a graph Eulerian. The only requirement is to use the same strategy to make the rule and participant graphs Eulerian.

### *3.1.2 Computing the rule and participant strings*

Given that the rule and participant graphs are Eulerian, the optimal solution to the Chinese postman problem gives a Eulerian circuit [42]. This circuit starts and ends at the same node and traverses each edge exactly once: listing the nodes and edges we obtain a Eulerian trail. The circuit can start from any node, though our experiences have shown that starting with a node connected to a dummy edge may

[1] The algorithm is similar to the one presented in [20] based on the transportation simplex [8].

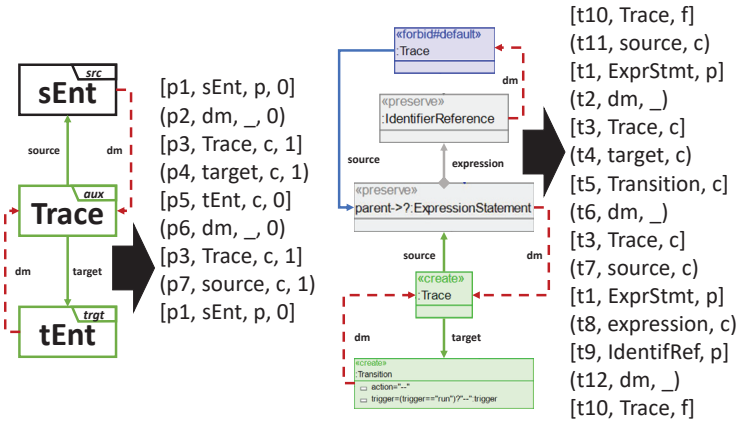



Fig. 6: (a) Eulerian trail of a part of the `EntitiesMapping` participant. (b) The corresponding participant string encoding. (c) Eulerian trail of a part of the `createTransition` rule. (d) The corresponding rule string encoding.

reduce the performance. Then, we transcribe the trail into a string representation. Note that, unlike edges, a node may appear more than once in the trail. This string is circular, and a graph may have many possible Eulerian trails. However, it is critical that the trail for the rules and participants are computed using the same strategy. Thus, we chose one strategy to obtain a unique trail for both. When traversing the graph, multiple trails are possible when a node has at least two outgoing edges. This is where we opt for a strategy that prioritizes the order in which to traverse edges. In our case, we give priority to non-dummy edges over dummy edges. Then, we base the order depending on the action of the edge/node: preserve, create, delete, forbid in this order. Finally, if a choice is available, we follow the alphabetical order of the values (e.g., `"source"` is chosen before `"target"`). The exact strategy is not relevant as long as it is the same strategy used for both rules and participants graphs to compute a unique Eurlerian circuit.

To represent the nodes and edges of the trail as string, we inspired ourselves from the string representation used for class diagrams in [20]. However, their approach is too simple to tokenize model transformations: we need to not only represent the connectivity of the graph, but also encode semantics specific to model transformations. Fig. 6 depicts the string encoding a Eulerian subgraph of a rule and a participant. We encode each node into a string token following the format $[id, type, action, fixed]$ for participants and $[id, type, action]$ for rules. We encode edges into string tokens similarly but between parenthesis. We refer to these strings as *role token* and *element token* respectively. *id* uniquely identifies the element. *type* is the type of the element. For the participant string, we use the name of the element as type, like `sEnt`. For the rule string, we use the name of the metamodel element as type, like `Transition`. In case of fixed nodes and edges for the participant string, like `Trace`, we set $fixed = 1$ otherwise 0. Fixing nodes enforces the detection

mechanism to match exactly the same element type. For example in Fig. 4, the *Transition* object should not match the *Trace* role because the transition is from the same metamodel as its *State* objects neighbors. The $action \in \{c, d, p, f, \_\}$ represents the create, delete, preserve, or forbid actions respectively. The underscore is used for dummy edges, for which no action is required, as they are not part of the original rule or participant. The graph is encoded by concatenating the tokens of the source node of an edge, followed by the edge, and then by the target node of the edge in the trail.

## 3.2 Detection phase

The goal of the detection is to find all instances of a design pattern participant, encoded in the participant string, that are present in the rule string. Bit-vector algorithms are known to be very efficient to approximate word matching in strings [30]. This requires to encode the strings into bit vectors. Kaczor et al. [20] have derived an iterative bit-vector processing algorithm to find exact and approximate occurrences of an object-oriented design pattern in a class diagram. Like for object-oriented patterns, this algorithm is also suitable for model transformations where pattern participants are encoded into short strings that can be found in longer strings representing transformation rules. However, the algorithm presented in [20] only deals with classes and associations in structural patterns. Since rule-based model transformation is a declarative paradigm, the original algorithm cannot be applied directly. Therefore, we significantly modify it to tackle the complexity that that model transformations and their design patterns provide.

### *3.2.1 Characteristic bit-vector*

To employ a bit-vector algorithm, we represent the participant and rule strings into vectors of bits. Let $t$ be any a token appearing in an encoded string $s = s_1 \dots s_n$, where $n$ is the length of the trail. We define the *characteristic vector* of $t$, denoted by $\vec{t} = t_1 \dots t_n$, as

$$t_i = \begin{cases} 1 \text{ if } s_i = t \\ 0 \text{ otherwise.} \end{cases}$$

Conceptually, it represents the positions of the token in the participant and rule strings. For example, in Fig. 6 (d), the characteristic vector of `t1` and `t4` are:

$$\texttt{[t1,Expr}\vec{\texttt{S}}\texttt{tmt,p]} \quad = \quad 001\underbrace{0000000}_{7}100 \text{ (i.e., two occurrences)}$$

$$\texttt{(t4,t}\vec{\texttt{a}}\texttt{rget,c)} \quad = \quad \underbrace{00000}_{5}1\underbrace{0000000}_{7} \text{ (i.e., one occurrence)}$$

Characteristic vectors are sequences of bits on which we can apply standard bit operations: bit-wise logical AND/OR operators and left/right shifts. A notable property of the way we construct the participant and rule strings is that tokens always appear in the same order modulo a shift. Thus, we consider that characteristic vectors are circular. For a characteristic vector $\vec{t} = t_1 t_2 \dots t_{n-1} t_n$, the left/right shifts are defined by shifting the position of the bits circularly by one to the left or the right as:

$$\underbrace{\vec{t} \ll 1 := t_2 \dots t_n t_1}_{\text{Left shift by 1}} \qquad \underbrace{\vec{t} \gg 1 := t_n t_1 \dots t_{n-1}}_{\text{Right shift by 1}}$$

*3.2.2 Matching participants in rules*

We rely on characteristic vectors to find the tokens of a transformation rule that play the role of a design pattern participant. Our algorithm, iteratively reads triplets of tokens $\langle node, edge, node \rangle$ in the participant string. It then identifies all possible matching triplets in the rule string by performing conjunctions and shifts. To ensure unification, we reuse tokens that are already matched in a triplet. Algorithm 1 separates the algorithm into two steps described in the subsequent algorithms.

**Algorithm 1:** DetectPatternInstances(participant, rule)

```
1  candidates ← FindCandidates(participant, rule)
2  maxSize = max |candidates[c]| ∀c ∈ candidates.keys()
3  matches ← ∅                                              // empty dictionary
4  foreach partTripl ∈ candidates.keys() do
5      pEdge, pBefore, pAfter ← partTripl.unpack()
6      matches[pEdge], matches[pBefore], matches[pAfter] ← Array(maxSize)
7  return Detect(candidates, 0, matches)
```

**Algorithm 2:** FindCandidates(participant, rule)

```
1  candidates ← ∅                                           // empty dictionary
2  foreach partToken ∈ participant.getEdges() do
3      tripl ← (partToken, partToken.before(), partToken.after())
4      candidates[tripl] ← ∅
5      foreach edgeTok ∈ rule.getEdges() do
6          if not Match(partToken, edgeTok) then continue
7          before ← ∅, after ← ∅                            // ordered sets
8          partT⃗oken ≫ 1
9          foreach afterTok ∈ rule do
10             conj⃗1 ← afterT⃗ok ∧ partT⃗oken
11             if conj⃗1 ≠ 0⃗ and Match(partToken.after(), afterTok) then
12                 after ← after ⊎ {afterTok}               // append
13                 conj⃗1 ≪ 2
14                 foreach beforeTok ∈ rule do
15                     conj⃗2 ← beforeT⃗ok ∧ conj⃗1
16                     if conj⃗2 ≠ 0⃗ and Match(partToken.before(), beforeTok) then
17                         before ← before ⊎ {beforeTok}    // append
18         for i = 0...Min(|before|, |after|) do // discarded if before or after are empty
19             candidates[tripl] ← candidates[tripl] ∪ {(edgeTok, before[i], after[i])}
20 return candidates
```

The first step is to find potential candidate rule tokens for each participant token. Algorithm 2 takes as input a participant and a rule strings to output a dictionary pairing each participant triplet to corresponding rule triplets. For example, suppose we wish to find the candidates of the triplet $\langle \texttt{p3}, \texttt{p7}, \texttt{p1} \rangle$ in Fig. 6.

We start by searching matches for the edge, i.e., `partToken = p7` on line 6. The MATCH function matches a token $p$ of the participant string to a token $t$ of the rule string following these rules:

1. If $p$ is a node then $t$ must be a node. If $p$ is an edge then $t$ must be an edge.
2. $t.action = p.action$
3. If $p.fixed = 1$ then $t.type = p.type$

In this case, `p7` matches `t7` and `t11`. The shifts and conjunctions on lines 8–17 ensure that the nodes before and after also match; otherwise the edge is discarded. The following shows how Algorithm 2 finds the rule triplet $\langle$`t3`, `t7`, `t1`$\rangle$ to match our participant triplet from the characteristic vectors:

$$\begin{aligned}
\vec{edgeTok} = \vec{t7}: \quad & \vec{p7} \gg 1 = 001\overbrace{0000000}^{7}100 \\
\vec{beforeTok} = \vec{t3}: \quad & \vec{p3} \gg 2 = 0010001000100 \\
\vec{conj}1: \quad & (\vec{p7} \gg 1) \wedge (\vec{p3} \gg 2) = 0010000000100 \\
\vec{afterTok} = \vec{t1}: \quad & \vec{p1} = 0010000000101 \\
\vec{conj}2: \quad & (\vec{p7} \gg 1) \wedge (\vec{p3} \gg 2) \wedge \vec{p1} = 0010000000100
\end{aligned}$$

On the last equation, we see how $\vec{conj}2$ rules out the candidate matching triplet $\langle$`t10`, `t11`, `t1`$\rangle$. On lines 18–19 of Algorithm 2, we only consider candidates that match the complete triplet of tokens.

**Algorithm 3:** DETECT(candidates, index, matches)

```
1  if index > |candidates.keys()| then return matches
2  partTripl ← candidates.keys()[index]
3  pEdge, pBefore, pAfter ← partTripl.unpack()
4  if pEdge.isDummy() and index > 0 then
5      return DETECT(candidates, index+1, matches)          // skip this triplet
6  for i = 0 . . . |candidates[partTripl]| do
7      rulTripl ← candidates[partTripl].pop()
8      rEdge, rBefore, rAfter ← rulTripl.unpack()
9      if matches[pEdge][i] = ∅ or index = 0 then
10         matches[pEdge][i] ← rEdge
11     if matches[pBefore][i] = ∅ then
12         matches[pBefore][i] ← rBefore
13     if matches[pAfter][i] = ∅ and (not pEdge.isDummy() or index > 0) then
14         matches[pAfter][i] ← rAfter
15     if VALID(partTripl, (matches[pEdge][i], matches[pBefore][i], matches[pAfter][i]))
         or index = 0 then
16         DETECT(candidates, index+1, matches)
17     else CLEARMATCH(matches, i)                          // discard this match
18 return matches
```

The candidate matches that Algorithm 2 outputs only consider triplets of tokens. Therefore, we must ensure that the whole participant matches cohesively among triplets. Algorithm 3 detects instances of each role of the participant. It creates a set of matches where a match pairs every role token from the participant

| Triplets | (p2,dm,_,0) | (p4,target,c,1) | | (p6,dm,_,0) | (p7,source,c,1) | |
|---|---|---|---|---|---|---|
| | [p1,sEnt,p,0] | [p3,Trace,c,1] | [p5,tEnt,c,0] | | [p3,Trace,c,1] | [p1,sEnt,p,0] |
| **Iterations** | | | | | | |
| 1 | [t1,ExprStmt,p] | | | | | [t1,ExprStmt,p] |
| 2 | [t1,ExprStmt,p] | [t3,Trace,c] | [t5,Transition,c] | | [t3, Trace,c] | [t1,ExprStmt,p] |
| 3 | [t1,ExprStmt,p] | [t3,Trace,c] | [t5,Transition,c] | «skip» | [t3, Trace,c] | [t1,ExprStmt,p] |
| 4 | [t1,ExprStmt,p] | [t3,Trace,c] | [t5,Transition,c] | | [t3, Trace,c] | [t1,ExprStmt,p] |
| 1 | [t9,IdentifRef,p] | | | | | [t9,IdentifRef,p] |
| 2 | [t9,IdentifRef,p] | [t3,Trace,c] | [t5,Transition,c] | | [t3,Trace,c] | [t9,IdentifRef,p] |
| 3 | [t9,IdentifRef,p] | [t3,Trace,c] | [t5,Transition,c] | «skip» | [t3,Trace,c] | [t9,IdentifRef,p] |
| 4 | [t9,IdentifRef,p] | [t3,Trace,c] | [t5,Transition,c] | | ~~[t3, Trace,c]~~ | ~~[t9,IdentifRef,p]~~ |

Fig. 7: Iterations of the Algorithm 3 showing matching nodes

string to an element token from the rule string. As shown on lines 4–6, Algorithm 1 initializes the data structure `matches` as dictionary where keys are individual roles of all participant triplets and values are empty arrays. Algorithm 3 recursively traverses each participant triplet and assigns the corresponding rule triplets. The loop on line 6 can easily be run in parallel since it processes each match of the participant independently. On lines 9–14, when assigning an element to a role, we first check if we have already assigned an element to that role before looking into the candidates output by Algorithm 2. This ensures the unification between the triplets of the same participant. On line 4, if a participant triplet contains a dummy edge, this triplet is skipped since it does not contribute to any role of the participant. The only exception is if the dummy edge is part of the initial triplet we are processing. In this case, we only assign the node before because the node after will be processed by the next triplet.

To illustrate Algorithm 3, Fig. 7 shows how we construct the matches for the participant string in (b) and the rule string in (d) of Fig. 6. Each row shows the current values of the `matches` dictionary at each iteration of the recursive calls (lines 5 and 16). The values of the iteration column correspond to the values of the `index` variable. At iteration 1, the initial triplet $\langle \texttt{p1}, \texttt{p2}, \texttt{p3} \rangle$ contains a dummy edge. Therefore, we only match `p1`. Algorithm 2 returns two possible candidates, `t1` and `t9`. They will be part of different matches in the match sets (line 2). Let us first consider the case of `t1`. At iteration 2, we process the next participant triplet to which we assign `t3` and `t5` to `p3` and `p5` respectively. At iteration 3, the next triplet contains a dummy edge, so it is ignored. At iteration 4, we notice that `p3` and `p1` are already assigned an element for the last triplet. On line 15, we verify the integrity of this match making sure they are well-formed according to Algorithm 2. Therefore, the function Valid verifies that the rule triplet matched exists in the `candidates` dictionary. Since it is the case, this match is an occurrence of the participant in the rule.

Let us now consider the second partial match where `p1` is assigned to element `t9`. The process is similar for iterations 2 and 3. At the last iteration, the integrity check fails since there is no edge from `t3` to `t9` that matches `p7`. This match is therefore discarded and line 17 only outputs the first match we described. The function ClearMatch removes the values at index `i` in all the arrays in `matches`.

### 3.3 Recovery phase

The detection phase outputs the rule elements matched to each role of a design pattern participant. Therefore, we obtain the set of rules that correspond to each participant. To reduce false positives, the recovery phase identifies the sets of rules corresponding to all the participants of a design pattern. Furthermore, the detection phase is based on the structure of individual participants.

To recover the complete design pattern instances, we must consider the global description of the design pattern and not only one of the participants. For instance, the scheduling scheme indicates whether the detected individual participants form a valid instance of the design pattern. DelTa offers several scheduling schema to order participants, such as sequencing, conditional bifurcation, choosing any participant, or not enforcing an order. Some model transformation languages, like Henshin, express rule ordering explicitly using similar schema. Others, like ATL, keep the order of the rules implicit. Therefore, a deep knowledge of the semantics of the transformation language is needed to decide whether the detected rules are executed in the order prescribed by the design pattern. There are other components, such as constraints and actions on attributes of rule elements or profiles/tags in DelTa participants, that should also be considered to recover complete design pattern instances. The mapping between the description of these additional elements (scheduling and other components) and the concrete transformations is not straightforward, because many options can be used for their implementations. This is why in the current state of our research, we perform the recovery phase manually.

To illustrate the recovery phase, let us consider the cas of the E-R pattern. Suppose that, during the detection phase, we found two instances E1 and E2 of the *entityMapping* participant, and two instances R1 and R2 of the *relationMapping* participant. The pattern suggests that the former should precede the latter (see Fig. 2). After analyzing the scheduling scheme of the model transformation, we identify that E1 is executed before R1, but E2 is executed after R2. Therefore, we conclude that the E-R pattern has only one valid instance E1-R1.

## 4 Detecting other design pattern specifications

Typically, the solution described by a design pattern is a generic template used as a guideline to understand how to implement the design pattern. In practice, it happens often that we implement a design pattern in a different way than the template solution. Therefore, a design pattern detection approach should consider not only the exact template solution provided in the definition of the design pattern, but should also detect other implementation variants that may be encountered in practice. Furthermore, existing model transformations may only partially implement a design pattern. Our detection process in Fig. 3 takes as input the specification of a design pattern participant. However, the participant does not necessarily need to be the complete or exact participant defined in the solution template of the design pattern. It can be an approximation of that participant, that is missing some roles or variant with an alternative structure that still preserves the the global description of the design pattern. Another important aspect to consider is that patterns may use other patterns in their implementation. Patterns can be speci-

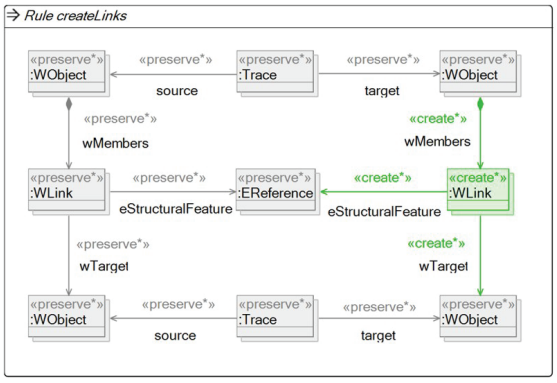



Fig. 8: Instance of a variant of the `relationMapping` participant

fied independently, but their detected occurrences can be checked with respect to the conformity with the expected usage relationships. In the remainder of this section, we characterize variants and approximations of model transformation design patterns and show that we can reuse the exact same detection process presented in Section 3 to detect them. Moreover, we illustrate some of the common usage relations between the considered patterns.

### 4.1 Variants

Variants of a design pattern modify the template solution, but they must preserve the pattern description. For example, the observer pattern in object-oriented design has one variant for the pull and another for the push approach. There is no general way of deriving all possible variants of a design pattern.

Many model transformation design pattern propose possible variants (called "variations" in [11, 22]). They can take various forms, such as using more roles in a participant, changing the type of a relation, or changing the action of a role. For example, for the E-R design pattern, the relation between the source and target entities can be many-to-one instead of the relation mapping one source to one target entity, as depicted in Fig. 2.

Consider the Henshin rule `createLinks` in Fig. 8. It implements a variant of the participant *relationMapping* in the E-R design pattern. First, we note that the single relation between `sEnt` and `sEnt2` is implemented by means of a `WLink` object with `wMembers` and `wTarget` associations. Secondly, we note that a single relation between `tEnt` and `tEnt2` is also implemented by means of a `WLink` object. Thirdly, this object has an additional `eStructuralFeature` association that is not required by the design pattern participant. This rule implements a typical variant of the design pattern where a participant or role of the design pattern is implemented by more than one element.

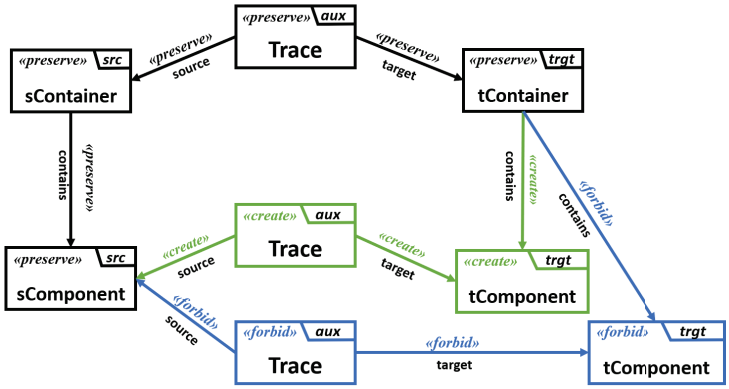



Fig. 9: The *latterPhase* participant of the Top-Down design pattern

### 4.2 Approximations

Sometimes, a design pattern is not implemented in its entirety in a transformation [28]. Although the underlying implementation may match part of a design pattern description, some roles of the participants may be missing. This typically has a negative effect on the quality of the transformation, such as reuse, cohesion, or coupling [22]. We consider an implementation as an approximation of a design pattern if it is missing or misusing roles.

Consider the Top-Down design pattern that decomposes a transformation into phases based on the target model composition structure [11]. It is composed of two participants, *formerPhase* and *latterPhase*, sequenced in this order. The *formerPhase* states that a rule shall create a `tContainer` entity in the target metamodel. This entity shall be traced to the corresponding `sContainer` entity in the source metamodel that contains an `sComponent` entity. The *latterPhase* (represented in Fig. 9) states that a rule shall create a `tComponent` entity contained in `tContainer`. This entity shall be traced to the corresponding `sComponent` entity in the source metamodel.

