Université de Montréal

# Improving Automation in Model-Driven Engineering using Examples

par
Martin Faunes Carvallo

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

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en computer science

June, 2013

## RÉSUMÉ

Cette thèse a pour but d'améliorer l'automatisation dans l'ingénierie dirigée par les modèles (MDE pour Model Driven Engineering). MDE est un paradigme qui promet de réduire la complexité du logiciel par l'utilisation intensive de modèles et des transformations automatiques entre modèles (TM). D'une façon simplifiée, dans la vision du MDE, les spécialistes utilisent plusieurs modèles pour représenter un logiciel, et ils produisent le code source en transformant automatiquement ces modèles. Conséquemment, l'automatisation est un facteur clé et un principe fondateur de MDE. En plus des TM, d'autres activités ont besoin d'automatisation, e.g. la définition des langages de modélisation et la migration de logiciels.

Dans ce contexte, la contribution principale de cette thèse est de proposer une approche générale pour améliorer l'automatisation du MDE. Notre approche est basée sur la recherche métaheuristique guidée par les exemples.

Nous appliquons cette approche sur deux problèmes importants de MDE, (1) la transformation des modèles et (2) la définition précise de languages de modélisation. Pour le premier problème, nous distinguons entre la transformation dans le contexte de la migration et les transformations générales entre modèles. Dans le cas de la migration, nous proposons une méthode de regroupement logiciel (Software Clustering) basée sur une métaheuristique guidée par des exemples de regroupement. De la même façon, pour les transformations générales, nous apprenons des transformations entre modèles en utilisant un algorithme de programmation génétique qui s'inspire des exemples des transformations passées. Pour la définition précise de langages de modélisation, nous proposons une méthode basée sur une recherche métaheuristique, qui dérive des règles de bonne formation pour les méta-modèles, avec l'objectif de bien discriminer entre modèles valides et invalides.

Les études empiriques que nous avons menées, montrent que les approches proposées obtiennent des bons résultats tant quantitatifs que qualitatifs. Ceux-ci nous permettent de conclure que l'amélioration de l'automatisation du MDE en utilisant des méthodes de recherche métaheuristique et des exemples peut contribuer à l'adoption plus large de MDE dans l'industrie à là venir.

**Mots clés : Ingénierie dirigée par les modèles, génie logiciel guidé par les exemples, génie logiciel automatisé, méta-modélisation, génie logiciel avec des méthodes de recherche heuristique.**

## ABSTRACT

This thesis aims to improve automation in Model Driven Engineering (MDE). MDE is a paradigm that promises to reduce software complexity by the mean of the intensive use of models and automatic model transformation (MT). Roughly speaking, in MDE vision, stakeholders use several models to represent the software, and produce source code by automatically transforming these models. Consequently, automation is a key factor and founding principle of MDE. In addition to MT, other MDE activities require automation, e.g. modeling language definition and software migration.

In this context, the main contribution of this thesis is proposing a general approach for improving automation in MDE. Our approach is based on meta-heuristic search guided by examples. We apply our approach to two important MDE problems, (1) model transformation and (2) precise modeling languages. For transformations, we distinguish between transformations in the context of migration and general model transformations.

In the case of migration, we propose a software clustering method based on a search algorithm guided by cluster examples. Similarly, for general transformations, we learn model transformations by a genetic programming algorithm taking inspiration from examples of past transformations.

For the problem of precise metamodeling, we propose a meta-heuristic search method to derive well-formedness rules for metamodels with the objective of discriminating examples of valid and invalid models.

Our empirical evaluation shows that the proposed approaches exhibit good results. These allow us to conclude that improving automation in MDE using meta-heuristic search and examples can contribute to a wider adoption of MDE in industry in the coming years.

**Keywords: Model-driven engineering, software engineering by examples, automated software engineering, metamodeling, search-based software engineering.**

# CONTENTS

## LIST OF TABLES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| CMT | Chain of Model Transformations |
| GA | Genetic Algorithm |
| GP | Genetic Programming |
| MDE | Model Driven Engineering |
| MT | Model Transformation |
| MTBD | Model Transformation By Demonstration |
| MTBE | Model Transformation By Example |
| n-to-m | Many-to-many |
| OO | Object Oriented |
| PSO | Particle Swarm Optimisation |
| SA | Simulated Annealing |
| SC | Software Clustering |
| SE | Software Engineering |
| TR | Transformation Rule |
| WFR | Well-Fromedness Rules |

# CHAPTER 1

# INTRODUCTION

## 1.1 Context

### 1.1.1 Model Driven Engineeting (MDE)

MDE is a paradigm that proposes reducing software system complexity by the mean of the intensive use of models and automatic model transformations.

The idea behind this paradigm is that software development, software understanding, and many software engineering activities, can be made easier by using the proper high-level abstraction models (domain models) to represent various perspectives of the software and by automatically transforming these models into low-level artifacts such as source code and test suites [33].

The assumption here is that dealing with domain models is considerably easier for many stakeholders than directly dealing with implementation artifacts.

### 1.1.2 The importance of automation and MDE

As mentioned earlier, the basic assumption of MDE is that domain models are easier to deal with than implementation artifacts. Still, a software system actually operates through its implementation artifacts. That implies that a mechanism is needed to, starting from domain models, produce, modify, and test low level artifacts. This mechanism cannot be manual. If this is the case, the advantages of the MDE paradigm will be compromised. Indeed, software developers will have to manually produce domain models and manually derive implementation artifacts from them. They also have to manually keep them up to date and manage the consistency between them after any modification. The experience showed that such manual tasks result in an important discrepancy between high-level representations of the software and their corresponding low-level ones.

Consequently, automation is a key factor and founding principle of MDE for many development and maintenance activities. In addition to model transformation, other examples of activities that require automation include the support for metamodel definition and testing, and the architecture of a software system recovery from the source code.

### 1.1.3 Automation difficulties

However, automation is not easy. Automating an activity comes always at a cost. First, the proper knowledge on how the activity is to be performed has to be gathered. Second, this knowledge must be converted into coherent and scalable algorithms. Finally, the algorithms have to be comprehensively validated over a test set that considers most of the common cases but also corner-case situations.

For many automation problems, gathering the proper knowledge and systematizing it is a challenge by itself. In the case of transformations, for example, even for widely used formalisms, large variations and sometimes conflicts can be observed in the available knowledge. Each expert can transform manually a specific model into another, but the elicitation of this knowledge into a general form that can be translated into a universal algorithm is another story. The situation is even worse when dealing with proprietary formalisms for which little knowledge is available.

Knowledge elicitation is not the only problem. The mapping of the general knowledge into an operational one is an important issue. The software specialist has to master both the problem domain (the knowledge) and the implementation domain (languages, tools, and environments). Now, let us keep an optimistic position. Suppose that the knowledge is gathered and converted into an algorithm, the next obstacle is to validate this algorithm with a reliable test activity.

## 1.2 Thesis problem

Not all the MDE activities require the same need and the same level of automation. In this thesis, we concentrate on two problems that we consider as fundamental to the MDE paradigm. The first problem is model transformation. This problem is studied under two perspectives: (1) general transformations that require to define the semantic equivalence between the constructs of two modeling languages, (2) transformations in the context of migration between programming/design paradigms. The second problem that we target in this thesis is the support for the definition of precise modeling languages.

### 1.2.1 Automating general model transformations

In our opinion, the activity that requires the highest level of automation is model transformation (call it MT for short). MT is the main mechanism that supports the MDE vision. Thus, the more automatic MT is the more effective will become MDE.

But, before going further, let us define MT. A Model transformation is a mechanism that receives, as input, a source model and generates, as output, it corresponding target model. The generation is done by applying, in general, a set of transformation rules. A transformation rule looks for a pattern in the source model and instantiates a corresponding pattern in the target model. An MT is defined at the metamodel level. That allows the creation of a general MT that transforms any instance of a source metamodel into its corresponding instance of the target metamodel.