Fig. 10 shows a Henshin rule stating that, if a `GenClass` is associated with an `EClass` containing an `EAttribute`, then the `GenClass` should contain a `GenFeature` and associate it to the `EAttribute`. Intuitively, we understand that it corresponds to the *latterPhase* participant of the Top-Down design pattern. However, we note differences between the two structures. To be a perfect match, the rule is missing a relation from `GenClass` to the forbidden `GenFeature` to complete the NAC of *latterPhase*. Furthermore, the participant requires using a `Trace` element to link the created `GenFeature` to the `EAttribute`. According to our representation, `Trace` would be a fixed role. However, this role is played by the `ecoreFeature` link and the `Rel` element. Since the rule reuses an element from the metamodel to implement a fixed role, they would not match. We consider this situation as a misuse of a role. Therefore, for these two reasons, this rule is an approximation of the `latterPhase` participant in the Top-Down design pattern.

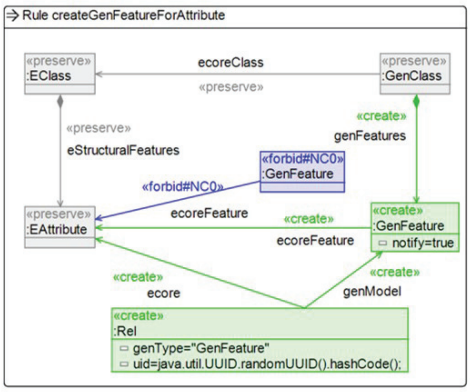



Fig. 10: Approximation of the `latterPhase` participant in Fig. 9

Design pattern participant variants or approximations can be derived manually to ensure their conformance with the corresponding design pattern. Some types of approximation can be derived automatically by, e.g., removing some roles. Some variants can be derived automatically once the transformation language is fixed. Kaczor et al. [20] define four different types of approximations for object-oriented design patterns, though they acknowledge there are more. These can be summarized in two major categories: when the role of a participant is missing or when a role is misused. Our approach can automatically detect model transformation design pattern approximations that fall under the first category. Given the Eulerian trail of a participant, we can automatically derive approximations by removing elements of the trail. The bit-vector algorithm remains the same. Therefore, for some approximations, we do not require to manually specify them (as opposed to [28]). The second category of approximations covers cases when the use of a role deviates from the original specification of the design pattern. Kaczor et al. propose ways to modify their algorithm to handle these cases. In some cases, they acknowledge that the approximation must be specified manually. Our detection approach can recover variants and approximations of a design pattern as long as they are explicitly defined. We apply the process in Fig. 3 where the input is a variant.

### 4.3 Co-occurrence of design patterns

Looking at the design patterns catalog, we notice that the structure of the participants of some design patterns are present in participants of other design patterns. This suggests that some design patterns are related with others. As outlined in Fig. 11, we distinguish at least three types of relations. The relations we present are the result of a critical analysis of all catalogued design patterns in [11, 22] complemented with the notion of design pattern relations introduced in [24].

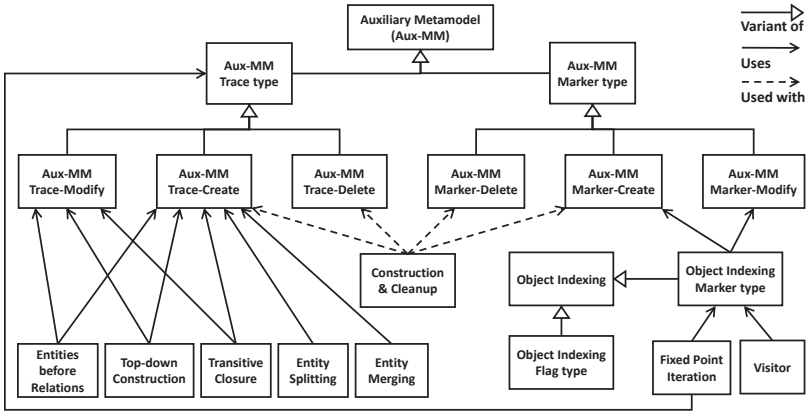



Fig. 11: Relations between some design patterns

When analyzing existing model transformation implementations, we identified variants of design patterns used in practice. As defined in Section 4.1, all variants of a design pattern share the same goal, but differ in the structure of their participants. A predominantly used design pattern in practice is the *Auxiliary Metamodel*. *"This pattern proposes to create an auxiliary metamodel for temporary elements used in the transformation that do not belong to either the source or target metamodels"* [22]. We have noted two main variants. The *Trace type* is a variant of this pattern that relies on an entity to keep a trace correspondence between entities of different metamodels. This variant is mostly used in exogenous transformations. However, for in-place transformations, we noted that developers often rely on a specific entity (from the same or another metamodel) to temporarily mark entities to be transformed, such as the rule in Fig. 12 taken from the `Ecore2GenModel` transformation[2]. The generic solution of the *Auxiliary Metamodel* pattern presents three possible variants of how it can be used when creating a new correspondence to an existing entity, deleting the corresponding entity, or modifying an attribute of the corresponding entity. We, therefore, obtain the six variants mentioned in Fig. 11.

The *Object Indexing* pattern uniquely indexes an entity in a first participant to enable its efficient lookup in a second participant. We identified that it comes in two variants: one using flag to mark an entity to index and another using a temporary marker pointing to the entity. The latter variant essentially uses the *Marker type* variant of the *Auxiliary Metamodel* pattern to index an entity. For this relation, if a pattern $A$ *uses* a pattern $B$, we will often find the structure of a participant of $B$ subsumed in a participant of $A$. For example, the *Visitor* pattern requires to mark an entity as already visited. This essentially uses an instance of the *Object Indexing* pattern. Another example of this relation is that the *E-R* pattern (see Fig. 2) presents an instance of the *Trace type* variant of the *Auxiliary*

[2] `https://www.eclipse.org/henshin/examples.php?example=ecore2genmodel`

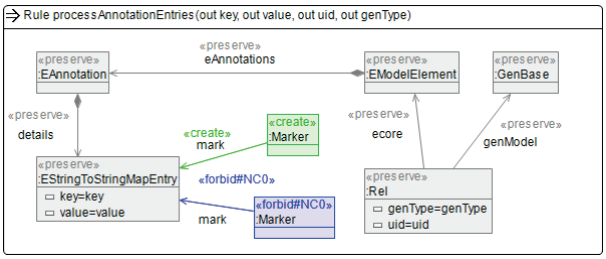



Fig. 12: A rule using markers

*Metamodel* pattern. More precisely, the *entityMapping* participant uses the create variant and the *relationMapping* uses the modify variant.

The third type of relation we identified is when a design pattern is often *used with* another one. For example, the *Construction & Cleanup* pattern often creates temporary entities at the beginning of the transformation and deletes them at the end. Therefore, it is often used with the *Auxiliary Metamodel* pattern to achieve its goal.

Although, the detection process does not consider explicitly usage relations between patterns, the detected occurrences can be checked to ensure that patterns are used correctly by other patterns. If not, these situations can be considered as potential refactoring opportunities.

## 5 Evaluation

To evaluate the usefulness and the relevance of our approach, we defined the following research questions:

**RQ1: How are design patterns used in model transformations?** The goal of this question is to understand if design patterns are actually employed in concrete transformations and in what form they are used (complete, variants, approximations). This also serves as a motivation for our approach.

**RQ2: How effective is our approach to detect design pattern variants and approximations?** With this question, we evaluate the ability of our approach to detect complete and partial pattern occurrences, whether they are standard patterns or variants.

**RQ3: What design patterns are typically used in combination with each other?** This question allows us to identify relations between design patterns and validate the co-occurrence premise that we describe in Section 4.3.

### 5.1 Setup

In our evaluation, the goal is to analyze quantitatively and qualitatively how our detection approach applies to concrete transformations. We relied on model transformations gathered from three sources as shown in Table 1. The selected transformations are described by the number of rules they contain, as well as by the

| Sources | Model transformations | # rules | # nodes | # relations | # min nodes/rule | # max nodes/rule |
|---|---|---|---|---|---|---|
| Henshin Website | ecore2genmodel | 8 | 55 | 59 | 4 | 6 |
| | ecore2rdb | 2 | 38 | 49 | 13 | 25 |
| | ecore2uml | 2 | 29 | 40 | 9 | 20 |
| | java2statemachine | 13 | 77 | 59 | 2 | 15 |
| | wrap copy | 3 | 20 | 19 | 4 | 9 |
| | classDiagram2relationSchema | 3 | 16 | 14 | 5 | 6 |
| ATL Zoo | book2publication | 5 | 20 | 16 | 2 | 6 |
| | ATOM2RSS | 17 | 73 | 59 | 2 | 6 |
| | RSS2ATOM | 6 | 28 | 27 | 3 | 6 |
| SecBPMN2-to-UMLsec | GR2_classOperation | 31 | 262 | 267 | 6 | 10 |
| | GR3_associations | 35 | 640 | 687 | 15 | 20 |
| | GR4_dependency | 28 | 393 | 440 | 12 | 15 |
| | GR5_abac | 15 | 131 | 131 | 7 | 9 |
| | GR5_secureDependency | 33 | 295 | 292 | 5 | 10 |
| | GR6_secureLinks | 29 | 323 | 374 | 2 | 15 |
| | GR7_secureLinks2 | 30 | 426 | 510 | 11 | 16 |
| | GR7_secureLinks3 | 24 | 396 | 444 | 15 | 18 |
| | GR7_secureLinks4 | 17 | 240 | 261 | 6 | 18 |

Table 1: Selected model transformations

total number of Henshin nodes and relations respectively involved in these rules. We also provide the minimum and maximum number of nodes per rules for each transformation.

The first source we used is the Henshin website[3]. We selected six exogenous transformations. To add more diversity to our transformation sample, we also considered ATL transformations from the ATL Zoo[4]. Although the literature proposes higher order transformations from ATL to graph transformation [33, 39], we were unable to reuse their tool support. Therefore, we limited ourselves to three ATL transformations because we had to reimplement them manually in Henshin following the guidelines of [33, 39]. Finally, the third source of transformations is the ReMoDD project reported in [37]. This project contains 10 model transformations that map a secure business process model, expressed in SecBPMN2, to a preliminary architectural UML model enriched with security policies, expressed in UMLsec. The size characteristics of these transformations are much higher than ones of the transformations from the two previous sources. It is worth mentioning that the selected transformations from the above-mentioned sources are exogenous transformations as most of the design patterns are present in exogenous transformation by nature. We did not consider endogenous transformations in these sources.

[3] `https://www.eclipse.org/henshin/examples.php`

[4] `https://www.eclipse.org/atl/atlTransformations/`

Table 2: Selected Design patterns

| **Patterns** | **Summary** | **Benefits** |
|---|---|---|
| *Auxiliary metamodel* | This pattern proposes to create an auxiliary metamodel for temporary elements used in the transformation that do not belong to either source or target metamodels. | Improves clarity, flexibility, and modularization |
| *Construction & cleanup* | This pattern structures a transformation by separating rules which construct model elements from those which delete elements. | Improves modularity |
| *Entities before relations* | This pattern is used in exogenous transformations to encode a mapping between two languages. It creates the elements in a language corresponding to elements from another language and establishes traceability links between the elements of source and target languages. | Improves debugging, error localization, and avoids circularity in processing |
| *Entity merging* | Two rules each create/update elements of the same target entity type, using different source entity types or elements. | Organizes instance data integration |
| *Entity splitting* | Two rules (with disjoint application conditions) map (instances of) the same source entity type to instances of different target entities. | Distinguishes cases in source data |
| *Fixed-point iteration* | Pattern for representing a "do-until" loop structure. It solves the problem by modifying the input model iteratively until a condition is satisfied. | Organizes fixed-point processing |
| *Object indexing* | All objects of an entity are indexed by a primary key value, to permit efficient lookup of objects by their key. | Reduces syntactic complexity |
| *Top-down phased construction* | This pattern decomposes a transformation into phases or stages, based on the target model composition structure. These phases can be carried out as separate subtransformations, composed sequentially. | Organizes processing |
| *Unique instantiation* | This pattern makes sure the created elements in a rule are unique and eliminates redundant creation of the same element by reuse. | Avoids duplicating instances |
| *Visitor* | This pattern traverses all the nodes in a tree and processes each entity individually. | Improves extensibility |

We considered all 15 design patterns from [11] to detect their instances on our sample of model transformations. However, we found no occurrences of five of them in the transformations we consider. Thus, Table 2 presents the sample of 10 design patterns we selected with a brief description and benefits. This information is extracted from the design pattern catalogs presented in [11, 22, 24]. The transformation patterns we use come from different categories [22]. The first category deals with rule modularization patterns. It includes the design patterns: *Auxiliary Metamodel*, *Construction & Cleanup*, *Entities before relations* (also called *Map objects before links* in [24]), *Entity Merging*, and *Entity splitting*. These patterns

aim at organizing the dependencies and relationships between rules in a transformation. *Unique Instantiation* and *Object Indexing* are two patterns from the optimization category. They are used to improve the execution efficiency at the rule or transformation level. According to Lano et al. [22], *Unique Instantiation* and *Auxiliary Metamodel* can be seen as fundamental patterns which forms a separate category. From the classical/external patterns category, we selected *Visitor*. The latter is heavily used in model-to-text transformations to navigate through a model. Finally, we selected two patterns from the architecture category: *Top-down Phased Construction* and *Fixed-point Iteration*. Both patterns are used to organize systems of transformations at the inter-transformation level to improve modularity and processing capabilities of systems. For more information about the selected patterns, we refer the reader to [23], which presents a survey on how these pattern are actually used in real transformations.

### 5.2 Design pattern usage in model transformation (RQ1)

To answer the first research question, we considered all the transformations described in Table 1. We performed a manual inspection of these transformations to detect all existing instances of the design patterns in Table 2. For each pattern, one co-author identified its usage in all the transformations and the other two validated the results. When manual detecting design patterns we first identify the rules matching the patricipants of the design pattern, and then we verify that the logic of the rule scheduling and attribute constraints conform to the constraints of the pattern. We had to consider complete and approximate patterns, as well as variants. The occurrences we found are presented in Table 3.

A quick glance at the results reveals that the *SecBPMN2-to-UMLsec* transformations contains higher numbers of pattern occurrences than those of the other transformations. This is expected considering the size of these transformations. Among the design the patterns, *Auxiliary Metamodel* is the most used. This was expected as this is fundamental pattern that allow different rules to manipulate the same objects in exogenous transformations. We distinguish between the three variants of the pattern encompassing the *Trace* and *Marker* versions (see Fig. 11). The *Create variant* is consistently used in all transformations because they are all creating elements in the target model. The *Modify variant* is found when a rule updates elements it matches. Since we found many instances of the *Create variant*, we expected to find many instances of the *Modify variant* as well. That is because the *Modify variant* preserves the existence of the elements created. Usually, the *Auxiliary Metamodel* variants are used together. We found only few instances of the *Delete variant* because not all transformations intend to remove elements. This is expected as we are dealing with exogenous transformations.

The *Construction & Cleanup* pattern is based, in general, on the combination of the *create* and *delete* variants of the *Auxiliary Metamodel* pattern. Most of developers use the *delete* variant in a *Construction & Cleanup* process. Thus, the instances we found for this pattern are almost the same as ones we found for the *Auxiliary Metamodel*.

*Entity Merging* and *Entity Splitting* are mostly used in *SecBPMN2-to-UMLsec* transformations. Similarly, we found a high number of instances of *Entities before Relations* within the third source and very few in transformations of the other

| **Model transformations** | *Auxiliary Metamodel (Create variant)* | *Auxiliary Metamodel (Modify variant)* | *Auxiliary Metamodel (Delete variant)* | *Construction & Cleanup* | *Entities before Relations* | *Entity Merging* | *Entity Splitting* | *Fixed-Point Iteration* | *Object Indexing* | *Top-down Phased Construction* | *Unique Instantiation* | *Visitor* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ecore2genmodel | 4 | | | | | | | | | 3 | 3 | |
| ecore2rdb | 3 | 3 | | | | | | | | 2 | 1 | |
| ecore2uml | 5 | 4 | | | | 1 | | | | 4 | 4 | |
| java2statemachine | 10 | | 1 | 1 | | | | X | | 1 | | |
| wrap copy | 1 | 3 | | | 1 | | | | | 3 | 3 | |
| classDiagram2relationSchema | 2 | 2 | | | 1 | | | | | | 2 | |
| book2publication | 3 | 3 | 1 | 1 | 1 | | | | 1 | | 2 | |
| ATOM2RSS | 8 | 17 | 3 | 3 | 2 | | | | 4 | | 4 | X |
| RSS2ATOM | 6 | 4 | | | 2 | | 2 | | | | 5 | |
| GR2_classOperation | 19 | 41 | | | 12 | 3 | 6 | | | 18 | | |
| GR3_associations | 35 | 58 | | | 23 | 1 | 3 | | | 58 | | |
| GR4_dependency | 28 | 56 | | | 37 | 1 | 1 | | | 56 | | |
| GR5_abac | 15 | 15 | | | | 1 | 2 | | | 15 | | |
| GR5_secureDependency | 30 | 33 | | | | 1 | 2 | | | 30 | | |
| GR6_secureLinks | 20 | 39 | | 6 | 14 | 4 | 6 | | | 10 | | |
| GR7_secureLinks2 | 30 | 48 | | 9 | 18 | 2 | 2 | | | 48 | | |
| GR7_secureLinks3 | 24 | 48 | | | 24 | 1 | 1 | | | 48 | | |
| GR7_secureLinks4 | 17 | 20 | | | 9 | 2 | 1 | | | 20 | | |

Table 3: Existing design pattern instances fetched by hand

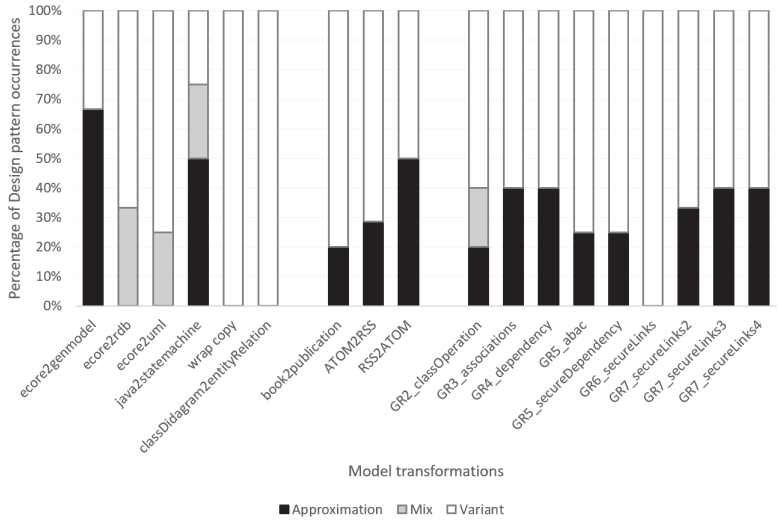



Fig. 13: Nature of design pattern occurrences in model transformations

sources. The same observations also hold for the *Top-down Phased Construction* pattern.

Among the remaining patterns, *Object Indexing* does not seem to be used in the majority of transformations. This is not surprising as this pattern is more useful for endogenous transformations. We found instances of the *Unique Instantiation* pattern in almost all transformations from Henshin and ATL. However, we found none in *SecBPMN2-to-UMLsec* because, given the nature of its large transformations, the same element is created several times. Finally, we found a very limited number of instances of *Fixed-point iteration* and *Visitor* patterns because they do not fit the implementation of most of the transformations in our sample. Since these two patterns are generally implemented by the whole transformation, and not an isolated subset, we indicate their occurrence with an "X" in Table 3.

From the perspective of the pattern nature, most of the found occurrences are different variants that implement the generic pattern. The other occurrences are approximations. Fig. 13 summarizes the distribution of the nature of occurrences found for each transformation. For each transformation rule involved in a design pattern, we report if it implements a variant of the pattern (white) or an approximation (black). The Mix category (gray) means that we found both variants and approximations occurrences of the same design pattern in the transformation. Except for *ecore2genmodel*, *java2statemachine*, and *RSS2ATOM*, rules involved in variants account for more 60%, reaching 100% for three transformations. We found rules involved in both variants and approximations in only four transformations.

For RQ1, we can conclude that design patterns are used in transformations, which concurs with the survey in [24]. Some of the patterns occur more frequently than others. Variants are more frequent than approximations. Finally, transfor-

mation rules can be involved in more than one occurrence and sometimes in both variants and approximations. This result clearly motivates the need for design pattern detection approaches able to detect variants and approximations.

## 5.3 Detecting design pattern variants and approximations (RQ2)

To answer the second research question, we applied our detection approach on all the selected model transformations in two steps: first for variants, then for approximations. Table 4 presents the results of both steps. Note that we consider two kinds of accuracy in the detection of patterns: one for the detection of participants and one for the recovery of the complete design pattern. In our experiments, there are no false positives for the former. However, when taking into account the constraints of the pattern (e.g., scheduling, attribute constraints), we manually filtered participants and their scheduling leading to false positives.

### *5.3.1 Detecting design pattern variants*

In a first step, we specified all design patterns and their different variants (described in Section 4) as input. Note that the *Auxiliary Metamodel* pattern does not involve rule scheduling. Hence, our approach, in its current state, is able to detect it fully automatically without the need for a manual recovery phase. For the other patterns, we combined automated detection of pattern participants with the recovery phase, as explained in Section 3.3.

From RQ1, most of the design pattern instances we expect are variants. The numbers on the left side of the slash in Table 4 show the results of detecting the variants of each design pattern. Our approach has detected all the expected instances of the design pattern variants and we found no false positive. The manual recovery phase, in addition to the verification of the scheduling, allows us to eliminate pattern participants for which the other participants to complete the pattern occurrences are not present.

### *5.3.2 Detecting design pattern approximations*

The remaining instances reported in Table 3 are approximations. Therefore, in the second step, we performed the detection on the 18 transformations with the pattern approximations of Section 4 as input. The results of detecting approximations are presented in Table 4.

Like for the variants, we detected all the expected instances of design pattern approximations. However, we found false positives for the *Auxiliary Metamodel* pattern approximations. When examining these occurrences, we discovered that they are resulting from the combination of two factors: (1) the specification of the approximation of a design pattern is very similar to the specification of one of its variants, and (2) the inability of our approach to express negative conditions in the detection.

More specifically, the variants using *Marker* allow, among others, to tag elements as already processed when executing a rule on a list of elements. For example, in the rule on the top left of Fig. 14, a `Chapter` is marked after adding the number of its pages in the `Publication`. In contrast, in some transformations, the

| Model transformations | *Auxiliary Metamodel (Create variant)* | *Auxiliary Metamodel (Modify variant)* | *Auxiliary Metamodel (Delete variant)* | *Construction & Cleanup* | *Entities before Relations* | *Entity Merging* | *Entity Splitting* | *Fixed-Point Iteration* | *Object Indexing* | *Top-down Phased Construction* | *Unique Instantiation* | *Visitor* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ecore2genmodel | /4 | | | | | | | | | /3 | 3/ | |
| ecore2rdb | 3/ | 2/1 | | | | | | | | /2 | 1/ | |
| ecore2uml | 5/ | 4/ | | | | 1/ | | | | 4/ | 4/ | |
| java2statemachine | 1/9 | | /1 | /1 | | | | /X | | /1 | | |
| wrap copy | 1/ | 3/ | | | 1/ | | | | | /3 | 3/ | |
| classDiagram2relationSchema | 2/ | 2/ | | | 1/ | | | | | | 2/ | |
| book2publication | 3/ | 3/ | 1/ | 1/ | 1/ | | | | | | 2/ | |
| ATOM2RSS | 8/ | 17/ | 3/ | 3/ | 2/ | | /1 | | /1 | | 4/ | /X |
| RSS2ATOM | 6/ | 4/ | | | 2/ | | /2 | | 4/ | | 5/ | |
| GR2_classOperation | 19/ | 41/ | | | /12 | 3/ | 6/ | | | 11/7 | | |
| GR3_associations | 35/ | 58/ | | | /23 | 1/ | 3/ | | | /58 | | |
| GR4_dependency | 28/ | 56/ | | | /37 | 1/ | 1/ | | | /56 | | |
| GR5_abac | 15/ | 15/ | | | | 1/ | 2/ | | | 15/ | | |
| GR5_secureDependency | 30/ | 33/ | | | | 1/ | 2/ | | | /30 | | |
| GR6_secureLinks | 20/ | 39/ | | 6/ | 14/ | 4/ | 6/ | | | 10/ | | |
| GR7_secureLinks2 | 30/ | 48/ | | 9/ | /18 | 2/ | 2/ | | | /48 | | |
| GR7_secureLinks3 | 24/ | 48/ | | | /24 | 1/ | 1/ | | | /48 | | |
| GR7_secureLinks4 | 17/ | 20/ | | | /9 | 2/ | 1/ | | | /20 | | |

Table 4: The variants and approximations of design pattern instances detected by our approach. $x/y$ denotes $x$ variants and $y$ approximations.

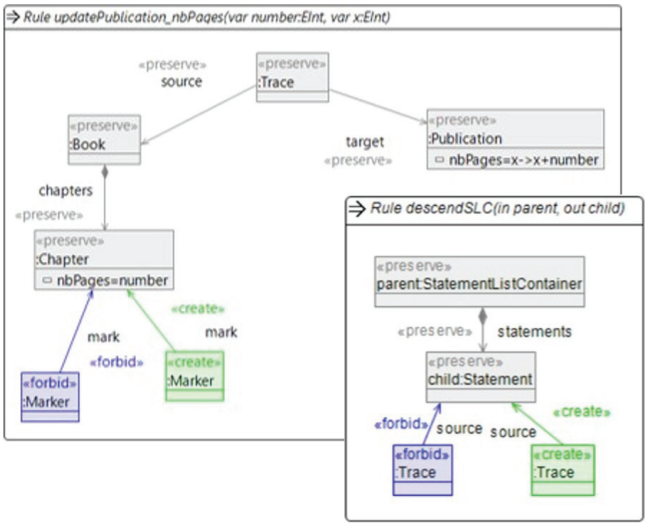



Fig. 14: *Trace* as an approximation of *Marker* for *Auxiliary metamodel* pattern.

*Trace* variant is used for the same purpose, like in the rule on the bottom right of Fig. 14. However, the *Trace* is not connected to a target element as defined by the design pattern. Thus, we consider this use of a `Trace` as an approximation. This approximation results in some false positives as shown in Fig. 15, because with our detection approach, it is not possible to express the negative condition that `Trace` should not be connected to a target element.

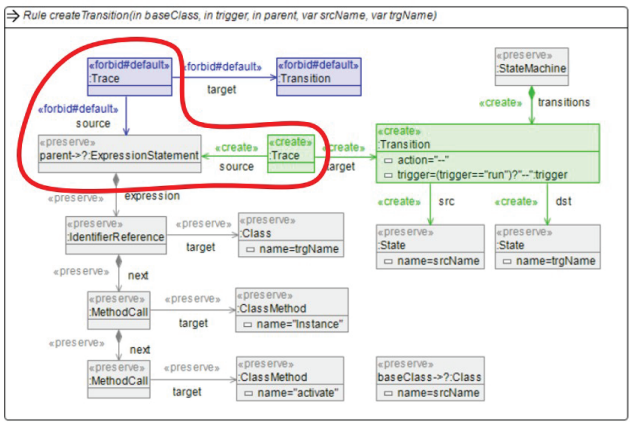



Fig. 15: Example of a false positive with the *Trace* vs *Marker* approximation.

### 5.4 Pattern combinations (RQ3)

To answer the third research question, we analyzed the pattern instances detected by our approach to assess whether *uses* dependencies between patterns depicted in Fig. 11 actually exist in transformations. More specifically, we examined which pattern participants are detected in each model transformation rule and checked the scheduling to determine the co-occurrences.

To measure the consistency with which co-occurrences of patterns are found, for each pair of patterns, we calculate the percentage of times the used pattern instances appear with the using pattern instances. For a pair of patterns $\langle userP, usedP \rangle$ from Fig. 11, let us consider the set $P$ of pairs of pattern occurrences of respectively $userP$ and $usedP$ in a transformation $T_k$:

$$P = \{(p_{ik}, p_{jk}) \mid p_{ik} \in userP \wedge p_{jk} \in usedP\} \tag{1}$$

For each pair in $P$, we consider that $p_{ik}$ uses $p_{jk}$ if all the participants of $p_{jk}$ appear simultaneously with the participants of $p_{ik}$ in the same rules of $T_k$. Furthermore, we distinguish between different usage levels depending on whether $p_{jk}$ is an instance of a variant or an approximation of $usedP$. Thus, we assign a usage level $uselev(p_{ik}, p_{jk})$ between $p_{jk}$ and $p_{jk}$ as follows:

$$uselev(p_{ik}, p_{jk}) = \begin{cases} 1 & p_{ik} \text{ uses a variant instance } p_{jk}, \\ 0.5 & p_{ik} \text{ uses an approximation instance } p_{jk}, \\ 0 & p_{ik} \text{ does not use } p_{jk}. \end{cases} \tag{2}$$

Following these definitions, we calculate, for each transformation $T_k$ in our transformation set, the usage score $usage(userP, usedP, T_k)$ as the average usage level of pairs of instances of $userP$ using instances of $usedP$. Note that, by the pattern specification, an instance of $userP$ may use more than one instance of $usedP$. Then, the average is calculated with respect to the number of instances of $userP$ in $T_k$ multiplied by the number $mf(userP, usedP)$ of instances of $usedP$ an instance of $userP$ is expected to have. Formally:

$$usage(userP, usedP, T_k) = \frac{\sum_{(p_{ik}, p_{jk}) \in P} uselev(p_{ik}, p_{jk})}{|\{p_{ik} \mid p_{ik} \in userP\}| \times mf(userP, usedP)} \tag{3}$$

Table 5 shows the usage scores between the user patterns (in columns) and the used patterns (in rows) for the considered transformations. We only report a score when the user patterns exist in the transformations. For the sake of conciseness, we aggregated the scores of all the transformations of *SecBPMN2-to-UMLsec*.

A first observation is that in all but a few cases, we confirmed empirically the use relationships of Fig. 11. For example, the instances of *Entities before Relations* consistently use the instances of the two variants *Trace Create* and *Trace Modify* of the *Auxiliary Metamodel* pattern, as indicated by a usage score of 100%. Similarly, although less frequent, *Entity Splitting* and *Entity Merging* also use in 100% of the cases *Trace Create* variant. We also obtained perfect usage scores (100%) for three other user patterns. *Construction & Cleanup* always uses *Create* and *Delete* for both *Trace* and *Marker* variants of the *Auxiliary Metamodel* pattern. *Visitor*

| | *Entities before relations* | *Entity splitting* | *Entity merging* | *Top-down construction* | *Construction & Cleanup* | *Object indexing* | *Visitor* | *Fixed-point iteration* | Transformations |
|---|---|---|---|---|---|---|---|---|---|
| ***Aux-MM Trace-Delete*** | | | | | 100% | | | | Java2StateMachine |
| ***Aux-MM Trace-Modify*** | | | | 50% | | | | | Ecore2GenModel |
| | 100% | | | 100% | | | | | Ecore2Rdb |
| | | | | 100% | | | | | Ecore2Uml |
| | | | | 100% | | | | | Java2StateMachine |
| | 100% | | | 67% | | | | | wrap-copy |
| | 100% | | | | | | | | classDiag2relationSche |
| | 100% | | | | | | | | Book2Publication |
| | 100% | | | | | | | 100% | ATOM2RSS |
| | 100% | | | | | | | | RSS2ATOM |
| | 100% | | | 100% | | | | | SecBPMN2-to-UMLsec |
| ***Aux-MM Trace-Create*** | | | | 50% | | | | | Ecore2GenModel |
| | 100% | | | 100% | | | | | Ecore2Rdb |
| | | | 100% | 100% | | | | | Ecore2Uml |
| | | | | 100% | 100% | | | | Java2StateMachine |
| | 100% | | | 67% | | | | | wrap-copy |
| | 100% | | | | | | | | classDiag2relationSche |
| | 100% | | | | | | | | Book2Publication |
| | 100% | 100% | | | | | | | ATOM2RSS |
| | 100% | | | | | | | | RSS2ATOM |
| | 100% | 100% | 100% | 100% | | | | | SecBPMN2-to-UMLsec |
| ***Aux-MM Marker-Delete*** | | | | | 100% | | | | Book2Publication |
| | | | | | 100% | | | | ATOM2RSS |
| ***Aux-MM Marker-Modify*** | | | | | | 100% | | | Book2Publication |
| | | | | | | 100% | 100% | 100% | ATOM2RSS |
| ***Aux-MM Marker-Create*** | | | | | | 50% | | | Java2StateMachine |
| | | | | | 100% | 100% | | | Book2Publication |
| | | | | | 100% | 100% | 100% | 100% | ATOM2RSS |
| ***Object indexing*** | | | | | | | 100% | 100% | ATOM2RSS |

Table 5: Usage scores for pairs of patterns

and *Fixed-point Iteration* use the *Marker Create* and *Marker Modify* variants of *Marker* type, as well as *Object Indexing* in all the detected instances.

The only two patterns for which we did not observe perfect scores for all the cases are *Top-down Phased Construction* and *Object Indexing*. The former uses the *Create* and *Modify* variants with scores of 100% in all the transformations but two. In `Ecore2GenModel`, the developer implementing *Top-down construction* used approximations of variants of the *Auxilary Metamodel* pattern, which explains the scores of 50%. In the second transformation `wrap-copy`, the score is slightly better (67%). We found that the developer used approximations of variants of this pattern only in some cases.

For *Object Indexing*, the only transformation in which we had unexpected results is `Java2StateMachine`. This design pattern is expected to use the *Marker Create* to index elements and *Marker Modify* to manipulate them in another rule. In the case of create, the developer always used *Trace* as an approximation which led to the score of 50%. The *Modify* variant was not used at all as the developer exploited another mechanism to access the indexed objects, i.e., rule invocation in the scheduling with the indexed elements passed as parameters.

It is worth mentioning here that the *user* patterns may have different variants, and these variants may use different patterns. In our evaluation, we rely on the detected variant of the *user* pattern to determine what are the expected *used* patterns. For example, we noticed that in *SecBPMN2-to-UMLsec* transformations, the variant implemented for *Construction & Cleanup* does not use *Auxiliary Metamodel*. Thus, we did not report any score of that pattern for this transformation.

In summary, we can conclude that, for our transformation sample, the usage relations between patterns, depicted in of Fig. 11, are generally respected by developers. Moreover, we believe that detecting missing or incomplete usage relations may offer good refactoring opportunities to improve the transformations. In the example of the `Ecore2GenModel` transformation mentioned earlier, `Trace` was used as an approximation of `Marker` as part of *Top-down Phased Construction*, which led to a score of 50%. The transformation can be refactored by changing `Trace` to `Marker`.

### 5.5 Threats to validity

The objective of our approach is to detect model transformation design patterns. We implemented and evaluated our approach for Henshin where model transformations are specified as nested graphs. It is legitimate to question whether our approach is applicable on transformations written in other rule-based languages. To address at least partially this concern, we showed in our evaluation that some ATL transformations can be mapped to graph-like representations, on which we can apply our detection approach. Moreover, the transformations in our sample are of medium size, so it does not allow us to have a thorough time performance evaluation of our detection approach. Experiments with larger transformations are needed to draw conclusions on performance.

For the representativeness of the explored design patterns, our evaluation used most of the design patterns in the catalog of Ergin et al. [11]. We acknowledge, however, that other design patterns may exist or can be defined in the future. Moreover, considering the variety of model transformation languages, instances of patterns may take various forms, which makes it challenging to specify all these

forms as input to our approach. More specifically, some of the design patterns considered in this work depend heavily on the transformation rule engine and the way the rule execution control is performed. For instance, this is the case of *Auxiliary Metamodel* which helps using objects across rules in the context of Henshin.

Another threat to validity of our results is the way we define the approximations. Although our approach is able to detect approximations of fully specified patterns in the case of missing elements, we had to specify explicitly the approximations in our evaluation.

Finally, the fact that the recovery phase is performed manually biases somehow the evaluation of our approach accuracy during the evaluation. Potential false positives can be discarded when aligning the participants and evaluating the schedule. However, we believe that the accuracy with which our approach detects the patterns participants alleviates the burden of assembling the pattern instances. Yet, there is a burden on the manual recovery phase where we should check all combinations of the detected participants for each pattern. In the future, we plan to visualize the detected instances in the transformation directly to significantly reduce the manual workload.

## 6 Discussion

Our experience with the model transformation design pattern detection approach has raised benefits and limitations that we discuss in what follows.

### 6.1 Comparison with other techniques

The only other automated approach we are aware of that is able to detect design patterns in model transformation is our previous rule-based approach [28]. Although the rule-based approach does not need a manual recovery phase, every design patterns must be manually encoded as rule facts. This means that every variant and approximation of a design pattern must be manually encoded. Instead, our proposed string matching apporach is able to automatically detect the participants of a design pattern, its variants, and its approximations.

To compare the time performance of both approaches, we chose the three variants of the *Auxiliary Metamodel* design pattern because they do not specify a scheduling or attribute constraints, thus they do not require a manual recovery phase. We executed both approaches on the same set of model transformations presented in Table 1. Both approaches were able to detect the same variants and approximations. As shown in Fig. 16, the average detection time for each pattern in a transformation is 39 $ms$ for the string matching detection as compared to 177 $ms$ for the rule-based detection. This represents a gain of around 80% in time performance for the string matching approach, concurring the findings in [20].

### 6.2 Improving the performance

During our experiments, the detection procedure executed instantaneously, in the order of milliseconds. Nevertheless, there is still room for improvement. We identi-

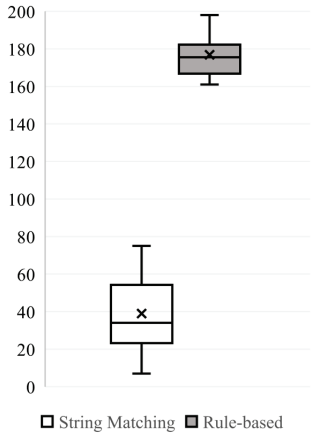



Fig. 16: Execution time for string matching vs rule-based detection

fied useful heuristics to increase the time performance of the detection mechanism. The order in which the triplets are read in Algorithm 2 influences the overall detection time. In general, treating less frequent roles first reduces the number of potential occurrences early in the detection process. For instance, a heuristic is to favor fixed roles and roles that occur less often in the Eulerian path. Also, it is preferable to avoid starting with a triplet containing a dummy edge because there are numerous occurrences of this edge in the participant and rule strings.

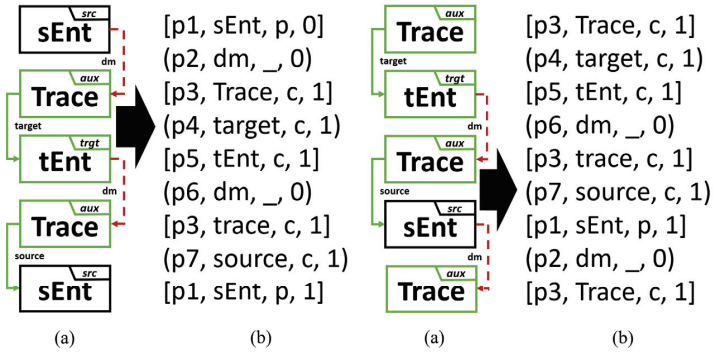



Fig. 17: Alternative string representations of the *entityMapping* participant.

Fig. 17 shows two alternative participant strings of the same participant *entityMapping*. Although they will produce the same result, the representation in (b)

will produce results in fewer iterations. That is because it starts with a fixed role (`Trace`) and the first triplet does not contain a dummy edge. Also, there are typically fewer `Trace` elements in concrete model transformation rules than elements from the input/output metamodel of the transformation.

There are different possibilities to automatically apply these heuristics in our approach. For example, once we obtain a participant or rule string, we can post-process it to apply the heuristics. Another possibility is to assign different weights to specific nodes and edges in the Eulerian graph and solve the Chinese Postman Problem using these weights.

### 6.3 Recovering the scheduling scheme

The catalog of model transformation design patterns can be partitioned into three categories: (i) design patterns not relying on a scheduling scheme, (ii) design patterns relying on a simple scheduling scheme, and (iii) design patterns heavily relying on a complex scheduling scheme.

In category (i), the design patterns contain either only one or a set of unordered participants to detect. This category comprises patterns like: *Auxiliary Metamodel*, *Entity Splitting*, and *Unique Instantiation*. For these design patterns, our approach detects their instances completely automatically. The recovery phase is not needed.

In category (ii), the complexity of the design patterns is in the roles of each participants, rather than in their scheduling. This category comprises patterns like: *Entity before Relations*, *Transitive Closure*, *Object Indexing*, *Phased construction*, *Entity Merging*, and *Construction & Cleanup*. In most of these design patterns, the participants must be ordered sequentially. Our approach detects individual participants regardless of the complexity of the structure of its roles. Nevertheless, the recovery phase simply has to check whether the order of the rules matching the participants is satisfied.

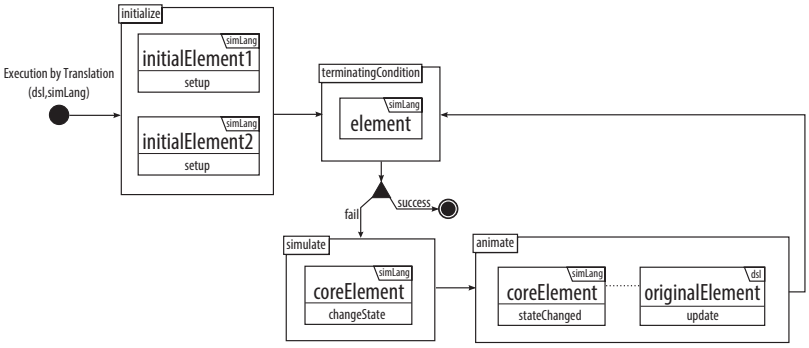



Fig. 18: The *Execution by Translation* design pattern with implicit trace elements (from [11]).

In category (iii), the complexity of the design patterns is mainly in the scheduling of the participants. This category comprises patterns like: *Fixed-point Iteration*, *Visitor*, *Simulating Universal Quantification*, and *Execution by Translation*. For these design patterns, the detection phase is very simple as we can see by the structure of the roles in the participants of Fig. 18. However, most of the workload is concentrated in the manual recovery phase. Nevertheless, this category comprises very few design patterns. Future research should therefore investigate how to automate this phase. Such an automated process should address the few scheduling scheme that design patterns rely on (conditional branching, parallelism, etc.). Furthermore, it should cope with the many different scheduling schemes offered by different model transformation languages. For example, on one hand, languages like MoTif [43] offer explicit and advanced constructs such as recursion. On the other hand, languages like ATL [18] often rely on implicit rule scheduling.

### 6.4 Detecting other components of a design pattern

Currently, we detect automatically structural features of participants, that is the graph structure of the roles. The detection of other features is left for the manual recovery phase. Consider the *Execution by Translation* pattern. According to [11], the description of this design pattern assumes the entities of a metamodel (`dsl`) are already mapped[5] to entities of another intermediate metamodel (`simLang`). Then, `simLang` is simulated and the corresponding entities in `dsl` are modified accordingly to animate the simulation. Fig. 18 presents the generic solution of the pattern. The *initialize* participant sets up the initial state of a model to start its simulation. The *terminatingCondition* participant checks if a certain condition on the model is satisfied to stop the simulation. If it is not satisfied, the *simulate* participant is activated to modify entities according to a specific criterion. The *animate* participant finds the entities in `dsl` corresponding to the entities modified in `simLang` and applies the necessary changes on them. The *terminatingCondition* participant is checked again and the simulation goes on.

The generic solution of this design pattern shows participants with one or two roles when, in practice, they could be played by a variable number of elements and relations in a model transformation implementation, depending on the application. Therefore, to reduce the false negative instances of design patterns to detect, we should not only consider the generic solution, but also look at different reification possibilities of the design pattern. In our detection approach, we can specify variants of each participant by varying the number of roles to detect most instances. Nevertheless, this also emphasizes the importance of the recovery phase that must correctly interpret the detected participants. There is important information about the design pattern that is not expressed in the generic solution. In particular, the intention of the design pattern is typically defined in natural language. Future work should investigate a procedure for capturing the intention to enhance the detection.

A participant may be played by more than one rule. For example, `initialize` in Fig. 18 can be implemented in multiple rules to initialize the state of the elements needed for the simulation. On the contrary, some participants are optional, like

[5] For example, through *Entities before Relations* or *Auxiliary Metamodel*.

`initialize`. For example, for specific application, the model transformation may not need to explicitly initialize the simulation because it relies on default values from the metamodel.