MT automation is not an easy task. To write transformations, the MDE specialist must have a deep knowledge about the source and target metamodels as well as the semantic equivalence between the two. In recent years, considerable advances have been made in modeling environments and tools. From the point of view of the transformation infrastructure, a large effort has been made to define languages for expressing transformation rules and thus make the writing of transformation programs easier. However, having a good transformation language is only one part of the solution; the most important part, as we mentioned, is to define/gather the knowledge on how to transform any model conforming to a particular metamodel into a model conforming to another metamodel. For many problems, this knowledge is incomplete or not available. This is the motivation behind the research on learning transformation rules.

The approaches that try to learn transformation programs start from concrete transformation examples. Basically, these approaches take as input one or many transformation examples (each in the form of a source and a target model) and produce as output a set of transformation rules that generalize the transformation described in the examples.

The above-mentioned approaches have many limitations. First, most of them [4, 15, 25, 47, 54, 57, 60] require that the two models in each example must be complemented by transformation traces (TT). TT are fine-grained mappings giving the correspondence between the elements in the source models and those in the target model. Obviously this kind of information is difficult to obtain in the recorded past examples. Second, many approaches [4, 34, 35, 47, 57] do not produce actual operational rules, but only conceptual mappings that have to be later, manually, transformed into operational rules. The third limitation is that the rules produced are very simple as they involve one element in the source model and produce one element in the target model (1-to-1 rules). With the exception of [4, 25], no approach can derive m-to-n rules. As another limitation, current approaches are not properly tested or the complete settings and results are not reported [4, 15, 25, 54, 57, 60] . Finally, there is a group of approaches that instead of generating transformation rules, generalize edition actions that the user performs [39, 55]. The limitation here is that the obtained editing patterns are not as reusable as the transformation rules.

Our goal is to propose a new Model Transformation By Example (MTBE) approach that solves/circumvents all the above mentioned limitations.

### 1.2.2 Automating transformations in the context of migration

In some software engineering tasks that produce high-abstraction models from low-abstraction ones, the problem to automate MT is even harder. Indeed, software migration processes can be seen as a CMT. First, the legacy code is transformed into a model conforming to the legacy paradigm (reverse engineering transformation). Then, this model is mapped to a new one conforming to the new paradigm (paradigm shift). Finally, the obtained model is transformed into code (code generation).

The first and third transformations are particular cases of general transformations (see the previous sub-section). The second one is completely different as, usually, the new paradigm is more abstract. Its elements are in general clusters of those of the legacy paradigm. Examples are variable and procedure grouping into classes ([48, 49, 53, 56], or class grouping into components [2]. In these cases, the transformation is a clustering mechanism that cannot be encoded like a general transformation.

Existing work in software clustering uses heuristics to guide the clustering process. One of these heuristics is that elements that are structurally connected in the legacy model should be grouped together. The empirical validation of such heuristics showed that they produce relatively poor results [2, 13, 28, 40, 41, 48, 49, 53, 56, 61].

Our goal is to generate clustering-like transformations using a new heuristic based on the conformance to migration examples.

### 1.2.3 Automating the support to precise modeling

Usually MTs should accept as input only valid source models to produce valid target models. This observation leads us to the second activity that requires an important level of automation in MDE, namely, the definition of precise modeling languages (metamodels). MDE relies on models to represent software systems from different perspectives. To define a valid model of a given perspective, the model has to be created using the constructs and following the rules defined on the metamodel that describes this perspective. For example, to be a valid UML class diagram, a diagram has to instantiate the constructs types defined in its metamodel, but also conform to the well-formedness rules attached to it.

Creating precise metamodels is also a difficult task. Metamodels are usually composed of two components, a domain structure and a set of well-formedness rules. While the domain structure describes the main domain concepts and their relationships, the well-formedness rules impose further restrictions over models. This is to avoid models that, even if build using the concepts and relationships defined on the domain structure, are malformed.

Creating the domain structure, although it is a difficult task, is in general well carried by the modeling community. Defining rules to further restrict the domain structure is by far harder because it requires a high degree of formalization. Moreover, such rules need to be coherent, complete and correct.

Current approaches in this field are mainly focused on recovering the domain structure [31, 50]. The little work on constraint recovery only targets programs/source code constraints [14, 17, 45, 62]. Our goal is to propose an approach for the derivation of well-formedness rules that uses examples of valid and invalid models.