Another observation concerns the use of *tags* in design patterns expressed in DelTa. A tag expresses a condition on a role, e.g., `stateChanged`, or an action on a role, e.g., `changeState`. They are abstract components that may be implemented in three different ways in a model transformation implementation: (i) ignored, (ii) with intermediate elements, or (iii) with non-structural elements. For (i), a typical example where a tag can be ignored is the `setup` action. The implementation of this tag can be implicit in the logic of the transformation, without explicitly having an attribute of an element of a rule implementing the tag. For (ii), tags may be implemented by intermediary entities and relations. For example, `changeState` and `stateChanged` can be implemented by a marker or a link like in Fig. 14. In this example, we also see the necessity of adding a NAC in the rule. Therefore, tags in this category can be implemented by various elements depending on the semantic of the model transformation. For (iii), tags can be implemented using other types of elements, such as constraints and actions on attributes of elements in a rule. This is typically the case when a transformation heavily uses helpers expressing OCL constraints in ATL. When a design pattern falls in one of these three categories, we rely on the recovery phase to detect occurrences of the pattern.

Finally, very few design patterns, like *Unique Instantiation* and *Parallel Composition*, rely on forbidden constraints. For example, in the former pattern, if a role appears in a participant, then it cannot appear in another one. Currently, our approach is not able to detect such constraints because string matching cannot detect the absence of occurrences.

## 7 Conclusion

Detecting design patterns in model transformations can facilitate the comprehension and the maintenance of these transformation programs. In this paper, we propose a generic two-steps approach to detect design patterns, their variations, and their approximations in model transformation implementations. In the first step, our approach takes as input a set of model transformation rules and the participants of a design pattern, both converted into strings, and uses a string-matching algorithm to automatically find occurrences of pattern participants in the transformations. This automated step is completed by a manual step that consists in assembling participant occurrences and checking for the control schemes to form the pattern occurrences. The ability of our approach to detect approximations, i.e., incomplete pattern occurrences, may help to propose potential opportunities to improve the model transformation implementation. Other possibilities of refactoring opportunities can be found when examining detected occurrences of different design patterns used jointly in a transformation.

To evaluate our approach, we applied it on 18 transformations written in Henshin to detect 10 design patterns. Our evaluation results showed that design patterns are actually used in transformations, and that our approach is able to detect their different forms, being variants or approximations. They also showed that, except for a few cases, developers comply with the usage relations between patterns.

To improve our work, we plan to investigate some research directions. First, we will study other algorithms to increase the automation of the second step of the detection process. To broaden the applicability of our approach to other transformation languages, we expect to explore mapping strategies for each language to encode transformations into strings that can be handled by the matching algorithm. We also work on a recommending strategy to propose refactoring solutions to incomplete pattern occurrences, similarly to the work in [29]. Finally, to better assess the performance of our approach, we plan to test it on large transformations.

## References


1. Al-Obeidallah, M., Petridis, M., Kapetanakis, S.: A survey on design pattern detection approaches. International Journal of Software Engineering **7**, 41–59 (2016)
2. Alnusair, A., Zhao, T., Yan, G.: Rule-based detection of design patterns in program code. STTT **16**(3), 315–334 (2014). DOI 10.1007/s10009-013-0292-z. URL `https://doi.org/10.1007/s10009-013-0292-z`
3. Antoniol, G., Fiutem, R., Cristoforetti, L.: Design pattern recovery in object-oriented software. In: 6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy, pp. 153–160 (1998). DOI 10.1109/WPC.1998.693342. URL `https://doi.org/10.1109/WPC.1998.693342`
4. Arcelli, F., Perin, F., Raibulet, C., Ravani, S.: Design pattern detection in java systems: A dynamic analysis based approach. In: Evaluation of Novel Approaches to Software Engineering - 3rd and 4th International Conferences, ENASE 2008/2009, Funchal, Madeira, Portugal, May 4-7, 2008 / Milan, Italy, May 9-10, 2009. Revised Selected Papers, pp. 163–179 (2009). DOI 10.1007/978-3-642-14819-4_12. URL `https://doi.org/10.1007/978-3-642-14819-4\_12`
5. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: D.C. Petriu, N. Rouquette, Ø. Haugen (eds.) Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 6394, pp. 121–135. Springer (2010). DOI 10.1007/978-3-642-16145-2_9. URL `https://doi.org/10.1007/978-3-642-16145-2\_9`
6. Batot, E., Sahraoui, H.A., Syriani, E., Molins, P., Sboui, W.: Systematic mapping study of model transformations for concrete problems. In: S. Hammoudi, L.F. Pires, B. Selic, P. Desfray (eds.) MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016, pp. 176–183. SciTePress (2016). DOI 10.5220/0005657301760183. URL `https://doi.org/10.5220/0005657301760183`
7. Bergeron, A., Hamel, S.: Vector algorithms for approximate string matching. Int. J. Found. Comput. Sci. **13**(1), 53–66 (2002). DOI 10.1142/S0129054102000947. URL `https://doi.org/10.1142/S0129054102000947`
8. Dantizig, G.B.: Linear Programming and Extensions. Princeton University Press (1991). URL `http://www.jstor.org/stable/j.ctt1cx3tvg`
9. von Detten, M., Becker, S.: Combining clustering and pattern detection for the reengineering of component-based software systems. In: 7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011. Boulder, CO, USA, June 20-24, 2011, Proceedings, pp. 23–32 (2011). DOI 10.1145/2000259.2000265. URL `https://doi.org/10.1145/2000259.2000265`
10. Dong, J., Lad, D.S., Zhao, Y.: Dp-miner: Design pattern discovery using matrix. In: 14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007), 26-29 March 2007, Tucson, Arizona, USA, pp. 371–380 (2007). DOI 10.1109/ECBS.2007.33. URL `https://doi.org/10.1109/ECBS.2007.33`
11. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Computer Languages, Systems & Structures **46**, 106–139 (2016). DOI http://dx.doi.org/10.1016/j.cl.2016.07.004. URL `https://doi.org/10.1016/j.cl.2016.07.004`

12. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus - reverse engineering tool and schema for C++. In: 18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada, pp. 172–181 (2002). DOI 10.1109/ICSM.2002.1167764. URL https://doi.org/10.1109/ICSM.2002.1167764
13. Guéhéneuc, Y., Guyomarc'h, J., Sahraoui, H.A.: Improving design-pattern identification: a new approach and an exploratory study. Softw. Qual. J. **18**(1), 145–174 (2010). DOI 10.1007/s11219-009-9082-y. URL https://doi.org/10.1007/s11219-009-9082-y
14. Guéhéneuc, Y., Sahraoui, H.A., Zaidi, F.: Fingerprinting design patterns. In: 11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004, pp. 172–181 (2004). DOI 10.1109/WCRE.2004.21. URL https://doi.org/10.1109/WCRE.2004.21
15. Holub, J.: Simulation of NFA in approximate string and sequence matching. In: Proceedings of the Prague Stringology Club Workshop 1997, Prague, Czech Republic, July 7, 1997, pp. 39–46. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University (1997). URL http://www.stringology.org/event/1997/p5.html
16. Iacob, M., Steen, M.W.A., Heerink, L.: Reusable model transformation patterns. In: Workshops Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference, ECOCW 2008, 16 September 2008, Munich, Germany, pp. 1–10 (2008). DOI 10.1109/EDOCW.2008.51. URL https://doi.org/10.1109/EDOCW.2008.51
17. Johannes, J., Zschaler, S., Fernández, M.A., Castillo, A., Kolovos, D.S., Paige, R.F.: Abstracting complex languages through transformation and composition. In: A. Schürr, B. Selic (eds.) Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5795, pp. 546–550. Springer (2009). DOI 10.1007/978-3-642-04425-0_41. URL https://doi.org/10.1007/978-3-642-04425-0_41
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1-2), 31–39 (2008). DOI 10.1016/j.scico.2007.08.002. URL https://doi.org/10.1016/j.scico.2007.08.002
19. Jurack, S., Tietje, J.: Solving the TTC 2011 reengineering case with henshin. In: Proceedings Fifth Transformation Tool Contest, TTC 2011, Zürich, Switzerland, June 29-30 2011., pp. 181–203 (2011). DOI 10.4204/EPTCS.74.17. URL https://doi.org/10.4204/EPTCS.74.17
20. Kaczor, O., Guéhéneuc, Y., Hamel, S.: Identification of design motifs with pattern matching algorithms. Information & Software Technology **52**(2), 152–168 (2010). DOI 10.1016/j.infsof.2009.08.006. URL https://doi.org/10.1016/j.infsof.2009.08.006
21. Kramer, C., Prechelt, L.: Design recovery by automated search for structural design patterns in object-oriented software. In: Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering, pp. 208–215 (1996). DOI 10.1109/WCRE.1996.558905
22. Lano, K., Rahimi, S.K.: Model-transformation design patterns. IEEE Transactions on Software Engineering **40**(12), 1224–1259 (2014). DOI 10.1109/TSE.2014.2354344. URL https://doi.org/10.1109/TSE.2014.2354344
23. Lano, K., Rahimi, S.K., Tehrani, S.Y., Sharbaf, M.: A survey of model transformation design pattern usage. In: Theory and Practice of Model Transformation - 10th Internat. Conf, ICMT 2017, Proceedings, *LNCS*, vol. 10374, pp. 108–118. Springer (2017). DOI 10.1007/978-3-319-61473-1_8. URL https://doi.org/10.1007/978-3-319-61473-1_8
24. Lano, K., Rahimi, S.K., Tehrani, S.Y., Sharbaf, M.: A survey of model transformation design patterns in practice. Journal of Systems and Software **140**, 48–73 (2018). DOI 10.1016/j.jss.2018.03.001. URL https://doi.org/10.1016/j.jss.2018.03.001
25. Lucia, A.D., Deufemia, V., Gravino, C., Risi, M.: Design pattern recovery through visual language parsing and source code analysis. Journal of Systems and Software **82**(7), 1177–1193 (2009). DOI 10.1016/j.jss.2009.02.012. URL https://doi.org/10.1016/j.jss.2009.02.012
26. Mayvan, B.B., Rasoolzadegan, A.: Design pattern detection based on the graph theory. Knowledge-Based Systems **120**, 211–225 (2017). DOI 10.1016/j.knosys.2017.01.007. URL https://doi.org/10.1016/j.knosys.2017.01.007
27. Mhawish, M.Y., Gupta, M.: Design pattern detection using ontology (2018)
28. Mokaddem, C., Sahraoui, H., Syriani, E.: Towards rule-based detection of design patterns in model transformations. In: System Analysis and Modeling. Technology-Specific Aspects of Models - 9th International Conference, SAM 2016, Saint-Melo, France, October 3-4,

2016, Proceedings, *LNCS*, vol. 9959, pp. 211–225. Springer, Saint-Malo (2016). DOI 10.1007/978-3-319-46613-2_14. URL https://doi.org/10.1007/978-3-319-46613-2_14
29. Mokaddem, C., Sahraoui, H., Syriani, E.: Recommending model refactoring rules from refactoring examples. In: A. Wasowski, R.F. Paige, Ø. Haugen (eds.) Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018, pp. 257–266. ACM (2018). DOI 10.1145/3239372.3239406. URL https://doi.org/10.1145/3239372.3239406
30. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM **46**(3), 395–415 (1999). DOI 10.1145/316542.316550. URL https://doi.org/10.1145/316542.316550
31. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology **48**(3), 443 – 453 (1970). DOI https://doi.org/10.1016/0022-2836(70)90057-4. URL http://www.sciencedirect.com/science/article/pii/0022283670900574
32. Niere, J., Schäfer, W., Wadsack, J.P., Wendehals, L., Welsh, J.: Towards pattern-based design recovery. In: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA, pp. 338–348 (2002). DOI 10.1145/581339.581382. URL https://doi.org/10.1145/581339.581382
33. Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M.: Full contract verification for ATL using symbolic execution. Software and Systems Modeling **17**(3), 815–849 (2018). DOI 10.1007/s10270-016-0548-7. URL https://doi.org/10.1007/s10270-016-0548-7
34. Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L., Verkamo, A.: Software metrics by architectural pattern mining. Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress) (2000)
35. Pandiselvam, P., Marimuthu, T., Lawrance, R.: A comparative study on string matching algorithm of biological sequences. Computing Research Repository (CoRR) **abs/1401.7416** (2014). URL http://arxiv.org/abs/1401.7416
36. Philippow, I., Streitferdt, D., Riebisch, M., Naumann, S.: An approach for reverse engineering of design patterns. Software and Systems Modeling **4**(1), 55–70 (2005). DOI 10.1007/s10270-004-0059-9. URL https://doi.org/10.1007/s10270-004-0059-9
37. Ramadan, Q., Salnitri, M., Strüber, D., Jürjens, J., Giorgini, P.: From secure business process modeling to design-level security verification. In: 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017, pp. 123–133. IEEE Computer Society (2017). DOI 10.1109/MODELS.2017.10. URL https://doi.org/10.1109/MODELS.2017.10
38. Rasool, G., Philippow, I., Mäder, P.: Design pattern recovery based on annotations. Advances in Engineering Software **41**(4), 519–526 (2010). DOI 10.1016/j.advengsoft.2009.10.014. URL https://doi.org/10.1016/j.advengsoft.2009.10.014
39. Richa, E., Borde, E., Pautet, L.: Translation of ATL to AGT and application to a code generator for simulink. Software and Systems Modeling **18**(1), 321–344 (2019). DOI 10.1007/s10270-017-0607-8. URL https://doi.org/10.1007/s10270-017-0607-8
40. Shi, N., Olsson, R.A.: Reverse engineering of design patterns from java source code. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan, pp. 123–134 (2006). DOI 10.1109/ASE.2006.57. URL https://doi.org/10.1109/ASE.2006.57
41. Smith, T., Waterman, M.: Identification of common molecular subsequences. Journal of Molecular Biology **147**(1), 195 – 197 (1981). DOI https://doi.org/10.1016/0022-2836(81)90087-5. URL http://www.sciencedirect.com/science/article/pii/0022283681900875
42. Srinivasan, G.: Operations Research : Principles and Applications. Prentice-Hall of India (2010). URL https://books.google.ca/books?id=wYfxffB62NUC
43. Syriani, E., Vangheluwe, H.: A Modular Timed Graph Transformation Language for Simulation-based Design. Software & Systems Modeling **12**(2), 387–414 (2013). DOI 10.1007/s10270-011-0205-0. URL https://doi.org/10.1007/s10270-011-0205-0
44. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design pattern detection using similarity scoring. IEEE Trans. Software Eng. **32**(11), 896–909 (2006). DOI 10.1109/TSE.2006.112. URL https://doi.org/10.1109/TSE.2006.112
45. Uchiyama, S., Kubo, A., Washizaki, H., Fukazawa, Y.: Detecting design patterns in object-oriented program source code by using metrics and machine learning (2014). DOI 10.4236/jsea.2014.712086
46. Yu, D., Zhang, Y., Chen, Z.: A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. Journal of Systems and Software

**103**, 1–16 (2015). DOI 10.1016/j.jss.2015.01.019. URL https://doi.org/10.1016/j.jss.2015.01.019

47. Zanoni, M., Fontana, F.A., Stella, F.: On applying machine learning techniques for design pattern detection. Journal of Systems and Software **103**, 102–117 (2015). DOI 10.1016/j.jss.2015.01.037. URL https://doi.org/10.1016/j.jss.2015.01.037

# Chapter 6

## Recommending Model Refactoring Rules from Refactoring Examples [1]

Chihab eddine Mokaddem,
Houari Sahraoui and Eugene Syriani
DIRO, Université de Montréal

This article represents the work I did for the third contribution of my thesis.
The co-authors are my supervisors who contributed to the writing.

[1]The content of this chapter has been published in the 21 th ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODElS'18). Reference (Mokaddem et al, 2018)

**Résumé.** Les modèles, comme d'autres artefacts de première classe tels que le code source, sont maintenus et peuvent être refactorisés pour améliorer leur qualité et, par conséquent, l'un des artefacts dérivés. Compte tenu de la taille des modèles manipulés, un support automatique est nécessaire pour les tâches de refactoring. Lorsque les règles de refactoring sont connues, un tel support est simplement la mise en œuvre de ces règles dans les éditeurs. Cependant, pour les langages de modélisation propriétaires et moins populaires, les règles de refactoring sont généralement difficiles à définir. Néanmoins, leurs connaissances sont souvent intégrées dans des exemples pratiques. Dans cet article, nous proposons une approche pour recommander des règles de refactoring que nous apprenons automatiquement à partir d'exemples de refactoring. L'évaluation de notre approche sur trois langages de modélisation montre que, en général, les règles apprises sont exactes.

# Recommending Model Refactoring Rules from Refactoring Examples

Chihab eddine Mokaddem
Université de Montréal
Montréal, QC, Canada
mokaddec@iro.umontreal.ca

Houari Sahraoui
Université de Montréal
Montréal, QC, Canada
sahraouh@iro.umontreal.ca

Eugene Syriani
Université de Montréal
Montréal, QC, Canada
syriani@iro.umontreal.ca

## ABSTRACT

Models, like other first-class artifacts such as source code, are maintained and may be refactored to improve their quality and, consequently, one of the derived artifacts. Considering the size of the manipulated models, automatic support is necessary for refactoring tasks. When the refactoring rules are known, such a support is simply the implementation of these rules in editors. However, for less popular and proprietary modeling languages, refactoring rules are generally difficult to define. Nevertheless, their knowledge is often embedded in practical examples. In this paper, we propose an approach to recommend refactoring rules that we lean automatically from refactoring examples. The evaluation of our approach on three modeling languages shows that, in general, the learned rules are accurate.

## CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Search-based software engineering**;

## KEYWORDS

model refactoring, genetic programming, by-example approach, search-based software engineering

**ACM Reference Format:**
Chihab eddine Mokaddem, Houari Sahraoui, and Eugene Syriani. 2018. Recommending Model Refactoring Rules from Refactoring Examples. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239372.3239406

## 1 INTRODUCTION

Model-driven engineering is increasingly popular in industry [23]. In industrial contexts, the complexity of models keeps on increasing and are they are used in various development and maintenance activities. Like other first-class artifacts such as source code, they are maintained and refactored to improve their quality. Considering the size of the manipulated models, automatic support is necessary to refactor these models.

Much work has been done on model refactoring [24]. Most of the contributions can be classified into two families: (1) tools and mechanisms to define and apply refactorings [17, 22, 25, 29, 31, 37], and (2) specific refactoring rule definition [6, 11, 34]. When the languages (metamodels), in which models are expressed, are of general purpose such as UML, there is a critical mass of researchers and users that allow to shape, test, and prove refactoring rules. This is not the case for domain-specific languages (DSLs), where it is not always possible to have such a critical mass to define refactoring rules, due to the specific expertise needed. Thus knowledge is not available to feed the refactoring tools.

When fully writing refactoring rules for DSLs is difficult, if not impossible, a promising alternative is to learn them from examples. This idea was successfully investigated for learning model transformation rules from examples [1, 9]. As model refactoring can be seen as a particular use of model transformation [18, 19], one can adapt these learning algorithms for refactoring. However, this adaption is difficult for two reasons. First, refactoring is an inplace model transformation. Most of the existing by example techniques are intended to generate a completely new model. Second, not the entire source model is affected by the transformation. Only specific situations in the model constitute refactoring opportunities.

In this paper, we propose an approach to recommend refactoring rules learned from examples. Our approach can apply to different scenarios. For example, a modeler starts performing refactoring on a large model. Then, after some occurrences, she feeds in the changed model fragments (before and after the changes) into our approach. The learning process can then suggest rules that she can apply (with or without modifications) to the rest of the model. Another scenario is to collect different versions of models on which manual refactoring had been applied in the past. Then, starting from these model versions, our approach derive refactoring rules to potentially apply on new models. In these scenarios, our approach does not pretend to provide absolute correct refactoring rules. Instead, it suggests the rules that best conform to the provided examples.

We view rule learning as an optimization problem and we solve it using genetic programming. Our algorithm searches for refactoring rules that best conform to the provided examples. Examples are pairs of models (or model fragments) before and after the refactoring. Our assumption is that it is easier for domain experts to provide concrete examples of situations where a refactoring must be applied and how to apply it, than defining fully-fledged, consistent, and general refactoring rules.

To evaluate our approach, we applied it to three known metamodels for which we have the sought refactoring rules beforehand

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*MODELS '18, October 14–19, 2018, Copenhagen, Denmark*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239406

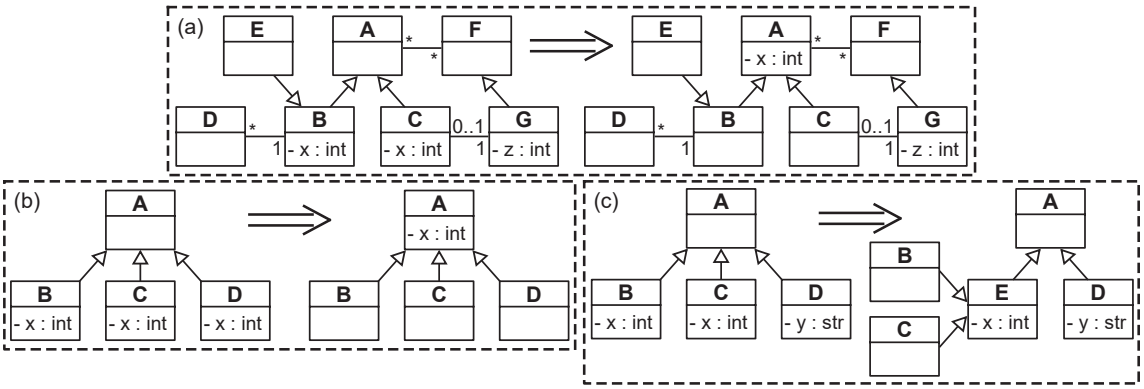



**Figure 1: Three examples to learn rules for the Pull-up field refactoring**

(ground truth).These metamodels have different kinds of refactoring with different complexities. Our results show that it is possible to learn complex refactoring rules, but the accuracy of these rules depends on the coverage of the provided examples.

In Section 2, we explain the challenges to learn refactoring transformations from examples. In Section 3, we present our approach based on genetic programming. In Section 4, we validate our solution by reporting an empirical experiment we conducted. In Section 5, we discuss the application and limitations of our approach. Finally, we review related work in Section 6 and conclude in Section 7.

## 2 CHALLENGES IN LEARNING REFACTORING TRANSFORMATIONS

### 2.1 Motivating example

Although there are well-documented refactoring patterns defined for known formalisms, e.g., UML class diagrams (UMLCD) [10], it is very difficult for a domain expert to express complete general refactoring rules for a DSL. Nevertheless, such non-software engineering experts typically provide their refactoring knowledge by means of examples. However, learning general rules from examples is not trivial, since it is very sensitive to the coverage of the examples.

Consider the *Pull-up field* refactoring pattern in UMLCD. It is usually described as *"if two subclasses have the same field, move that field to their super-class"* [10]. To learn the rule for this refactoring as a model transformation, the domain expert could provide the example illustrated in Figure 1 (a). From this example, one could easily deduce a single model transformation rule where the precondition pattern, a.k.a. left-hand side (LHS), consists of a super-class $C_1$ with two sub-classes $C_2$ and $C_3$ having both an attribute $A_1$. This pattern is a generalization of the subset of the left model with classes `A`, `B` and `C` and attributes `x`. The action part of the rule, a.k.a. the right-hand side (RHS), consists of deleting $A_1$ from $C_2$ and $C_3$, and creating an attribute $A_1$ in $C_1$.

An important challenge with learning this rule from this example is that the algorithm should detect the part that has been modified so that it correctly identifies opportunities to apply this refactoring on any input UMLCD model. Existing approaches learning model transformation from examples cannot be reused for learning refactoring rules. Some of them [1, 9] try to learn outplace transformations, where a new model is produced from another one. Others [12, 32] try to only learn the sequence of refactoring rule application given the rules and input model, from example. For a refactoring, the model transformations to learn are inplace: it is the same model that is modified [29]. For such transformations, the challenge is to detect the transformation occurrence correctly in the model, rather than focusing on the changes to perform. **This is therefore a key contribution of this article.** In the previous example, the rule must detect that two subclasses need to be present from the model pair in Figure 1 (a). That is because a counterexample is also present between classes `F` and `G`. This helps to reduce the search space for finding the correct precondition to apply a refactoring.

Another challenge is that the rule deduced from solely this example only works when there are two subclasses. If there are three or more subclasses as in Figure 1 (b), applying this rule will only remove `x` from two subclasses. A better model transformation for this refactoring would consist of two rules. A first rule duplicates an attribute in the super-class when this attribute exists in two sub-classes. A second rule removes an attribute from a sub-class when this attribute exists in the super-class. **This is in fact the transformation output by our algorithm.** Nevertheless, this revised transformation still has corner cases when it is erroneous. For example, if the model to refactor is the one in the left part of Figure 1 (c), this transformation will pull attribute `x` up to `A` even if it should not be defined on all its subclasses (i.e., `D`). In this case, the correct model transformation should pull the common attribute to a new intermediate subclass, leading to the desired refactored model illustrated on the right of Figure 1 (c).

This example illustrates that even for commonly known refactoring patterns in software engineering, manually expressing the general model transformation that implements a refactoring pattern is very difficult. It is even more difficult for domain-specific experts who are not used to think algorithmically, such as in [6].

It also shows how critical the appropriate choice of examples is to learn the right refactoring rules.

## 2.2 Problem formulation

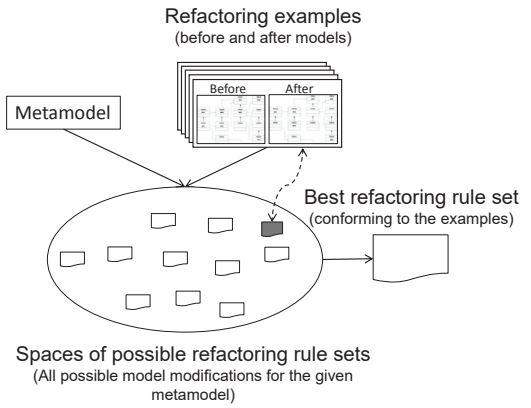



**Figure 2: Refactoring learning as an optimization problem**

Consider the situation where no explicit refactoring knowledge is available. The only available information is the DSL described by means of a metamodel and a set of examples each containing a pair of models, i.e., before and after the refactoring, as depicted in Figure 2. In this situation, learning refactoring rules can be seen a search problem in which we explore the very large space of all possible refactoring rule sets that can be written for the concerned metamodel. To guide the search, candidate rule sets are evaluated using the conformance of their behavior with the provided examples, i.e., applying the refactoring rules to the before models of the example pairs and comparing the obtained models with the corresponding after models.

# 3 LEARNING REFACTORING TRANSFORMATIONS FROM EXAMPLES

In this section, we start by presenting the basics of genetic programming. Then, we show how we adapt this algorithm to the problem of refactoring rule learning.

## 3.1 Genetic programming

Genetic programming (GP) is an evolutionary algorithm whose goal is to automatically create computer programs, from examples of inputs/outputs, to solve problems [28]. Figure 3 sketches its general process. The algorithm starts by generating an initial population of solutions (programs) of a given size. Then, it evaluates these solutions by means of a fitness function. This function generally measures the ability of a program to produce the expected outputs from the provided inputs. The next step of the algorithm is to evolve the initial population through a given number of iterations (generations) by combining three types of operations: elitism, crossover, and mutation. Thus, for each generation, a fixed number of the top solutions are automatically reproduced in the new population following the elitism principle. This principle ensures that the best solutions are not lost during the evolution. Then, to fill the remaining slots in the population, pairs of parent solutions are selected following a strategy that favors the fittest solutions, while still giving a chance to all the solutions. An example of such strategies is the roulette-wheel selection that consists in assigning a probability to each solution to be selected, proportionally to its fitness. Each selected pair of parent solutions is used, with a certain probability, to produce two child solutions, thanks to the crossover operator. The child solutions (or parent solutions if the crossover is not performed) are mutated with a given probability. When a stop condition is satisfied (generally a fixed number of generations), GP returns the best solution found.

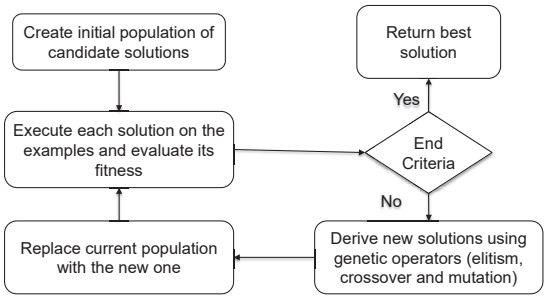



**Figure 3: Overview of the genetic programming process**

The above-described algorithm is generic and can be applied to derive any kind of program. To adapt the algorithm to a specific problem, we shall define: (1) how to encode a program (in our case a set of refactoring rules), (2) how to generate an initial population of programs, (3) how to evaluate a program, and (4) how to produce new programs from existing ones using genetic operators. The remainder of this section details these specifics to learn refactoring rules from examples.

## 3.2 Encoding refactoring rule sets

To learn the refactoring rules, we have a set of $m$ examples, where each example $i$ consists of a source model $s_i$ and a target model $e_i$. The task is to find all refactoring rules that transform any model $s_i$ into $e_i$. Therefore, a candidate solution is a set of refactoring rules, which we call *refactoring set* in the following. We denote each individual transformation by $R = \{r_1, r_2, \ldots, r_k\}$ for $k \in \mathbb{N}$, composed of $k$ refactoring rules $r$ to be executed. Note that a set of rules $R$ encodes a set of refactorings because the provided example pairs may correspond to more than one refactoring pattern application. The initial population is therefore composed of $n$ sets of refactoring rules.

```
(defrule PullUpField
  ?c1 <- (Class(name ?A))
  ?c2 <- (Class(name ?B))
  ?c3 <- (Class(name ?C))
  ?i1 <- (Inheritance(subclass ?B)(superclass ?A))
  ?i2 <- (Inheritance(subclass ?C)(superclass ?A))
```

```
7   ?a1 <- (Attribute(name ?attName)(class ?B))
8   ?a2 <- (Attribute(name ?attName)(class ?C))
9   (test (and (neq ?c1 ?c2)(neq ?c1 ?c3)(neq ?c2 ?c3)))
10  =>
11  (assert (Attribute(name ?attName)(classe ?A)))
12  (retract ?a1)
13  (retract ?a2))
```

**Listing 1: A rule encoded in the Jess languagee**

To encode a refactoring rule, we use the general purpose declarative language Jess (Java Expert System Shell) [13]. We specify a metamodel by a set of *fact templates*, by textually encoding the metamodel elements, such as classes and inheritance relationships for UMLCD. We encode models as *facts*, instances of the fact templates, e.g., `class(name Student)` and `inheritance(subclass Student)(superclass Person)`. The example given in Listing 1 shows a candidate rule for the *Pull-up field* refactoring[1]. The LHS of the rule encodes a refacotring opportunity, i.e., a pattern to search for in the source models (lines 2–9 in our example). The RHS of the rule lists the sequence of operation to perform on the LHS pattern instances (lines 11–13). The pattern to match includes three classes (lines 2–4). Two of them should inherit from the third one (lines 5–6). The two subclasses should also have an identical attribute (lines 7–8). When an instance of this pattern is found, the operation to perform are: add an attribute in the superclass with the same name as the ones of the subclasses (line 11) and remove the attribute from the subclasses (lines 12–13). We can see that this rule will correctly detect the refactoring in the example pair of Figure 1 (a).

### 3.3 Initial creation of refactoring sets

As shown in Figure 3, the first step is to create randomly an initial population of $n$ refactoring sets. Each refactoring set contains a random number of rules within a parameterized interval. The LHS of a rule is created by generating randomly a bounded number of fragments based on the metamodel types. The state conditions on the fragment are generated randomly based on the element properties as described in [9].

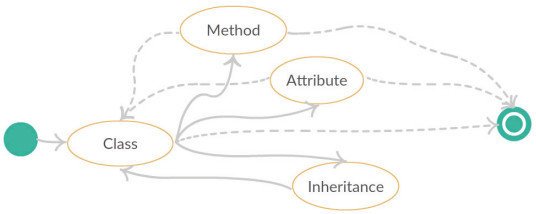



**Figure 4: FTG to create fragments of a excerpt of the Class Diagram metamodel**

To help creating consistent (strongly connected) patterns to search for, we use a strategy based on the constraints imposed by the metamodel structure. To this end, we build a fragment type graph (FTG), a sort of automaton that defines the dynamics of the pattern elements' creation. Figure 4 shows an excerpt of an FTG corresponding to a simplified UMLCD metamodel. In such a graph, nodes are either element types of the metamodel to instantiate or the start/end nodes. When traversing this graph, the current node indicates the metamodel element from which we can create the next element. These nodes are connected by two types of edges. A creation edge (depicted as a solid arrow) indicates that the target node can be created after the source node is created. Both created elements are connected according to the metamodel syntax. For example, the edge between the *Class* and *Attribute* means when a class $C1$ is created, the next step may be the creation of an attribute $A1$. $A1$ is then considered as an attribute of $C1$. The second type of edge is the back edge (dashed arrow). It indicates that from the current element, we cannot further create a metamodel element connected to it. For example, once we create the attribute $A1$, there is no further element to create that can be connected to it. The back edge outgoing from $A1$ sets $C1$ as the following element. When a node is the source of many creation edges, one of the edges is selected randomly. For example, after setting $C1$ as the current node, it is possible to create a method $M1$ for $C1$, another attribute $A2$ for $C1$ or an inheritance relationship $I1$ with $C1$ as the superclass or the subclass. In the latter case, the current instance $I1$ requires the creation of a new class $C2$ to play the role of the superclass. $C2$ becomes the current instance and so on and so forth. When a node has many back edges, then one of them is selected arbitrary. If we select the edge whose target is the end node, the pattern creation stops and this constitute the LHS of the rule.

To create the RHS of rules, we randomly select addition and deletion operators with randomly defined elements.

### 3.4 Evaluating refactoring sets

We determine how well a set of refactoring rules $R$ implements the transformation of the $m$ examples provided using a global fitness function $F$. As shown in Equation (1), it averages the fitness functions $f_i(R)$ that calculate how "close" $R$ is from producing the expected result for each example.

$$F(R) = \frac{\sum_{i=1}^{m} f_i(R)}{m} \tag{1}$$

Like other approaches that learn transformations [1, 2, 8, 9], to evaluate $f_i(R)$, we apply $R$ on the considered source model $s_i$ of the example pair $i$ to produce a model $p_i$ that we compare with $e_i$. However, because we deal with inplace transformations, we must ensure that $R$ only modifies the concerned model elements and preserves the others. To define $f_i$, let us consider the following four sets. $A$ and $D$ are the sets of elements added and deleted by $R$ respectively, when comparing $s_i$ with $p_i$. $EA$ and $ED$ are the sets of elements expected to be added and deleted respectively, when comparing $s_i$ with $e_i$. $SD$ is the set of elements that should not be deleted, when comparing $s_i$, $p_i$ and $e_i$. Then, we compute the following sets. $CA = A \cap EA$ and $CD = D \cap ED$ are the sets of elements that have been correctly added and deleted respectively. $IA = A \setminus EA$ and $ID = D \setminus ED$ are the sets of elements that have been incorrectly added and deleted respectively. To avoid bias favoring rules with higher number of elements, these sets are computed for each type $t \in T_i$, $T_i$ being the set of types present in the metamodel and instantiated in example $i$.

[1] Note that this is a simplified rule, types cardinalities and other properties are also taken into account.

The fitness function $f_i(R)$ includes for each type $t$ two components, $mod_t(R)$ and $pres_t(R)$ that calculate respectively the model modification and preservation scores when applying $R$ to $s_i$[2]:

$$f_i = \frac{1}{|T_i|} \times \sum_{t \in T_i} \alpha \times mod_t + \beta \times pres_t \quad (2)$$

$mod_t(R)$ is the proportion of elements of type $t$ added or deleted by $R$ among the expected ones. If no change is expected for a type $t$, then $mod_t(R) = 1$. Formally:

$$mod_t = \begin{cases} 1, & \text{if } |EA_t| + |ED_t| = 0 \\ \frac{|CA_t|+|CD_t|}{|EA_t|+|ED_t|}, & \text{otherwise} \end{cases} \quad (3)$$

$pres_t(R)$ is the proportion of elements in $s_{it}$ (elements of type $t$) that are not supposed to change and that are actually preserved. It is defined formally as:

$$pres_t = \frac{1}{2} \times \left( \left( 1 - \frac{|IA_t|}{|s_{it}| + |IA_t|} \right) + \left( 1 - \frac{|ID_t|}{|SD_i|} \right) \right) \quad (4)$$

### 3.5 Deriving new refactoring sets

To generate the next population of rule sets, we start by selecting a given number of the fittest ones and include them directly in the generated population. This elitism strategy allows us to preserve the best genetic material across generations. Then, to fill the remaining slots, we perform the crossover and mutation operators on the rule sets of this generation. More specifically, we select a pair of rule sets following the roulette-wheel strategy. This assigns to each rule set a probability to be selected proportionally to its fitness. When two parent rule sets are selected three equiprobable scenarios can be performed: crossover only, mutation only, or both. Whatever the chosen scenario is, the crossover and mutation operations are performed with a certain probability.

*Crossover.* The crossover operator produces two new rule sets by combining the rules of the parent rule sets. Let us consider the two rule sets $R_a$ and $R_b$. If the crossover is chosen, we apply it with a certain probability as follows. *Cut-points*, $x$ and $y$ are respectively selected to partition each set of rules into two: $R_a = \{r_{a1}, \ldots, r_{ax}, r_{ax+1}, \ldots, r_{ak}\}$ and $R_b = \{r_{b1}, \ldots, r_{by}, r_{by+1}, \ldots, r_{bl}\}$. The we recombine the partitions to produce two new rule sets: $R_{a'} = \{r_{a1}, \ldots, r_{ax}, r_{by+1}, \ldots, r_{bl}\}$ and $R_{b'} = \{r_{b1}, \ldots, r_{by}, r_{ax+1}, \ldots, r_{ak}\}$.

*Mutation.* If mutation is chosen for a given rule set, one of four mutation operators modifies the rules themselves. At the rule set level, an operator adds/removes a randomly generated/selected rule (as described in Section 3.3). At the rule level, an operator adds/removes pattern elements in the LHS, while still conforming to the FTG. Finally, two operators adds/removes modification operations in the RHS: one for assertion and one for deletion operations (see Section 3.2). To avoid any bias, all operators are selected with equal probability. However, a great part of the problem complexity is to search for the accurate refactoring opportunity before performing the refactoring. Therefore, we assign the LHS mutation operator twice as much probability as the others.

[2]To make the notation less cluttered, we omit "$(R)$" in the various components of $f_i$ equations.

## 4 VALIDATION

We evaluate our approach along the following three research questions:

**RQ1 Do the learned refactoring sets refactor models correctly?** We first verify that all the refactorings have been correctly applied, such that the discovered transformation rules produce the correct target models.

**RQ2 Do the expected refactorings appear in the learned rules?** We then validate that the transformation rules correctly implement the intended refactorings from the examples. This guarantees that we do not achieve the production of the correct target model resulting from an arbitrary combination of rules, but thanks to the correct refactoring rules.

**RQ3 Are the results obtained attributable to our approach?** Since we rely on a probabilistic evolutionary approach, we have to check if we consistently obtain similar results across executions, and if these results are better than those of the best solution found by a random exploration which considers an equal number of solutions.

### 4.1 Experiment setting

This section details the evaluation setting. This includes the considered metamodels and their respective refactorings, the examples used in the learning process, the parameters of the GP algorithm, and the method used to answer the research questions.

*4.1.1 Data collection.* To test our approach, we use three metamodels, for which the refactoring rules are known (ground truth) and there are sufficient data sources to collect examples from. For each metamodel, we collected examples as pairs of before and after models available online or in the literature. Often, the model examples include multiple refactorings. Table 1 summarizes the data used. For each metamodel, we list the refactoring types performed in the examples, the number of refactoring-type occurrences found in the examples, and number of models considered.

*UML class diagrams (UMLCD).* For UMLCD, we reverse engineered partial models from two well-known open-source Java projects: *ArgoUML* and *Xerces-J*. Since these projects have many releases available, we selected model versions before and after refactoring. Using XSLT, we implemented templates to generate sets of facts in Jess from the XML models.

From the given examples, we identified the four refactorings[3] reported in Table 1 as cataloged in [10]. It is interesting to note that some refactorings can overlap. For example, *Clean up attribute* is also part of *Pull up field*. We anticipate that their respective transformations will share a rule.

*Workflow Petri Nets (WPN).* WPN is a particular class of Petri nets with a single source place, a single sink place, and all places and transitions are on a path from the source to the sink. In [35], the authors formalize three refactorings that improve the execution of the WPN by removing redundant elements in a way that does

[3]It is not necessarily a refactoring per se, but a form of anomaly correction.

**Table 1: Selected metamodels, refactorings, and model examples**

| Metamodel | Refactoring | Description | Occurrences | Models |
|---|---|---|---|---|
| UMLCD | Pull up field | Move an attribute shared by two classes to a superclass | 3 | 5 |
| | Pull up method | Declare the method in the superclass and keep the method definition in the current class | 4 | |
| | Pull up association | Move an association shared by two classes to a superclass | 1 | |
| | Clean up attribute | Delete an attribute of a sub-class if it is also defined in the superclass | 4 | |
| WPN | Removing implicit place | Remove a place that does not change the overall marking of the net | 2 | 4 |
| | Removing EFC structures | If a set of places all have arcs to the same set of transitions, introduce an intermediate transition and place to direct the token flow from the set of places to the set of transitions | 3 | |
| | Removing TP-cross structures | If a set of transitions all have arcs to the same set of places, introduce an intermediate place and transition to direct the token flow from the set of transitions to the set of places | 1 | |
| BPMN | Pull up incoming sequence flow | If a sequence flow connects two tasks where one is in a process and the other in a sub-process, the flow should connect the task to the sub-process directly | 2 | 3 |
| | Pull up outgoing sequence flow | Similar as above, but from the sub-process to the task | 2 | |
| | Replace sequence by message flow | Replace a sequence flow between two classes in different pools by a message flow | 2 | |
| | Explicit data association | If a sequence flow connects two tasks via an artifact, connect the two tasks directly with a sequence flow and use a data association for the artifact | 2 | |
| | Symmetric modeling | Every nested opening gateway must have its corresponding closing gateway in the same order | 1 | |

not change the observable behavior of the net. The refactoring are complex to implement as the authors provide several algorithms to apply them. In [35], they also provide four examples of WPN.

*Business Process Model and Notation (BPMN).* BPMN is a widely used formalism for business analysts to create, implement, and monitor processes [26]. Several works have proposed refactoring opportunities to improve the semantics and readability of BPMN models. The work in [27] provides four good and bad practices related to task connectivity. We also considered an additional refactoring called *Symmetric modeling* from [5]. We collected three models from the OMG standard [26] to detect these refactoring opportunities.

*4.1.2 Algorithm parameters.* Like for any evolutionary algorithm, one needs to specify the parameters to run our GP-based approach. To tune these parameters, we ran our approach with different configurations on different data sets. For the validation, we retained the following configuration. We set the size of the initial populations to 30 candidate refactoring rule sets. This size is constant for all evolutions of the population. We set both probabilities of crossover and mutation to 0.9. Unlike classical genetic algorithms, having a high mutation probability is not unusual for GP algorithms (see for instance [30]). For the elitism, we injected the top 3 refactoring rule sets into the next generation.

*4.1.3 Methodology.* To answer RQ1, we compare the produced model $p_i$ with the expected model $e_i$, relying on precision and recall measures. In Equation (5), we define precision as the ratio between number of correct modifications and the total number of modifications, based on the sets defined in Section 3.

$$Precision = \frac{1}{|T_i|} \times \sum_{t \in T_i} \frac{|CA_t| + |CD_t|}{|CA_t| + |CD_t| + |IA_t| + |ID_t|} \quad (5)$$

In Equation (6), we define the recall as the ratio between number of correct modifications and total number of expected modifications, which we defined as *mod* in Equation (3).

$$Recall = \frac{1}{|T_i|} \times \sum_{t \in T_i} \frac{|CA_t| + |CD_t|}{|EA_t| + |ED_t|} \quad (6)$$

For RQ2, we focus on the quality of the refactoring rules obtained with respect to the known refactoring rules. We manually inspect the best set of refactoring rules obtained for each metamodel to determine how close they are to the expected ones. We determine if an obtained rule completely or partially matches the expected one, or if it is not expected at all. We also check if some expected rules were missing.