## 1.3 Contributions

The general contribution of our thesis is improving automation in MDE. We propose solutions to the automation problems based on examples. We see the automation as an optimization problem where solutions are searched based on examples.

In particular, we address automation on two important activities, (1) model transformation and (2) modeling language definition. In (1), we distinguish between (a) general transformations and (b) transformation in the context of migration.

In the problem of general model transformation, we contribute by proposing an approach that searches for transformation rules based on a genetic-programming search that is guided by conformance with examples. Unlike the other approaches, our approach does not need transformation traces.

Our contribution is characterized as follows.

– Proposal of a transformation rule derivation process based on a search (Genetic Programming) that is guided by conformance with examples.
– The proposed approach does not need transformation traces, and thus it is easy to be used.
– The proposed approach produces operational many-to-many transformation rules that can have complex conditions.
– The generic GP algorithm is adapted to the problem of deriving transformation rules.
– Adaptation includes a solution encoding for transformation rules compatible with GP, genetic operators (mutations and crossover) to be used in GP, a technique to create the initial population, and an efficient fitness function to guide the search. The fitness function compares target model produced by a set of transformation rules with the expected target model (the one given as example).
– The proposed approach is validated over two transformation problems: UML class diagram to relational schema and sequence diagram to state machines.

– We also created a base of examples that could be used as a benchmark for the evaluation of other approaches.

In the migration, we propose a software clustering approach that search for a solution by conformance with example clusters. Our approach main characteristics are as follows.

– Adaptation of Software Clustering (SC) to the process of MT in the context of migrating.
– Proposal of an SC method based on a search that is performed in two steps. A global search is performed with Particle Swarm Optimization (PSO), and then the result is polished by a local search, in particular, Simulated Annealing (SA).
– Adaptation of PSO and SA to the clustering problem in the context of MT in migrating.
– Proposal of a solution and example encoding compatible with the search process, and an efficient heuristic to guide the searches that compares candidate clusters with example clusters. Since the search process performs this comparison many times, the efficiency, here is a key factor.
– The proposed approach is validated on the specific problem of migrating procedural programs to the object-oriented paradigm.
– We create a reusable base of examples composed of procedural programs and their corresponding OO programs.

Finally, for the problem of modeling language definition, our contribution is an approach that automatically derives well-formedness rules for metamodels. The approach is based on a GP search that is guided by conformance with example and counter-examples models.

Our contributions are as follows.

– Formalization of the synthesis of well-formedness rules as a search problem.
– Definition of a set of operators to automatically synthesize OCL expressions.
– A series of experiments that demonstrate the effectiveness of the approach and provide a set of lessons learned for automatic model search and mutation.

## 1.4 Thesis structure

The remainder of this thesis is organized as follows. Chapter 2 reviews the state of the art related to our three contributions. In chapter 3, we draw the big picture of our thesis and discuss the role of the three proposed approaches to reach the global objective of the thesis. Chapters 4, 5 and 6, give the details of the three approaches in the form of articles. The document finishes by presenting, in chapter 7, a conclusion and some future perspectives of our work.

# CHAPTER 2

# RELATED WORK

The objective of this chapter is to present the related work of this thesis. As said before, this thesis is about automating MDE based on optimization searches guided by examples. Thus, in this chapter we survey works of this domain. The chapter is divided in three sections in which we discuss the three specific problems addressed. First we discuss the related work of model transformation in the context of migration, second we discuss the related work of general model transformation and finally we discuss the related work of improving modeling language definition.

## 2.1 Related work on model transformation in the context of migration

In this section we present the related work of the first approach [18] presented in this thesis, entitled "Deriving High-Level Abstractions from Legacy Software Using Example-Driven Clustering".

The contributions in this area have different objectives, represent the problem and the solution in many different ways, and use several algorithms. But they all have in common two aspects that we use to classify the approaches.

The first common aspect is that, all contributions perform an analysis of low-level model to extract facts, and based on this facts, they reify high-level models. The techniques used to perform this analysis can be divided into two types: static or dynamic. Static analysis is performed over the low-level model definition (e.g. source code). Dynamic analysis is performed over the source code execution traces.

The second common aspect of the surveyed techniques, is that, all contributions see the reifying of high-level models from low-level models as a grouping problem. In such a grouping, artifacts in the low-level model form groups that reify high-level artifacts. The groups are created following some intuition. We call this intuition *the meaningfulness hypothesis*. An example of hypothesis is that "elements that work together should be grouped together".

In what follows, we present the state-of-the-art, according to the proposed classification. At the end of the subsection we present a table summarizing the contributions.

### 2.1.1 Approaches based on static analysis

The first group of approaches reify high-level artifact by grouping low-level artifacts. The groups are formed by analyzing facts obtained from the low-level model after a static analysis. That is, the supposition here is that low-level artifacts have, in their definition, enough information to reify high-level artifacts, and that there is no need of observing their behavior.

#### 2.1.1.1 Cohesion and coupling

This subcategory of static analysis based approaches, reify high-level artifacts from source code based on the supposition that, elements that are close(far) in the application level should be close(far) in the domain level. They transform this supposition into an algorithm that measures the solution quality based on two metrics maximizing cohesion and by minimizing coupling, where cohesion measures the quantity of interconnection between elements of the same group and coupling measures the interconnection between elements of different groups. The relevant approaches are the following:

In 1998, Mancoridis et al. [40] propose a collection of algorithms to facilitate the automatic recovery of the modular structure of a software system for source code modules. The approach starts by creating a dependency graph of the source code modules (source code files) and their dependencies. The graph is, then, partitioned using different algorithms. The quality of a partition is measured in terms of module intra and inter dependency (cohesion and coupling). The algorithms used are optimal clustering that evaluates every possible graph partition, and three suboptimal algorithms that rely on neighboring, hill-climbing and genetic algorithms, respectively. The approach is implemented in a tool called Bunch and it is evaluated over four software systems.

In 1999, Mancoridis et al. [41] extend their previous work [40] by enabling the integration of designer knowledge about the system structure into an, otherwise, fully automatic cluster process. In particular the approach adds a functionality that allows the user to manually define clusters, the *orphan adoption*. They also allow to classify modules as omnipresent suppliers and omnipresent client, with suppliers modules not being subject to the clustering process.

In 2002, Harman et al. [28] present a search-based approach for automatic software modularization using a fitness function based on cohesion and coupling. The approach uses a genetic algorithm and has two main contributions, (1) a modularization representation such that each modularization has only one representation and (2) an adaptation of a genetic algorithm that preserves groups of elements (building blocks). The approach is tested over toy systems of about 24 elements.

In 2002, Sahraoui et al. [48] focus on grouping procedures and global variables to identify objects in procedural code. The approach sees the grouping problem as a partitioning problem of a call graph that relates procedures and global variables, in which the objective is minimizing coupling and maximizing cohesion. The partitioning problem is solved using a genetic algorithm and conceptual clustering which is a variation of hierarchical clustering. The approach is evaluated over three existing systems.

#### 2.1.1.2 Static metrics

This subcategory of static analysis based approaches, reify high-level artifacts from source code based on the supposition that beyond cohesion and coupling, there exist a set of static metrics that can be optimized to create good groups. This extension of the previous subgroup of works is based almost on the same underlying supposition. The relevant approaches are the following:

In 2005, Seng et al. [51] propose a clustering approach to improve subsystem decompositions. The approach is based on a genetic algorithm that optimizes a fitness function based on five OO metrics: coupling, cohesion, complexity, cyclic dependencies and bottlenecks. To facilitate the convergence, one half of the initial population is created favoring fittest individuals and, to favor diversity, the other half is created randomly. They use the classic one-point crossover and they define four mutations: splitting a cluster, merging two clusters, delete a cluster and adding a one element cluster to a bigger cluster. They test their approach over a large OO open source system to reorganize its classes into packages. The results show that the algorithm is able to find a good suboptimal solution for the five components of the fitness function.

In 2007, Harman and Tratt [27] present an approach for refactoring OO programs based on Pareto optimality. The approach consists in optimizing coupling and the standard deviation of the number of methods per class. The optimization is performed for each metric independently. For each metric, a search based algorithm (hill climbing) is run. The algorithm searches for a local optimum by randomly moving methods between classes. Finally, a Pareto front is built to integrate the information of the two runs and it is up to the user to select a solution in the Pareto front.