To answer RQ3, we perform five runs of our approach for UMLCD with 10, 000 generations. We also use five runs of random generation, each having 30 000 rule sets (equivalent to 30 solutions per generation × 10 000 generation). We generate the random rule sets according to the initial population generation procedure. Then for each run (GP and random), we select the solution with the best fitness. We compare our approach with the random generation in terms of precision and recall of the obtained rule sets.

## 4.2 Results and interpretation

*4.2.1 RQ1: Correct refactored models.* The left part of Table 2 presents the results of the fitness, precision, and recall scores that report on the quality of the refactored models. For each metamodel, we report the results of three executions as the learning process is probabilistic by nature. The three executions use exactly the same settings, including the same initial population. For UMLCD, we observe perfect results on three executions for both precision and recall when comparing the refactored models generated by the learned rules with those given in the examples.

The precision in the case of WPN is also almost perfect. Except for one execution where a slight modification was considered as incorrect, all additions and deletions of model elements were appropriate. However, the recall is relatively low (between 74% and 84%), meaning that some modifications where not performed as expected. After analyzing the obtained models, this recall score can be explained by the following considerations. The WPN metamodel contains few elements, essentially places, transitions, and arcs. Then,

**Table 2: Results of learned refactorings with GP**

| Metamodel | Run | Fitness | Precision | Recall | Rules obtained by GP | | | Expected rules |
|---|---|---|---|---|---|---|---|---|
| | | | | | Exact | Partial | Unexpected | |
| UMLCD | 1 | 100% | 100% | 100% | 3 | 1 | 0 | 4 |
| | 2 | 100% | 100% | 100% | 3 | 1 | 0 | |
| | 3 | 100% | 100% | 100% | 3 | 1 | 0 | |
| WPN | 1 | 88.7% | 98.0% | 77.2% | 1 | 3 | 0 | 3 |
| | 2 | 82.0% | 100% | 83.8% | 2 | 1 | 1 | |
| | 3 | 82.0% | 100% | 74.5% | 2 | 2 | 0 | |
| BPMN | 1 | 98.8% | 72.6% | 89.3% | 2 | 2 | 2 | 5 |
| | 2 | 98.8% | 72.6% | 89.3% | 2 | 1 | 3 | |
| | 3 | 99.4% | 81.0% | 94.8% | 3 | 2 | 1 | |

the sought refactoring are complex variations of instances of the same few element types. For example, a refactoring rule removes an unnecessary place depending on other places and transitions. To learn such a rule, the provided before- and after-refactoring examples should include enough fragments differentiating this situation from other similar situations where the refactoring is not necessary.

For the BPMN metamodel, we observe an opposite tendency, i.e., a better recall and a lower precision. The lower precision can be explained by the fact that some additions and deletions were made because the learned rules missed some conditions to have a more precise pattern to search for.

*4.2.2 RQ2: Correct refactoring rules.* Producing the correct models using the learned rules does not mean that these rules are those expected. For this research question, we investigate whether the learned rules are correct with respect to the known and expected refactorings. Note that a specific type of refactoring can be realized by means of one or more refactoring rules. Thus we will not have one-to-one comparisons between the learned and expect rules. We rather decide if a learned rule has a correct contribution to a refactoring alone or in conjunction with other rules.

The right part of Table 2 presents an overview of the rule sets obtained by our approach compared to the expected one. For the UMLCD metamodel, we were expecting four refactoring types (see Table 1). We obtained the exact rule for the *Clean up attribute*. The combinations of this rule with respectively two other rules implement exactly the *Pull up field* and *Pull up method* refactorings. We have the same situation with *Pull up association* with the exception that one condition was missing in one rule, i.e., checking that an inheritance relationship exists only for one of the two classes having the association to pull up. This missing condition did not affect the precision and recall in RQ1, because, there was no counterexample in the provided models where a missing inheritance prevents from performing the refactoring.

For the WPN metamodel, we were able to learn all three refactorings. However, the GP consistently found, for the three executions, one more rule. In one execution, this additional rule has nothing to do with the expected refactorings. In the other cases, the additional rules contribute partially to the refactorings together with the other rules. The learned rules are not trivial. Figure 5 illustrates the rule we obtained for the *Removing TP-cross structure* refactoring. It looks for two transitions, each connected to the same two places and

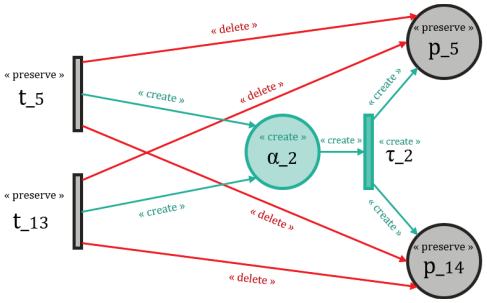



**Figure 5: Rule obtained by GP for the *Removing TP-cross structures* refactoring**

then replaces these arcs with a place and a transition to connect the initial pair of transitions to the initial pair of places.

For the BPMN metamodel, the GP also generated more rules than expected. But in this case, many of these rules were not contributing to the expected refactoring: this explains the relatively low precision score for RQ1. We accurately obtained the expected rules for two refactorings, namely *Symmetric modeling* and *Explicit data association*. For the three other refactorings, the GP was not able to derive the complete refactoring rules. As illustrated in Table 2, it only partially matched some of the rules and missed others. This situation illustrates the fact that the derived rules do not necessarily have the exact same structure (pattern elements and relations) as the rules we would have written by hand. Nevertheless, the learning process produces rules that, overall correctly perform the refactoring with respect to the provided examples.

*4.2.3 RQ3: GP vs random.* Before comparing the precision and recall of the learned refactoring rule sets, we first look at the convergence of the five GP executions. Figure 6 shows the curves corresponding to the best fitness evolution during the respective executions. These curves always converge towards a fitness score of 100% (as reported in Table 2) after a certain number of generations.

To compare with the random generation, we obtained perfect precision and recall for all the five executions of GP. As Table 3 shows, the precision is below 51% and the recall is below 56% for the

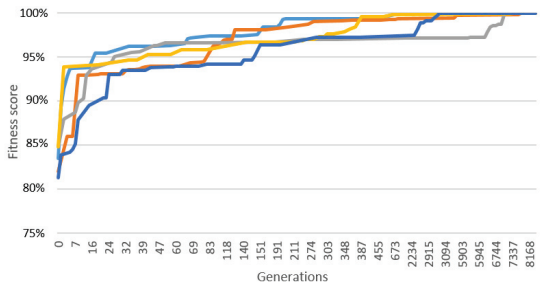



**Figure 6: Evolution of fitness score for UMLCD over five runs using our approach**

five random executions. We can conclude that the results obtained for UMLCD are attributable to our learning algorithm and not to the number of explored solutions.

**Table 3: Results of random generation for UMLCD**

| Run | Fitness | Precision | Recall |
|---|---|---|---|
| 1 | 85.3% | 50.7% | 55.4% |
| 2 | 86.6% | 26.9% | 42.1% |
| 3 | 86.3% | 45.0% | 49.8% |
| 4 | 86.3% | 42.0% | 45.0% |
| 5 | 86.6% | 48.6% | 48.8% |

### 4.3 Threats to validity

The evaluation of our approach has shown that it can learn complex refactorings in many cases. However, a threat to the validity of these results is the generalization of these results to DSLs, as mentioned in our motivations. Although experimenting with such DSLs is important, the need of having known refactorings (ground truth) and available model examples, limits our possibilities. Still, we believe that the considered metamodels and refactorings are complex enough and exhibit a good variety to represent an acceptable setting for a preliminary evaluation.

The choice of the examples is another possible threat. We selected examples that are almost prototypic. In real-life scenarios, models can be modified for different purposes, so it is not always possible to distinguish refactorings from other changes. However, our setting matches the scenario where a modeler starts performing the refactoring on some fragments and then gives the initial and modified fragments to our learning algorithm.

Another validity threat is concerned with the stop criterion for the GP algorithm. We ran our algorithm for a fixed amount of time (hours) and then checked how many generations were needed to find the best rule set. In a real-life scenario, waiting for such an amount of time is not always feasible, especially if the modeler uses our approach in an interactive way.

## 5 DISCUSSION AND LIMITATIONS

Like many other example-based approaches, the quality of the learned rules strongly depends on the quality of the examples provided. As we have shown in the BPMN case, it is utterly important to provide examples having enough variations to distinguish when different, but close, situations may lead (examples) or not (counterexamples) to a refactoring.

Related to this observation, we also noticed that the partially matched rules were under or over-constrained. Many of these situations could have been avoided had we integrated negative conditions. Our rule creation procedure generates conditions that check for the presence of elements instantiating a pattern. However, in some cases, one has to also check for the absence of elements. Integrating negative conditions in the learning process will increase the expressiveness of the learned rules.

In our current implementation, we are able to successfully detect structural anomalies in the model to be refactored, i.e., related to the presence of metamodel element instances and their connectivity. We have found cases in WPN, where the precondition of a rule requires universal quantification (e.g., *Removing TP-cross structures*). In a by-example approach, it is hard to generalize a rule from a finite sample set of examples. As in Figure 5, the GP was only able to find rules for two and three transitions, because the examples did not contain situations with more transitions. However, the general rule in [35] states that it applies to an arbitrary number of transitions. This also comes from the limitation of the model transformation paradigm that relies on existential quantification. It is, nevertheless, possible in Jess to add queries and functions in the LHS to generalize the matching. Such queries can be implemented according to navigation possibilities for a given metamodel, independently from the sought refactorings. For example, in UMLCD, a primitive query can consist of generating the set of the associations involving a given class. During the learning, this set can be used to test the absence of associations (size equals 0, for example). This approach was successfully used in [1].

Other types of refactorings are not structural by nature. For instance, in UMLCD, a pair of models example may exhibit shortening an attribute with a very long name. The intent behind this semantic anomaly correction is not only very hard to encode, but also hard to derive from a by-example approach. As this refactoring is not related to the model structure, generating the rule conditions (LHS) is not possible in the current version. To handle this, string and arithmetic operations must be used, as done in [1].

## 6 RELATED WORK

In the literature, many tools have been proposed to refactor models [17, 22, 25, 29, 31, 37]. Although they use model transformation to refactor models, they do not learn refactorings automatically. In this section, we discuss approaches that automate model refactoring and approaches that derive model transformations from examples or demonstrations.

### 6.1 Search-based model refactoring

Our approach is a search-based technique and, like very few others, is dedicated to refactoring models [21]. Most search techniques rely on evolutionary algorithms, especially genetic algorithms. In [12],

the authors also propose to generate a sequence of refactoring by measuring the similarities between provided examples. Instead, our fitness function measures explicitly correct and incorrect modifications. SORMASA [3] is a tool that assists the user by suggesting a set of model refactorings. It relies on a mono-objective evolutionary algorithm aiming at increasing cohesion and reducing coupling of UMLCD models. In [14], the author present an approach that attempts to introduce design patterns in UMLCD models by optimizing specific software design metrics.

There are, nevertheless, some approaches that consider behavior preservation. For example in [20], the authors implement a multi-objective evolutionary algorithm to optimize the trade-off between improving the quality related UML models.

### 6.2 Model transformations by demonstration

By demonstration approaches are used to derive inplace transformations automatically. In [4], the authors proposed an approach to alleviate the complexity of developing model refactoring operations. They derive semi-automatically endogenous model transformations by analyzing user editing actions when refactoring models. They collect all atomic operations performed by the user by comparing the initial and final models. The operations are then saved in a difference model from which they propose a set of pre/postconditions of the refactoring operations to the user for manual refinement. Sun et al. [33] propose a similar approach for deriving similar transformations. To collect the operations, they extend the modeling environment to monitor the user editing actions. The recorded operations are then analyzed to remove incoherent and unnecessary ones. Using an inference engine, they express user intentions as reusable transformation patterns. In [15], the authors proposed an automatic inference of inplace transformations. Their approach can infer advanced rule features, such as negative application conditions, multi-object patterns, and global invariants.

### 6.3 Model transformation by examples

Several works have investigated how to learning model transformations from examples [7, 36, 38]. However, these approaches are limited to exogenous and outplace model-to-model transformations. Learning inplace transformations from examples is more complex because we must preserve the unaffected elements of the model. Faunes et al. [8, 9] used GP learn transformation rules from examples of outplace transformations. Two articles [1, 16] tried to address more complex transformations. Although the latter did not need to learn rules, the former employed a strategy based on dividing the learning process in different steps, using adaptive mutation, and using fine-grained transformation traces in the examples.

## 7 CONCLUSION

In this paper, we present an approach to recommend refactoring rules learned from examples. The examples are pairs of models representing the versions before and after the application of refactorings. The learning is performed using GP by evolving a population of randomly generated sets of model transformation rules guided by the conformance to the provided examples. As such our approach does not pretend to find the absolute refactoring rules for a given modeling language. It instead finds the rules that best conform to the considered examples.

To evaluate our approach, we applied it on three metamodels together with refactored model examples. This evaluation showed that our approach can learn complex refactoring rules and that the obtained results are not attributable to the number of explored solution, but to our search strategy.

Although the obtained results are satisfactory, there is room for improvement. First, expressiveness of the learned rules can be enhanced by considering new constructs such as negative conditions, navigation primitives, and arithmetic and string operations. The performance of our algorithm in terms of execution time deserves to be improved. We plan to optimize the best set of rules found by the GP to improve the efficiency of the rule engine. Finally, the accuracy of our approach can be enhanced by considering more sophisticated genetic operators and search strategies. For example, adaptation mutation can be used to adapt the search strategy to the characteristics of the current population, as in [1]. This includes changing the mutation probability when no improvement is noticed for many generations or if the diversity inside the populations decreases. Another way of improving the search strategy is by experimenting with multi-objective GP algorithms. The *mod* and *pres* scores can be implemented as two different objectives without the need of defining weights to aggregate them. Minimizing the size of the rules can be another objective to avoid rules with unnecessary conditions. In the future, we also plan to test our approach on a larger set of metamodel, including DSLs, for which it is difficult to define refactoring rules.

## REFERENCES


[1] Islem Baki and Houari Sahraoui. 2016. Multi-Step Learning and Adaptive Search for Learning Complex Model Transformations from Examples. *ACM Transactions on Software Engineering Methodology* 25, 3 (2016), 20:1–20:37.

[2] Islem Baki, Houari Sahraoui, Quentin Cobbaert, Philippe Masson, and Martin Faunes. 2014. Learning Implicit and Explicit Control in Model Transformations by Example. In *Model-Driven Engineering Languages and Systems*. Springer, 636–652.

[3] Thierry Bodhuin, Gerardo Canfora, and Luigi Troiano. 2007. SORMASA: A tool for Suggesting Model Refactoring Actions by Metrics-led Genetic Algorithm. In *Workshop on Refactoring Tools in conjunction with ECOOP*. 23–24.

[4] Petra Brosch, Philip Langer, Martina Seidl, and Manuel Wimmer. 2009. Towards end-user adaptable model versioning: The by-example operation recorder. In *ICSE Workshop on Comparison and Versioning of Software Models*. IEEE Computer Society, 55–60.

[5] Camunda. 2018. BPMN Examples–Best Practices for creating BPMN 2.0 process diagrams. https://camunda.com/bpmn/examples/. (2018). (last accessed: apr 2018).

[6] J. Cunha, J. P. Fernandes, P. Martins, R. Pereira, and J. Saraiva. 2014. Refactoring Meets Model-Driven Spreadsheet Evolution. In *Quality of Information and Communications Technology*. 196–201.

[7] Xavier Dolques, Marianne Huchard, Clémentine Nebut, and Philippe Reitz. 2010. Learning Transformation Rules from Transformation Examples: An Approach Based on Relational Concept Analysis. In *Enterprise Distributed Object Computing Workshops*. 27–32.

[8] Martin Faunes, Houari Sahraoui, and Mounir Boukadoum. 2012. Generating Model Transformation Rules from Examples using an Evolutionary Algorithm. In *Automated Software Engineering*. 1–4.

[9] Martin Faunes, Houari Sahraoui, and Mounir Boukadoum. 2013. Genetic-programming approach to learn model transformation rules from examples. In *Theory and Practice of Model Transformations (LNCS)*, Vol. 7909. Springer, 17–32.

[10] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[11] A. Garrido, G. Rossi, and D. Distante. 2007. Model Refactoring in Web Applications. In *IEEE International Workshop on Web Site Evolution*. 89–96.

[12] Adnane Ghannem, Ghizlane El-Boussaidi, and Marouane Kessentini. 2014. Model refactoring using examples: a search-based approach. *Journal of Software: Evolution and Process* 26, 7 (2014), 692–713.

[13] Ernest Friedman Hill. 2003. *Jess in Action: Java Rule-Based Systems.* Manning Greenwich, CT.

[14] Adam C. Jensen and Betty H. C. Cheng. 2010. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Genetic and Evolutionary Computation Conference*. 1341–1348.

[15] Timo Kehrer, Abdullah M. Alshanqiti, and Reiko Heckel. 2017. Automatic Inference of Rule-Based Specifications of Complex In-place Model Transformations. In *Theory and Practice of Model Transformation*. 92–107.

[16] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. 2012. Search-based model transformation by example. *Software & System Modeling* 11, 2 (2012), 209–226.

[17] Yasser A. Khan and Mohamed El Attar. 2016. Using model transformation to refactor use case models based on antipatterns. *Information Systems Frontiers* 18, 1 (2016), 171–204.

[18] S. Kolahdouz Rahimi, K. Lano, S. Pillay, J. Troya, and P. Van Gorp. 2014. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming* 85 (2014), 5–40.

[19] Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M.K. Selim, Eugene Syriani, and Manuel Wimmer. 2014. Model Transformation Intents and Their Properties. *Software & Systems Modeling* 15, 3 (2014), 685–705.

[20] Usman Mansoor, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. 2017. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Software Quality Journal* 25, 2 (2017), 473–501.

[21] Thainá Mariani and Silvia Regina Vergilio. 2017. A systematic review on search-based refactoring. *Information & Software Technology* 83 (2017), 14–34.

[22] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. 2009. Generic Model Refactorings. In *Model Driven Engineering Languages and Systems (LNCS)*. Springer, 628–643.

[23] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel A Fernandez, Bjørn Nordmoen, and Mathias Fritzsche. 2013. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling* 12, 3 (2013), 619–639.

[24] Maddeh Mohamed, Mohamed Romdhani, and Khaled Ghédira. 2009. Classification of model refactoring approaches. *Journal of Object Technology* 8, 6 (2009), 121–126.

[25] Olaf Muliawan and Dirk Janssens. 2010. Model refactoring using MoTMoT. *International Journal on Software Tools for Technology Transfer* 12, 3-4 (2010), 201–209.

[26] Object Management Group 2011. *Business Process Model and Notation (BPMN)* (2 ed.). Object Management Group.

[27] Gregor Polancic. 2013. *Understanding BPMN Connections.* Technical Report. Orbus software.

[28] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. 2008. *A field guide to genetic programming.* Lulu Enterprises, UK Ltd.

[29] Ivan Porres. 2005. Rule-based update transformations and their application to model refactorings. *Software & Systems Modeling* 4, 4 (2005), 368–385.

[30] Sam Ratcliff, David Robert White, and John A. Clark. 2011. Searching for invariants using genetic programming and mutation testing. In *The Genetic and Evolutionary Computation Conference (GECCO)*.

[31] Jan Reimann, Mirko Seifert, and Uwe Aßmann. 2010. Role-Based Generic Model Refactoring. In *Model Driven Engineering Languages and Systems*. Springer, 78–92.

[32] Olaf Seng, Johannes Stammel, and David Burkhart. 2006. Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems. In *Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*. ACM, 1909–1916.

[33] Yu Sun, Jeff Gray, and Jules White. 2011. MT-Scribe: an end-user approach to automate software model evolution. In *International Conference on Software Engineering*. ACM, 980–982.

[34] Mohammad Tanhaei, Jafar Habibi, and Seyed-Hassan Mirian Hosseinabadi. 2016. Automating feature model refactoring: A Model transformation approach. *Information and Software Technology* 80 (2016), 138–157.

[35] Ichiro Toyoshima, Shingo Yamaguchi, and Jia Zhang. 2015. A Refactoring Algorithm of Workflows Based on Petri Nets. In *International Congress on Advanced Applied Informatics*. IEEE, 79–84.

[36] Daniel Varró. 2006. Model Transformation by Example. In *Model Driven Engineering Languages and Systems*. Springer, 410–424.

[37] I. Verebi. 2015. A model-based approach to software refactoring. In *International Conference on Software Maintenance and Evolution*. 606–609.

[38] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. 2007. Towards Model Transformation Generation By-Example. In *Annual Hawaii International Conference on System Sciences*. 285–295.

# Chapter 7

## Conclusion

Refactoring is a maintenance process that aims to improve the quality of software. *The thesis aims at determining to which extent refactoring be automated in MDE.* We considered two scenarios that depend on the availability of the refactoring knowledge. The first comes when the knowledge is explicitly available. The second when it is not. Hence the knowledge is implicit.

### 1. Contributions

We summarize our three contributions following the two situations of refactoring that we identified.

#### 1.1. Knowledge-based refactoring

We exploited design patterns as explicit knowledge that helps for refactoring automation. Detecting design patterns opens different refactoring opportunities. Hence, we proposed two techniques to facilitate design pattern detection in model transformations. We chose model transformation as a target because of the lack of existing detection approaches on this artifact. The detection aims at determining to which extent instances of a design pattern are implemented in a transformation. We detected two levels of implementation of design patterns: instances completely implementing a design pattern and instances partially implementing a design pattern. Our techniques allowed detecting partial and complete design pattern instances together with their variants. The first technique uses a set of rules to specify the detection of instances of a given design pattern. The proposed approach first detects the pattern participants through the rules. Then, it ensures that the control flow over these rules corresponds to the scheduling scheme in the design pattern. The second

contribution uses a string matching technique for the detection. Model transformations and design pattern participants are mapped to sequences of elements represented as strings. The proposed approach uses a generic technique that is based on a bit-vector algorithm. The advantage of the second approach, compared to the rule-based detection, is that it does not require to enumerate the specification of all possible variations or partial versions of the design pattern. Both contributions are validated on transformations expressed in the Henshin language. The results showed the ability of both contributions to detect different design patterns, their variants, and their approximations.

### 1.2. Example-based refactoring

When refactoring knowledge is not explicitly available, a.k.a. implicit knowledge, we proposed to infer it from refactoring examples. In this context, we defined an approach to learn refactoring rules from examples. We use an approach based on genetic programming to perform the learning. The approach learns executable rules that do not violate the conformance to the metamodel, for different types of models. We considered three metamodels to validate this contribution. The validation showed that our contribution can learn complex refactoring rules and that the obtained results are not attributable to the number of explored solutions, but to our research strategy. The obtained results provide compelling evidence that it is possible to automate refactoring, even when knowledge is not available, by exploiting examples.

The contributions confirm that it is possible to automate refactoring in MDE when refactoring knowledge is explicit or implicit. The design pattern detection contributions showed how refactoring opportunities can be detected and how the patterns can be used as a refactoring solution. The learning-based contribution showed how refactoring knowledge can be inferred from refactoring examples. This contribution searches for refactoring opportunities and treats them in the same set of rules that represent the extracted explicit knowledge.

## 2. Limits and future work

In this section, we discuss the limitations of our work and open research directions related to the frame of our thesis.

### 2.1. With respect to the problem space

Our vision of refactoring in MDE involves three dimensions of variability: the availability of knowledge and its nature, the artifacts to refactor, and the languages in which the artifacts are expressed. For each dimension, we only considered in this thesis a few possibilities.

For the availability of refactoring knowledge, we considered the case where refactoring knowledge exists in the form of design patterns. We did not study other forms of knowledge that can be exploited to automate refactoring. In the future, we plan to explore other alternatives, such as design smells or quality-based heuristics. Similarly, when the knowledge is not explicitly available, we studied the use of input/output examples as a substitute. Here, again, other sources of data can be exploited, such as modification histories or other artifact instances from which we can infer templates.

Regarding the types of artifacts to refactor, we only considered models and model transformations. In the future, we plan to investigate the refactoring of other artifacts such as metamodels, modeling constraints, and concrete syntax.

Finally, for the representation languages, although the refactoring learning for models is agnostic with respect to the model representations, we made the choice to consider only Henshin transformations in our studies of knowledge-based refactoring. Our approach is generalizable to graph-based model transformations, such MoTif (Syriani and Vangheluwe, 2013), GrGen.NET (Geiß and Kroll, 2007), eMoflon (Anjorin et al, 2011). However, other model transformation approaches, such as ATL (Jouault et al, 2008) and QVT (Kurtev, 2007) require adapting the pattern detection to include, for example helpers in the case of ATL.

### 2.2. With respect to the solution space

In this thesis, we used three techniques to implement the proposed contributions.

For the first contribution of design-pattern detection, the transformations are represented as sets of facts in the Jess language, and transformation design patterns as sets of rules executable by the Jess engine (Hill, 2003). We used Jess because it is a convenient language to build research prototypes. However, this language is not common in the MDE community, which makes it difficult to integrate our prototype with other tools in the MDE pipeline, such as the Eclipse Modeling Framework(Steinberg et al, 2008). To address this limitation, we

plan to adapt our approach to well-known model transformation languages, such as Henshin or ATL.

In the second contribution, we represented both transformations and transformation design patterns as strings. We used a string matching technique that is based on a bit-vector algorithm. This technique has the advantages to be efficient and generalizable. However, not all of the variants and approximations can be found with the common form of the pattern specification. In the future, we plan to automatically generate specifications of the most probable variants and approximations from the unique specification of the pattern. Moreover, we will also explore other detection techniques that reduce the amount of input information that should be provided.

For the third contribution, we rely on a genetic programming technique to learn rules from examples. It would be interesting to explore other techniques, such as classical and emerging machine learning algorithms, to learn more generalizable refactoring rules.

# Bibliography


[van der Aalst and ter Hofstede 2005] Aalst, Wil M. P. van der ; Hofstede, Arthur H. M. ter: YAWL: yet another workflow language. In: *Inf. Syst.* 30 (2005), Nr. 4, p. 245–275. – URL https://doi.org/10.1016/j.is.2004.02.002

[Agrawal et al 2005] Agrawal, Aditya ; Vizhanyo, Attila ; Kalmar, Zsolt ; Shi, Feng ; Narayanan, Anantha ; Karsai, Gabor: Reusable Idioms and Patterns in Graph Transformation Languages. In: *Electr. Notes Theor. Comput. Sci.* 127 (2005), Nr. 1, p. 181–192. – URL https://doi.org/10.1016/j.entcs.2004.12.035

[Akiyama et al 2011] Akiyama, Motohiro ; Hayashi, Shinpei ; Kobayashi, Takashi ; Saeki, Motoshi: Supporting Design Model Refactoring for Improving Class Responsibility Assignment. In: *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, URL https://doi.org/10.1007/978-3-642-24485-8_33, 2011, p. 455–469

[Al-Dallal 2015] Al-Dallal, Jehad: Identifying refactoring opportunities in object-oriented code: A systematic literature review. In: *Inf. Softw. Technol.* 58 (2015), p. 231–249. – URL https://doi.org/10.1016/j.infsof.2014.08.002

[Al-Jamimi and Ahmed 2017] Al-Jamimi, Hamdi A. ; Ahmed, Moataz A.: Model Driven Development Transformations using Inductive Logic Programming. In: *International Journal of Advanced Computer Science and Applications* 8 (2017), Nr. 11. – URL http://dx.doi.org/10.14569/IJACSA.2017.081166

[Al-Obeidallah et al 2016a] Al-Obeidallah, Mohammed ; Petridis, Miltos ; Kapetanakis, Stelios: A Survey on Design Pattern Detection Approaches. In: *International Journal of Software Engineering* 7 (2016), 12, p. 41–59

[Al-Obeidallah et al 2018] Al-Obeidallah, Mohammed ; Petridis, Miltos ; Kapetanakis, Stelios: A Multiple Phases Approach for Design Patterns Recovery Based on Structural and Method Signature Features. In: *IJSI* 6 (2018), Nr. 3, p. 36–52. – URL https://doi.org/10.4018/IJSI.2018070103

[Al-Obeidallah et al 2016b] Al-Obeidallah, Mohammed G. ; Petridis, Miltos ; Kapetanakis, Stelios: A Survey on Design Pattern Detection Approaches. In: *International Journal of Software Engineering (IJSE)* 7 (2016)

[Alkhazi et al 2020] Alkhazi, Bader ; Abid, Chaima ; Kessentini, Marouane ; Wimmer, Manuel: On the value of quality attributes for refactoring ATL model transformations: A multi-objective approach. In: *Inf. Softw. Technol.* 120 (2020), p. 106243. – URL https://doi.org/10.1016/j.infsof.2019.106243

[Alkhazi et al 2016] Alkhazi, Bader ; Ruas, Terry ; Kessentini, Marouane ; Wimmer, Manuel ; Grosky, William I.: Automated refactoring of ATL model transformations: a search-based approach. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, URL http://dl.acm.org/citation.cfm?id=2976782, 2016, p. 295–304

[Alnusair et al 2014] Alnusair, Awny ; Zhao, Tian ; Yan, Gongjun: Rule-based detection of design patterns in program code. In: *Int. J. Softw. Tools Technol. Transf.* 16 (2014), Nr. 3, p. 315–334. – URL https://doi.org/10.1007/s10009-013-0292-z

[Ammar and Bhiri 2015] Ammar, Boulbaba B. ; Bhiri, Mohamed T.: Pattern-based model refactoring for the introduction association relationship. In: *Journal of King Saud University-Computer and Information Sciences* 27 (2015), Nr. 2, p. 170–180

[Anjorin et al 2011] Anjorin, Anthony ; Lauder, Marius ; Patzina, Sven ; Schürr, Andy: Emoflon: leveraging EMF and professional CASE tools. In: Heiss, Hans-Ulrich (Editor) ; Pepper, Peter (Editor) ; Schlingloff, Holger (Editor) ; Schneider, Jörg (Editor): *41. Jahrestagung der Gesellschaft für Informatik, Informatik schafft Communities, INFORMATIK 2011, Berlin, Germany, October 4-7, 2011, Abstract Proceedings* Volume P-192, GI, 2011, p. 281. – URL https://dl.gi.de/20.500.12116/18638

[Antoniol et al 1998] Antoniol, Giuliano ; Fiutem, Roberto ; Cristoforetti, L.: Design Pattern Recovery in Object-Oriented Software. In: *6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy*, URL

https://doi.org/10.1109/WPC.1998.693342, 1998, p. 153–160

[Arcelli and Pompeo 2017] Arcelli, Davide ; Pompeo, Daniele D.: Applying Design Patterns to Remove Software Performance Antipatterns: A Preliminary Approach. In: *Procedia Computer Science* 109 (2017), p. 521 – 528. – URL http://www.sciencedirect.com/science/article/pii/S1877050917309948. – 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal. – ISSN 1877-0509

[Arcelli et al 2009] Arcelli, Francesca ; Perin, Fabrizio ; Raibulet, Claudia ; Ravani, Stefano: Design Pattern Detection in Java Systems: A Dynamic Analysis Based Approach. In: *Evaluation of Novel Approaches to Software Engineering - 3rd and 4th International Conferences, ENASE 2008/2009, Funchal, Madeira, Portugal, May 4-7, 2008 / Milan, Italy, May 9-10, 2009. Revised Selected Papers*, URL https://doi.org/10.1007/978-3-642-14819-4_12, 2009, p. 163–179

[Astels 2002] Astels, Dave: Refactoring With UML. In: *Proceedings of 3rd International Conference eXtreme Programming and Flexible Processes in Software Engineering*, 2002, p. 67–70

[Atwood 2006] Atwood, Dan: BPM process patterns: Repeatable design for BPM process models. (2006)

[Baki and Sahraoui 2016] Baki, Islem ; Sahraoui, Houari: Multi-Step Learning and Adaptive Search for Learning Complex Model Transformations from Examples. In: *ACM Transactions on Software Engineering Methodology* 25 (2016), Nr. 3, p. 20:1–20:37

[Baki et al 2014] Baki, Islem ; Sahraoui, Houari ; Cobbaert, Quentin ; Masson, Philippe ; Faunes, Martin: Learning Implicit and Explicit Control in Model Transformations by Example. In: *Model-Driven Engineering Languages and Systems*, Springer, 2014, p. 636–652

[Ballis et al 2008] Ballis, Demis ; Baruzzo, Andrea ; Comini, Marco: A Rule-based Method to Match Software Patterns Against UML Models. In: *Electr. Notes Theor. Comput. Sci.* 219 (2008), p. 51–66. – URL https://doi.org/10.1016/j.entcs.2008.10.034

[Balogh and Varró 2009] Balogh, Zoltán ; Varró, Dániel: Model transformation by example using inductive logic programming. In: *Internation Journal on Software and Systems Modeling* 8 (2009), Nr. 3, p. 347–364

[Biermann et al 2006] Biermann, Enrico ; Ehrig, Karsten ; Köhler, Christian ; Kuhns, Günter ; Taentzer, Gabriele ; Weiss, Eduard: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, URL https://doi.org/10.1007/11880240_30, 2006, p. 425–439

[Bodhuin et al 2007] Bodhuin, Thierry ; Canfora, Gerardo ; Troiano, Luigi: SORMASA: A tool for Suggesting Model Refactoring Actions by Metrics-led Genetic Algorithm. In: *Workshop on Refactoring Tools in conjunction with ECOOP*, 2007, p. 23–24

[Boger et al 2002] Boger, Marko ; Sturm, Thorsten ; Fragemann, Per: Refactoring Browser for UML. In: *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers*, URL https://doi.org/10.1007/3-540-36557-5_26, 2002, p. 366–377

[Bouhours et al 2009] Bouhours, Cédric ; Leblanc, Hervé ; Percebois, Christian: Bad Smells in Design and Design Patterns. In: *Journal of Object Technology* 8 (2009), Nr. 3, p. 43–63. – URL https://doi.org/10.5381/jot.2009.8.3.c5

[Brambilla et al 2011] Brambilla, Marco ; Fraternali, Piero ; Vaca, Carmen: BPMN and Design Patterns for Engineering Social BPM Solutions. In: Daniel, Florian (Editor) ; Barkaoui, Kamel (Editor) ; Dustdar, Schahram (Editor): *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I* Volume 99, Springer, 2011, p. 219–230. – URL https://doi.org/10.1007/978-3-642-28108-2_22

[Braun and Esswein 2014] Braun, Richard ; Esswein, Werner: Classification of Domain-Specific BPMN Extensions. In: Frank, Ulrich (Editor) ; Loucopoulos, Pericles (Editor) ; Pastor, Oscar (Editor) ; Petrounias, Ilias (Editor): *The Practice of Enterprise Modeling - 7th IFIP WG 8.1 Working Conference, PoEM 2014, Manchester, UK, November 12-13, 2014. Proceedings* Volume 197, Springer, 2014, p. 42–57. – URL https://doi.org/10.1007/978-3-662-45501-2_4

[Brown 1996] Brown, Kyle: Design Reverse-engineering and Automated Design-pattern Detection in Smalltalk. Raleigh, NC, USA : North Carolina State University at Raleigh, 1996. – Research Report

[Brown et al 1998] Brown, W. J. ; Malveau, R. C. ; Mccormick, H. W. ; Mowbray, T. J.: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, 1998

[Bucchiarone et al 2020] Bucchiarone, Antonio ; Cabot, Jordi ; Paige, Richard F. ; Pierantonio, Alfonso: Grand challenges in model-driven engineering: an analysis of the state of the research. In: *Software and Systems Modeling* 19 (2020), Nr. 1, p. 5–13. – URL `https://doi.org/10.1007/s10270-019-00773-6`

[Cinnéide 2001] Cinnéide, Mel O.: *Automated application of design patterns: a refactoring approach*, Trinity College Dublin, Ph.D. thesis, 2001

[Dao et al 2006] Dao, Michel ; Huchard, Marianne ; Hacene, Mohamed R. ; Roume, Cyril ; Valtchev, Petko: Towards Practical Tools for Mining Abstractions in UML Models. In: *ICEIS 2006 - Proceedings of the Eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration, Paphos, Cyprus, May 23-27, 2006*, 2006, p. 276–283

[De Lucia et al 2010] De Lucia, Andrea ; Deufemia, Vincenzo ; Gravino, Carmine ; Risi, Michele: Improving behavioral design pattern detection through model checking. In: *European Conference on Software Maintenance and Reengineering*, 2010, p. 176–185

[Deguil 2008] Deguil, Romain: *Mapping entre un référentiel d'exigences et un modèle de maturité: application à l'industrie pharmaceutique*, INPT, Ph.D. thesis, 2008

[Dobrzanski and Kuzniarz 2006] Dobrzanski, Lukasz ; Kuzniarz, Ludwik: An approach to refactoring of executable UML models. In: *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, URL `http://doi.acm.org/10.1145/1141277.1141574`, 2006, p. 1273–1279

[Dolques et al 2010] Dolques, Xavier ; Huchard, Marianne ; Nebut, Clémentine ; Reitz, Philippe: Learning Transformation Rules from Transformation Examples: An Approach Based on Relational Concept Analysis. In: *Enterprise Distributed Object Computing Workshops*, 2010, p. 27–32

[Dong et al 2008] Dong, Jing ; Sun, Yongtao ; Zhao, Yajing: Design pattern detection by template matching. In: *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, URL http://doi.acm.org/10.1145/1363686.1363864, 2008, p. 765–769

[Dong et al 2009] Dong, Jing ; Zhao, Yajing ; Peng, Tu: A Review of Design Pattern Mining Techniques. In: *Int. J. Softw. Eng. Knowl. Eng.* 19 (2009), Nr. 6, p. 823–855. – URL https://doi.org/10.1142/S021819400900443X

[El-Attar and Miller 2010] El-Attar, Mohamed ; Miller, James: Improving the quality of use case models using antipatterns. In: *Software and System Modeling* 9 (2010), Nr. 2, p. 141–160. – URL https://doi.org/10.1007/s10270-009-0112-9

[Ergin and Syriani 2013] Ergin, Hüseyin ; Syriani, Eugene: Identification and Application of a Model Transformation Design Pattern. In: *ACM Southeast Conference*, ACM, 2013 (ACMSE'13)

[Ergin and Syriani 2014] Ergin, Hüseyin ; Syriani, Eugene: Towards a Language for Graph-Based Model Transformation Design Patterns. In: *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, URL https://doi.org/10.1007/978-3-319-08789-4_7, 2014, p. 91–105

[Ergin et al 2016] Ergin, Hüseyin ; Syriani, Eugene ; Gray, Jeff: Design pattern oriented development of model transformations. In: *Computer Languages, Systems & Structures* 46 (2016), p. 106–139. – URL https://doi.org/10.1016/j.cl.2016.07.004

[Faunes et al 2013] Faunes, Martin ; Sahraoui, Houari ; Boukadoum, Mounir: Genetic-programming approach to learn model transformation rules from examples. In: *Theory and Practice of Model Transformations* Volume 7909, Springer, 2013, p. 17–32

[Fowler 1999a] Fowler, Martin: *Refactoring - Improving the Design of Existing Code.* Addison-Wesley, 1999 (Addison Wesley object technology series). – URL http://martinfowler.com/books/refactoring.html. – ISBN 978-0-201-48567-7

[Fowler 1999b] Fowler, Martin: *Refactoring - Improving the Design of Existing Code.* Addison-Wesley, 1999 (Addison Wesley object technology series). – URL http://martinfowler.com/books/refactoring.html. – ISBN 978-0-201-48567-7

[France and Rumpe 2007] France, R. ; Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: *Future of Software Engineering - FOSE '07*, 2007, p. 37–54

[France et al 2003] France, Robert B. ; Ghosh, Sudipto ; Song, Eunjee ; Kim, Dae-Kyoo: A Metamodeling Approach to Pattern-Based Model Refactoring. In: *IEEE Software* 20 (2003), Nr. 5, p. 52–58. – URL https://doi.org/10.1109/MS.2003.1231152

[Gamma et al 1994] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994

[García-Magariño et al 2009] García-Magariño, Iván ; Rougemaille, Sylvain ; Fuentes-Fernández, Rubén ; Migeon, Frédéric ; Gleizes, Marie P. ; Gómez-Sanz, Jorge J.: A Tool for Generating Model Transformations By-Example in Multi-Agent Systems. In: *7th International Conference on Practical Applications of Agents and Multi-Agent Systems, PAAMS 2009, Salamanca, Spain, 25-27 March 2009*, URL https://doi.org/10.1007/978-3-642-00487-2_8, 2009, p. 70–79

[Geiß and Kroll 2007] Geiss, Rubino ; Kroll, Moritz: GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In: Schürr, Andy (Editor) ; Nagl, Manfred (Editor) ; Zündorf, Albert (Editor): *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers* Volume 5088, Springer, 2007, p. 568–569. – URL https://doi.org/10.1007/978-3-540-89020-1_38

[Guéhéneuc et al 2010] Guéhéneuc, Yann-Gaël ; Guyomarc'h, Jean-Yves ; Sahraoui, Houari: Improving design-pattern identification: a new approach and an exploratory study. In: *Software Quality Journal* 18 (2010), Nr. 1, p. 145–174

[Guéhéneuc et al 2004] Guéhéneuc, Yann-Gaël ; Sahraoui, Houari A. ; Zaidi, Farouk: Fingerprinting Design Patterns. In: *11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004*, URL https://doi.org/10.1109/WCRE.2004.21, 2004, p. 172–181

[Harel and Rumpe 2004] Harel, David ; Rumpe, Bernhard: Meaningful Modeling: What's the Semantics of "Semantics"? In: *Computer* 37 (2004), Nr. 10, p. 64–72. – URL https://doi.org/10.1109/MC.2004.172

[Hegedűs et al 2012] Hegedűs, Péter ; Bán, Dénes ; Ferenc, Rudolf ; Gyimóthy, Tibor: Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability. In: Kim, Tai-hoon (Editor) ; Ramos, Carlos (Editor) ; Kim, Haeng-kon (Editor) ; Kiumi, Akingbehin (Editor) ; Mohammed, Sabah (Editor) ; Ślęzak, Dominik (Editor): *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 138–145. – ISBN 978-3-642-35267-6

[Hill 2003] Hill, Ernest F.: *Jess in Action: Java Rule-Based Systems*. Manning Greenwich, CT, 2003. – ISBN 1930110898

[Iacob et al 2008] Iacob, Maria-Eugenia ; Steen, Maarten W. A. ; Heerink, Lex: Reusable Model Transformation Patterns. In: *Workshops Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference, ECOCW 2008, 16 September 2008, Munich, Germany*, URL `https://doi.org/10.1109/EDOCW.2008.51`, 2008, p. 1–10

[Jensen and Cheng 2010] Jensen, Adam C. ; Cheng, Betty H. C.: On the use of genetic programming for automated refactoring and the introduction of design patterns. In: *Genetic and Evolutionary Computation Conference*, 2010, p. 1341–1348

[Jouault et al 2008] Jouault, Frédéric ; Allilaire, Freddy ; Bézivin, Jean ; Kurtev, Ivan: ATL: A model transformation tool. In: *Sci. Comput. Program.* 72 (2008), Nr. 1-2, p. 31–39. – URL `https://doi.org/10.1016/j.scico.2007.08.002`

[Kaczor et al 2010] Kaczor, Olivier ; Guéhéneuc, Yann-Gaël ; Hamel, Sylvie: Identification of design motifs with pattern matching algorithms. In: *Information & Software Technology* 52 (2010), Nr. 2, p. 152–168. – URL `https://doi.org/10.1016/j.infsof.2009.08.006`

[Kehrer et al 2017] Kehrer, Timo ; Alshanqiti, Abdullah M. ; Heckel, Reiko: Automatic Inference of Rule-Based Specifications of Complex In-place Model Transformations. In: *Theory and Practice of Model Transformation*, 2017, p. 92–107

[Kelly and Tolvanen 2008] Kelly, Steven ; Tolvanen, Juha-Pekka: *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008. – URL `http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html`. – ISBN 978-0-470-03666-2

[Kerievsky 2005] Kerievsky, J.: *Refactoring to Patterns*. Addison-Wesley, 2005 (A Martin Fowler signature book). – URL https://books.google.ca/books?id=ebBQAAAAMAAJ. – ISBN 9780321213358