In 2009, Abdeen et al. [1] present an heuristic search-based approach for automatically improving package structure in OO software systems in order to avoid direct cyclic-connectivity. The approach uses Simulated Annealing (SA), for automatically increasing package cohesion and automatically reducing package coupling and package direct cyclic-connectivity. The optimization is done by moving classes over packages. The classes to move are selected randomly giving more probability to bad classes. The selected classes are randomly moved to a package in the neighborhood. The authors test their approach on four open source software systems and the results show that their approach is able to reduce coupling and direct cyclic-connectivity.

#### 2.1.1.3 Concept analysis

Also based on static analysis, the following subcategory of approaches are based on the idea that elements that represent common concepts in the application level should be grouped together in the domain level. In particular, these approaches form candidate high-level artifacts by performing formal concept analysis (Creation of concept or Galois lattices). The relevant approaches are the following:

In 1997, Sahraoui *et al.* [49] propose a computer supported approach aimed at easing the migration of procedural software systems to object-oriented technology. The approach first identifies candidate objects (groups of variables) in procedural source code using a Galois lattice. The lattice is created from an adjacency matrix that relates procedures to variables used by these procedures, which is the result of a static analysis of the source code. The candidate objects are then refined by using the same technique. Finally, an algorithm assigns the procedures to the identified objects in order to define their behavior. A example is used to illustrate the approach which is also validated on a large procedural system.

In 1999, Stiff and Reps [53] apply Concept Analysis for the modularization problem in the conversion of procedural C programs to OO C++ programs. In this approach, groups of procedures and variables become candidate objects. The main difference with [49] is that (1) the initial adjacency matrix also considers the return data type and the argument data types of procedures, and (2) it allows the use of complementary information added by the user such as negative information to represent situations in which a procedure does not have to use the user defined data type. The approach is tested over various C programs like those of the SPEC benchmark.

#### 2.1.1.4 Distance

Based on static analysis, as well, this subcategory of approaches creates groups, by measuring the distance of the low level elements. The underlying supposition here is that a mix of elements close in the application level should be close in the domain level with the one used by those that perform concept analysis and Galois lattices. That is, that elements that represent common concepts in the application level should be grouped together in the domain level. The relevant approaches are the following:

In 2006, Czibula and Şerban [13] presented an approach for refactoring based on *k*-means clustering. The approach consists of three steps. First, the relevant data is collected from source code and other artifacts. Second, the *k*-means clustering is applied. The *k*-means algorithm iteratively creates *k* clusters (groups of objects) that minimizes a fitness function that measures the distance between the objects in a cluster. In the third step, a list of refactoring actions is proposed by comparing the clustering result to the current system organization. They test their approach on a large OO open source software system where they reorganize classes, methods and attributes. They automatically compare the results of their approach with a known good partition by calculating two metrics: accuracy and precision. The authors claim to have results close to 99% for these two metrics.

In 1999, van Deursen and Kuipers [56] propose a method for identifying objects by semi-automatically restructuring the legacy data structures in large COBOL systems. The technique performs agglomerative clustering analysis under the hypothesis that record fields that are related in implementation are also related in the application domain. This hypothesis is treated as a fitness function that measures the euclidean distance in terms of programs that use the fields. The algorithm, then, tries to minimize the distance between fields of the same cluster and maximize the distance between elements of different clusters.

#### 2.1.1.5 Others

Finally, there are approaches that, after a static analysis, use other kind of hypothesis to create high-level artifacts as group of low-level artifacts.

In 1993 Canfora *et al.* [10] proposed a method to address the problem of identifying reusable abstract data types, in source code, in the context of reverse engineering and re-engineering. The method proposed searches for candidate abstract data types by looking for isolated subgraphs in a graph that links procedures and user defined data types. In their method, the graph is the result of a static analysis of the source code and the process of looking for isolated subgraphs is perform in Prolog. A case study is used for validation and to analyze the feasibility of the proposed method.