[Kessentini et al 2011] Kessentini, M. ; Kessentini, W. ; Sahraoui, H. ; Boukadoum, M. ; Ouni, A.: Design Defects Detection and Correction by Example. In: *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2011, p. 81–90

[Kessentini et al 2008] Kessentini, Marouane ; Sahraoui, Houari A. ; Boukadoum, Mounir: Model Transformation as an Optimization Problem. In: *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, URL https://doi.org/10.1007/978-3-540-87875-9_12, 2008, p. 159–173

[Kessentini et al 2010] Kessentini, Marouane ; Wimmer, Manuel ; Sahraoui, Houari A. ; Boukadoum, Mounir: Generating transformation rules from examples for behavioral models. In: *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, Paris, France, June 14, 2010*, URL http://doi.acm.org/10.1145/1811147.1811149, 2010, p. 2

[Khan and El-Attar 2016] Khan, Yasser A. ; El-Attar, Mohamed: Using model transformation to refactor use case models based on antipatterns. In: *Information Systems Frontiers* 18 (2016), Nr. 1, p. 171–204. – URL https://doi.org/10.1007/s10796-014-9528-z

[Kim 2008] Kim, Dae-Kyoo: Software Quality Improvement via Pattern-Based Model Refactoring. In: *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008, Nanjing, China, December 3 - 5, 2008*, URL https://doi.org/10.1109/HASE.2008.10, 2008, p. 293–302

[Koschke 2006] Koschke, Rainer: Survey of Research on Software Clones. In: Koschke, Rainer (Editor) ; Merlo, Ettore (Editor) ; Walenstein, Andrew (Editor): *Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006* Volume 06301, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. – URL http://drops.dagstuhl.de/opus/volltexte/2007/962

[Kramer and Prechelt 1996] Kramer, C. ; Prechelt, L.: Design recovery by automated search for structural design patterns in object-oriented software. In: *Proceedings of WCRE*

*'96: 4rd Working Conference on Reverse Engineering*, Nov 1996, p. 208–215

[Kühne 2006] Kühne, Thomas: Matters of (Meta-)Modeling. In: *Software and Systems Modeling* 5 (2006), Nr. 4, p. 369–385. – URL https://doi.org/10.1007/s10270-006-0017-9

[Kurtev 2007] Kurtev, Ivan: State of the Art of QVT: A Model Transformation Language Standard. In: Schürr, Andy (Editor) ; Nagl, Manfred (Editor) ; Zündorf, Albert (Editor): *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers* Volume 5088, Springer, 2007, p. 377–393. – URL https://doi.org/10.1007/978-3-540-89020-1_26

[Langdon et al 2008] Langdon, William B. ; Poli, Riccardo ; McPhee, Nicholas F. ; Koza, John R.: Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications. In: *Computational Intelligence: A Compendium* Volume 115. Springer, 2008, p. 927–1028

[Langer et al 2010] Langer, Philip ; Wimmer, Manuel ; Kappel, Gerti: Model-to-model Transformations by Demonstration. In: *Proceedings of the International Conference on Model Transformation (ICMT)*, 2010, p. 153–167

[Lano and Kolahdouz-Rahimi 2014] Lano, Kevin ; Kolahdouz-Rahimi, Shekoufeh: Model Transformation Design Patterns. In: *IEEE Trans. Software Eng.* 40 (2014), Nr. 12, p. 1224–1259. – URL https://doi.org/10.1109/TSE.2014.2354344

[Lano et al 2018] Lano, Kevin ; Kolahdouz-Rahimi, Shekoufeh ; Tehrani, Sobhan Y. ; Sharbaf, Mohammadreza: A survey of model transformation design patterns in practice. In: *Journal of Systems and Software* 140 (2018), p. 48–73. – URL https://doi.org/10.1016/j.jss.2018.03.001

[Lehman 1980] Lehman, Meir M.: On understanding laws, evolution, and conservation in the large-program life cycle. In: *Journal of Systems and Software* 1 (1980), p. 213–221. – URL https://doi.org/10.1016/0164-1212(79)90022-0

[Levendovszky et al 2002] Levendovszky, Tihamer ; Karsai, Gabor ; Maroti, Miklos ; Lédeczi, Ákos ; Charaf, Hassan: Model Reuse with Metamodel-Based Transformations. In: *Software Reuse: Methods, Techniques, and Tools, 7th International Conference, ICSR-7, Austin, TX, USA, April 15-19, 2002, Proceedings*, URL https://doi.org/10.1007/

3-540-46020-9_12, 2002, p. 166–178

[Levendovszky et al 2009] Levendovszky, Tihamer ; Lengyel, László ; Mészáros, Tamás: Supporting domain-specific model patterns with metamodeling. In: *Software and System Modeling* 8 (2009), Nr. 4, p. 501–520. – URL https://doi.org/10.1007/s10270-009-0118-3

[Liu et al 2006] Liu, Hui ; Ma, Zhiyi ; Zhang, Lu ; Shao, Weizhong: Detecting Duplications in Sequence Diagrams Based on Suffix Trees. In: *13th Asia-Pacific Software Engineering Conference (APSEC 2006), 6-8 December 2006, Bangalore, India*, URL https://doi.org/10.1109/APSEC.2006.32, 2006, p. 269–276

[Llano and Pooley 2009] Llano, Maria T. ; Pooley, Rob: UML Specification and Correction of Object-Oriented Anti-patterns. In: *The Fourth International Conference on Software Engineering Advances, ICSEA 2009, 20-25 September 2009, Porto, Portugal*, URL https://doi.org/10.1109/ICSEA.2009.15, 2009, p. 39–44

[Lúcio et al 2014] Lúcio, Levi ; Amrani, Moussa ; Dingel, Juergen ; Lambers, Leen ; Salay, Rick ; Selim, Gehan . ; Syriani, Eugene ; Wimmer, Manuel: Model Transformation Intents and Their Properties. In: *Software & Systems Modeling* 15 (2014), Nr. 3, p. 685–705

[Mansoor et al 2017] Mansoor, Usman ; Kessentini, Marouane ; Wimmer, Manuel ; Deb, Kalyanmoy: Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. In: *Software Quality Journal* 25 (2017), Nr. 2, p. 473–501

[Martino and Esposito 2016] Martino, Beniamino D. ; Esposito, Antonio: A rule-based procedure for automatic recognition of design patterns in UML diagrams. In: *Softw., Pract. Exper.* 46 (2016), Nr. 7, p. 983–1007. – URL https://doi.org/10.1002/spe.2336

[Mayvan and Rasoolzadegan 2017] Mayvan, B. B. ; Rasoolzadegan, Abbas: Design pattern detection based on the graph theory. In: *Knowledge-Based Systems* 120 (2017), p. 211–225. – URL https://doi.org/10.1016/j.knosys.2017.01.007

[Mengerink et al 2017] Mengerink, Josh ; Serebrenik, Alexander ; Schiffelers, Ramon ; Brand, Mark van den: Automated Analyses of Model-Driven Artifacts: Obtaining Insights Into Real-Life Application of MDE. In: *International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, 2017

[Mens and Gorp 2006] Mens, Tom ; Gorp, Pieter V.: A Taxonomy of Model Transformation. In: *Electr. Notes Theor. Comput. Sci.* 152 (2006), p. 125–142. – URL https://doi.org/10.1016/j.entcs.2005.10.021

[Mens et al 2007] Mens, Tom ; Taentzer, Gabriele ; Müller, Dirk: Model-driven Software Refactoring. In: *1st Workshop on Refactoring Tools, WRT 2007, in conjunction with 21st European Conference on Object-Oriented Programming, July 30 - August 03, 2007, Berlin, Proceedings*, URL http://netfiles.uiuc.edu/dig/RefactoringWorkshop/, 2007, p. 25–27

[Mens and Tourwé 2004] Mens, Tom ; Tourwé, Tom: A Survey of Software Refactoring. In: *IEEE Trans. Software Eng.* 30 (2004), Nr. 2, p. 126–139. – URL https://doi.org/10.1109/TSE.2004.1265817

[Meyers and Vangheluwe 2011] Meyers, Bart ; Vangheluwe, Hans: A framework for evolution of modelling languages. In: *Science of Computer Programming* 76 (2011), Nr. 12, p. 1223 – 1246. – URL http://www.sciencedirect.com/science/article/pii/S0167642311000141. – Special Issue on Software Evolution, Adaptability and Variability. – ISSN 0167-6423

[Misbhauddin and Alshayeb 2015] Misbhauddin, Mohammed ; Alshayeb, Mohammad: UML model refactoring: a systematic literature review. In: *Empirical Software Engineering* 20 (2015), Nr. 1, p. 206–251

[Mohagheghi et al 2013] Mohagheghi, Parastoo ; Gilani, Wasif ; Stefanescu, Alin ; Fernandez, Miguel A. ; Nordmoen, Bjørn ; Fritzsche, Mathias: Where does model-driven engineering help? Experiences from three industrial cases. In: *Software & Systems Modeling* 12 (2013), Nr. 3, p. 619–639

[Mokaddem et al 2016] Mokaddem, Chihab ; Sahraoui, Houari ; Syriani, Eugene: Towards Rule-Based Detection of Design Patterns in Model Transformations. In: *System Analysis and Modeling. Technology-Specific Aspects of Models - 9th International Conference, SAM 2016, Saint-Melo, France, October 3-4, 2016, Proceedings* Volume 9959. Saint-Malo : Springer, oct 2016, p. 211–225. – URL https://doi.org/10.1007/978-3-319-46613-2_14

[Mokaddem et al 2018] Mokaddem, Chihab ; Sahraoui, Houari ; Syriani, Eugene: Recommending Model Refactoring Rules from Refactoring Examples. In: Wasowski,

Andrzej (Editor) ; Paige, Richard F. (Editor) ; Haugen, Øystein (Editor): *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, ACM, 2018, p. 257–266. – URL https://doi.org/10.1145/3239372.3239406

[Mokaddem et al 2021] Mokaddem, Chihab ; Sahraoui, Houari ; Syriani, Eugene: A Generic Approach to Detect Design Patterns in Model Transformations Using a String-Matching Algorithm. In: *Software and System Modeling* (2021)

[Muller 2006] Muller, Pierre-Alain: *De la modélisation objet des logiciels à la metamodélisation des langages informatiques*, Université Rennes 1, Ph.D. thesis, 2006

[Niere et al 2002] Niere, Jörg ; Schäfer, Wilhelm ; Wadsack, Jörg P. ; Wendehals, Lothar ; Welsh, Jim: Towards pattern-based design recovery. In: *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, URL https://doi.org/10.1145/581339.581382, 2002, p. 338–348

[Porres 2003] Porres, Ivan: Model Refactorings as Rule-Based Update Transformations. In: *«UML» 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, URL https://doi.org/10.1007/978-3-540-45221-8_16, 2003, p. 159–174

[Prechelt and Krämer 1998] Prechelt, Lutz ; Krämer, Christian: Functionality versus practicality: Employing existing tools for recovering structural design patterns. In: *Journal of Universal Computer Science* 4 (1998), Nr. 11, p. 866–882

[Pretschner and Prenninger 2007] Pretschner, Alexander ; Prenninger, Wolfgang: Computing refactorings of state machines. In: *Software and Systems Modeling* 6 (2007), December, Nr. 4, p. 381–399. – URL http://dx.doi.org/10.1007/s10270-006-0037-5

[Priya 2014] Priya, R. K.: A survey: Design pattern detection approaches with metrics. In: *2014 IEEE National Conference on Emerging Trends In New Renewable Energy Sources And Energy Management (NCET NRES EM)*, Dec 2014, p. 22–26

[Ramasamy et al 2015] Ramasamy, Subburaj ; Jekese, Gladman ; Hwata, Chiedza: Impact of Object Oriented Design Patterns on Software Development. In: *International Journal of Scientific & Engineering Research* 6 (2015), Nr. 12. – ISSN 2229-5518

[Rasool et al 2010] Rasool, Ghulam ; Philippow, Ilka ; Mäder, Patrick: Design pattern recovery based on annotations. In: *Advances in Engineering Software* 41 (2010), Nr. 4,

p. 519–526. – URL https://doi.org/10.1016/j.advengsoft.2009.10.014

[Ren et al 2003] Ren, Shengbing ; Rui, Kexing ; Butler, Gregory: Refactoring the Scenario Specification: A Message Sequence Chart Approach. In: *Object-Oriented Information Systems, 9th International Conference, OOIS 2003, Geneva, Switzerland, September 2-5, 2003, Proceedings*, URL https://doi.org/10.1007/978-3-540-45242-3_29, 2003, p. 294–298

[Rohit et al 2005] Rohit, Gheyi ; Tiago, Massoni ; Paulo, Borba: Type-safe Refactorings for Alloy. In: *Proceedings 8th Brazilian Symposium on Formal Methods* Porto Alegre, Brazil (event), 2005, p. 174–190

[Saada et al 2012] Saada, Hajer ; Dolques, Xavier ; Huchard, Marianne ; Nebut, Clémentine ; Sahraoui, Houari: Generation of operational transformation rules from examples of model transformations. In: *Model Driven Engineering Languages and Systems*, 2012

[Saliha 2006] Saliha, Bouden: *Étude de la traçabilité entre refactorisations du modèle de classes et refactorisations du code*, Université de Montréal, Ph.D. thesis, 2006. – 1 vol. (160 p.) p. – Thèse de doctorat dirigée par Gueheneuc Yann-Gael

[Schach 1993] Schach, Stephen R.: *Software engineering (2. ed.)*. Irwin, 1993. – ISBN 978-0-256-12998-4

[Schulz et al 1998] Schulz, B. ; Genssler, T. ; Mohr, B. ; Zimmer, W.: On the computer aided introduction of design patterns into object-oriented systems. In: *Proceedings Technology of Object-Oriented Languages. TOOLS 27 (Cat. No.98EX224)*, Sep 1998, p. 258–267

[Sendall and Kozaczynski 2003] Sendall, Shane ; Kozaczynski, Wojtek: Model Transformation: The Heart and Soul of Model-Driven Software Development. In: *IEEE Softw.* 20 (2003), Nr. 5, p. 42–45. – URL https://doi.org/10.1109/MS.2003.1231150

[Shahir et al 2009] Shahir, Hamed Y. ; Kouroshfar, Ehsan ; Ramsin, Raman: Using Design Patterns for Refactoring Real-World Models. In: *35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009, Patras, Greece, August 27-29, 2009, Proceedings*, URL https://doi.org/10.1109/SEAA.2009.56, 2009, p. 436–441

[Singh and Kaur 2018] Singh, Satwinder ; Kaur, Sharanpreet: A systematic literature review: Refactoring for disclosing code smells in object oriented software. In: *Ain Shams*

*Engineering Journal* 9 (2018), Nr. 4, p. 2129 – 2151. – URL http://www.sciencedirect.com/science/article/pii/S2090447917300412. – ISSN 2090-4479

[Steinberg et al 2008] Steinberg, Dave ; Budinsky, Frank ; Merks, Ed ; Paternostro, Marcelo: *EMF: eclipse modeling framework*. Pearson Education, 2008

[Stoianov and Şora 2010] Stoianov, A. ; Şora, I.: Detecting patterns and antipatterns in software using Prolog rules. In: *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*, 2010, p. 253–258

[Van Der Straeten and D'Hondt 2006] Straeten, Ragnhild Van Der ; D'Hondt, Maja: Model Refactorings Through Rule-based Inconsistency Resolution. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2006 (SAC '06), p. 1210–1217

[Straeten and D'Hondt 2006] Straeten, Ragnhild Van D. ; D'Hondt, Maja: Model refactorings through rule-based inconsistency resolution. In: *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, URL http://doi.acm.org/10.1145/1141277.1141564, 2006, p. 1210–1217

[Strommer and Wimmer 2008] Strommer, Michael ; Wimmer, Manuel: A Framework for Model Transformation By-Example: Concepts and Tool Support. In: *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*, URL https://doi.org/10.1007/978-3-540-69824-1_21, 2008, p. 372–391

[Strüber et al 2016a] Strüber, Daniel ; Plöger, Jennifer ; Acretoaie, Vlad: Clone Detection for Graph-Based Model Transformation Languages. In: *Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings*, URL https://doi.org/10.1007/978-3-319-42064-6_13, 2016, p. 191–206

[Strüber et al 2016b] Strüber, Daniel ; Rubin, Julia ; Arendt, Thorsten ; Chechik, Marsha ; Taentzer, Gabriele ; Plöger, Jennifer: RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules. In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, URL https://doi.org/10.1007/

978-3-662-49665-7_8, 2016, p. 122–140

[Sun and Gray 2009] Sun, Jules ; Gray, Jeff: Model Transformation by Demonstration. In: Schürr, Andy (Editor) ; Selic, Bran (Editor): *Proceedings of the International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, 2009, p. 712–726

[Sunyé et al 2001] Sunyé, Gerson ; Pollet, Damien ; Traon, Yves L. ; Jézéquel, Jean-Marc: Refactoring UML Models. In: *«UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, URL https://doi.org/10.1007/3-540-45441-1_11, 2001, p. 134–148

[Syriani and Ergin 2012] Syriani, Eugene ; Ergin, Hüseyin: Operational semantics of UML activity diagram: An application in project management. In: *Second IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012, Chicago, IL, USA, September 24, 2012*, IEEE Computer Society, 2012, p. 1–8. – URL https://doi.org/10.1109/MoDRE.2012.6360083

[Syriani and Vangheluwe 2013] Syriani, Eugene ; Vangheluwe, Hans: A Modular Timed Graph Transformation Language for Simulation-based Design. In: *Software & Systems Modeling* 12 (2013), Nr. 2, p. 387–414

[Taentzer et al 2012] Taentzer, Gabriele ; Arendt, Thorsten ; Ermel, Claudia ; Heckel, Reiko: Towards refactoring of rule-based, in-place model transformation systems. In: *Proceedings of the First Workshop on the Analysis of Model Transformations, AMT@MODELS 2012, Innsbruck, Austria, October 2, 2012*, URL http://doi.acm.org/10.1145/2432497.2432506, 2012, p. 41–46

[Tairas and Cabot 2011] Tairas, Robert ; Cabot, Jordi: Cloning in DSLs: Experiments with OCL. In: Sloane, Anthony M. (Editor) ; Assmann, Uwe (Editor): *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers* Volume 6940, Springer, 2011, p. 60–76. – URL https://doi.org/10.1007/978-3-642-28830-2_4

[Tichy et al 2013] Tichy, Matthias ; Krause, Christian ; Liebel, Grischa: Detecting Performance Bad Smells for Henshin Model Transformations. In: *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, URL http://ceur-ws.org/Vol-1077/amt13_submission_6.pdf, 2013

[Tisi et al 2009] Tisi, Massimo ; Jouault, Frédéric ; Fraternali, Piero ; Ceri, Stefano ; Bézivin, Jean: On the Use of Higher-Order Model Transformations. In: Paige, Richard F. (Editor) ; Hartman, Alan (Editor) ; Rensink, Arend (Editor): *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings* Volume 5562, Springer, 2009, p. 18–33. – URL https://doi.org/10.1007/978-3-642-02674-4_3

[Štolc and Polášek 2010] Štolc, M. ; Polášek, I.: A visual based framework for the model refactoring techniques. In: *2010 IEEE 8th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Jan 2010, p. 72–82

[Toyoshima et al 2015] Toyoshima, Ichiro ; Yamaguchi, Shingo ; Zhang, Jia: A Refactoring Algorithm of Workflows Based on Petri Nets. In: *International Congress on Advanced Applied Informatics*, IEEE, 2015, p. 79–84

[Tsantalis et al 2006] Tsantalis, Nikolaos ; Chatzigeorgiou, Alexander ; Stephanides, George ; Halkidis, Spyros T.: Design Pattern Detection Using Similarity Scoring. In: *IEEE Trans. Software Eng.* 32 (2006), Nr. 11, p. 896–909. – URL https://doi.org/10.1109/TSE.2006.112

[Van Kempen et al 2005] Van Kempen, Marc ; Chaudron, Michel ; Kourie, Derrick ; Boake, Andrew: Towards proving preservation of behaviour of refactoring of UML models. In: *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries* South African Institute for Computer Scientists and Information Technologists (event), 2005, p. 252–259

[Varró 2006] Varró, Daniel: Model Transformation by Example. In: *Model Driven Engineering Languages and Systems*. Springer, 2006, p. 410–424

[Verebi 2015] Verebi, Ioana: A model-based approach to software refactoring. In: *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, URL https://doi.org/10.1109/ICSM.2015.7332524, 2015, p. 606–609

[Wimmer et al 2012] Wimmer, Manuel ; Perez, Salvador M. ; Jouault, Frédéric ; Cabot, Jordi: A Catalogue of Refactorings for Model-to-Model Transformations. In: *Journal of Object Technology* 11 (2012), Nr. 2, p. 2: 1–40. – URL https://doi.org/10.

5381/jot.2012.11.2.a2

[Wimmer et al 2007] Wimmer, Manuel ; Strommer, Michael ; Kargl, Horst ; Kramler, Gerhard: Towards Model Transformation Generation By-Example. In: *Annual Hawaii International Conference on System Sciences*, 2007, p. 285–295

[Yarahmadi and Hasheminejad 2020] Yarahmadi, Hadis ; Hasheminejad, Seyed Mohammad H.: Design pattern detection approaches: a systematic review of the literature. In: *Artif. Intell. Rev.* 53 (2020), Nr. 8, p. 5789–5846. – URL https://doi.org/10.1007/s10462-020-09834-5

[Zanoni et al 2015] Zanoni, Marco ; Fontana, Francesca A. ; Stella, Fabio: On applying machine learning techniques for design pattern detection. In: *Journal of Systems and Software* 103 (2015), p. 102–117. – URL https://doi.org/10.1016/j.jss.2015.01.037

[Zarour et al 2020] Zarour, Karim ; Benmerzoug, Djamel ; Guermouche, Nawal ; Drira, Khalil: A systematic literature review on BPMN extensions. In: *Bus. Process. Manag. J.* 26 (2020), Nr. 6, p. 1473–1503. – URL https://doi.org/10.1108/BPMJ-01-2019-0040

[Zhang et al 2005] Zhang, Jing ; Lin, Yuehua ; Gray, Jeff: Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine. In: Beydeda, Sami (Editor) ; Book, Matthias (Editor) ; Gruhn, Volker (Editor): *Model-Driven Software Development*. Springer, 2005, p. 199–217. – URL https://doi.org/10.1007/3-540-28554-7_9