In 2006, Washizaki and Fukazawa [58] propose an approach for the refactoring of OO programs. The refactoring is made by identifying and extracting reusable components in the target programs and by making the proper modifications on the surrounding parts of original programs. The approach works as follows. First, a graph is created to represent the classes and their relationships (in particular, inheritance, reference and instantiation). Then, the graph is processed by a clustering algorithm to search for facade patterns. Finally, the targeted programs are modified in order to use this cluster as component. Even if the authors test their approach over several open source systems it is not clear how the approach performs since the number of components to be found is unknown.

### 2.1.2 Dynamic analysis

The second group of approaches, also reify high-level artifact by grouping low-level artifacts. However, these approaches are based on the idea that what is important when reifying high-level artifacts is their dynamic behavior rather than their static definition. Thus, these approaches use, as input, facts obtained from the execution of the low-level artifacts (execution traces) to reify high-level artifacts.

In 2005, Xiao and Tzerpos [61] proposed a clustering approach based on dynamic dependencies between software artifacts. The approach starts by creating a dynamic and weighted dependency graph for the studied system where the nodes are system source-code files and the edges are their dynamic dependencies. Then, the graph is partitioned using a clustering method. Finally the results are compared with those obtained from static dependencies. Sadly, the comparison shows that the static dependencies perform better than dynamic ones, even though, the latter can still give some good results.

In 2010, Allier et al. [2] propose an approach to restructure legacy OO applications into component-based applications. The approach is based on execution traces. The candidate components are identified by clustering the classes that appear frequently together in the execution traces of use cases. The clustering process is performed first by a global search. Then its results are refined by a local search. They used a genetic algorithm and SA respectively. Both search strategies are guided by a fitness functions that optimises cohesion and coupling. The approach is validated through the study of three legacy Java applications. Among the results they claim to have found more that 50% of the components of the largest application.

### 2.1.3 Summary

Table 2.I summarizes the main aspects of the related work on model transformation in the context of migration. As it can be seen in the table, most contributions are based on static analysis. Only two contributions are based on dynamic analysis. Among those based on static analysis, more that a half are based on cohesion and coupling or OO metrics, which are, the most common hypothesis. The other half is based on several hypothesis being the most common those based on concept analysis and distance. The objective that the surveyed approaches aim at are mainly migration and refactoring, but also, architecture recovering and reverse engineering. The most common source and target paradigm considered in these migration approaches are procedural to OO. But procedural to structured and OO to component have also been considered. On the other hand, when refactoring the most common paradigm is OO. Finally, among the algorithms used, the most common are clustering algorithms but also optimization search, GA, concept analysis and Galois lattice.

## 2.2 Related work on general model transformation

In this section we present the related work of the second approach [22] presented in this thesis, entitled "Genetic-Programming Approach to Learn Model Transformation Rules from Examples".

Like the first approach, the contributions in this area represent the problem and the solution in many different ways, use several algorithms and propose solutions of different levels of generality. However, in order to classify them, we pay attention to two main characteristics: (1) Model Transformation by Examples (MTBE) vs Model Transformation by Demonstration (MTBD) and (2) abstract vs operational transformations.

First, we divide the existing work between MTBE approaches and MTBD approaches. MTBE contributions are those that, based on examples of past transformations (pairs of source and target models), produce transformation rules, or mappings leading to transformation rules, or that directly transform a model. MTBD contributions are those that, based on examples of edition actions, transform a model.

Second, we divide the MTBE approaches between those that generate operational transformation rules and those that derive abstract mappings or that actually transform a model but do not produce any kind of transformation knowledge.

| Dynamic or Static | Meaningfulness hypothesis | Year | Authors | Objective | Source Paradigm | Target paradigm | Algorithm |
|---|---|---|---|---|---|---|---|
| Static | Cohesion and Coupling | 1998 | Mancoridis et al. | Architecture recovering | Procedural and OO | | Optimal clustering, Neighboring partitions, Hill climbing, GA |
| | | 1999 | Mancoridis et al. | Architecture recovering | Procedural and OO | | Optimal clustering, Neighboring partitions, Hill climbing, GA |
| | | 2002 | Sahraoui et al. | Migration | Procedural | OO | Optimal clustering, Neighboring partitions, Hill climbing, GA |
| | | 2002 | Harman et al. | Refactoring | Not mentioned | | Optimal clustering, Neighboring partitions, Hill climbing, GA |
| | Concept analysis | 1997 | Sahraoui et al. | Migration | Procedural | OO | Galois latice |
| | | 1999 | Stiff and Reps | Migration | Procedural | OO | Concept analysis |
| | Distance | 2006 | Czibula and Serban | Refactoring | OO | | k-means clustering |
| | | 1999 | van Deursen and Kuipers | Migration | Procedural | OO | Agglomerative clustering |
| | Others | 2005 | Washizaki and Fukazawa | Refactoring | OO | | Optimal clustering Based on a neighboring graph |
| | | 1993 | Canfora et al. | Migration | Procedural | Structured | Full search based on prolog |
| | OO Metrics | 2005 | Seng et al. | Refactoring | OO | | GA |
| | | 2007 | Harman and Tratt | Refactoring | OO | | Hill climbing and Pareto |
| | | 2009 | Abdeen et al. | Refactoring | OO | | Hill climbing |
| Dynamic | Cohesion and Coupling | 2005 | Xiao and Tzerpos | Reverse engineering | OO | | Hierarchical clustering, ACDC, GA, Agglomerative |
| | | 2010 | Allier et al. | Migration | OO | Component | GA |

Table 2.I: Summary of the related work on model transformation in the context of migration

In the following subsections, we present the state-of-the-art according to the above-mentioned classification. A summary table of the main contributions is provided at the end of the section.

### 2.2.1 MTBE

#### 2.2.1.1 Transformation rules

In 2006, Varró [57] proposed a first approach of MTBE. In his work, he derives transformation rules starting from a set of prototypical transformation examples with interrelated models. The examples are provided by the user. This semi-automatic and iterative process starts by analyzing the mappings between source and target example models along with the respective meta-models. The transformation rules are finally produced using an ad-hoc algorithm. Varró's approach can derive only 1-to-1 transformation rules, which is an important limitation, considering that the majority of transformation problems need n-to-m rules. The rules derived by this approach, can test the presence or the absence of elements in the source models. This level of expressiveness, which is enough for many transformation problems, is, however, insufficient for many common transformation problems like UML Class diagram to Relational diagram. Although source and target model examples could be collected from past manual transformations, providing the fine-grained semantic equivalence between the contained model constructs is difficult to guarantee. Finally, Varró did not publish the validation data. Consequently, we do not know how the approach behaves in a realistic scenario.

In 2007, Wimmer et al. proposed in [60] an approach that produces 1-to-1 transformation rules based on transformation examples and their traces. The derivation process is similar to the one proposed by Varró in [57], with the difference that Wimmer produces executable ATL [32] rules. As mentioned earlier, producing only 1-to-1 transformation rules is an important limitation for the approach applicability as most of the existing transformation problems cannot be addressed. Here again, the lack of a validation step does not allow assessing how the approach performs in realistic situations.

In 2008, Strommer et al. [54] extend the previous approach of Wimmer by enabling 2-to-1 rules in a derivation process based on pattern matching. From the implementation standpoint, the authors developed an Eclipse plug-in prototype for model transformation. Like the previous approaches, this contribution has not been validated in concrete settings, and the question on how the approach performs in realistic situations remains unanswered.

In 2009, Balogh et al. [4], improve the original work of Varró by using inductive programming logic (ILP) [42] instead of the original ad-hoc heuristic. However, like for the approach of Varró, the main idea is to derive rules using transformation mappings. One difference between the approach of Strommer et al. [54] and the one of Balogh et al. is that the former uses pattern matching to derive transformation rules and the latter uses inductive programming logic (ILP) [42]. This approach produces n-to-m rules from Prolog clauses that are obtained through a semi-automatic process in which the user has to add logical assertions until the ILP inference engine can produce the desired rule. In addition to the need of giving detailed mappings, the fact that the user has to interact with the ILP inference engine is a limitation of this approach. Like Varró, Balogh et al. also produce rule conditions of low expressiveness (only able to test the presence and absence of source elements). Balogh et al. claim being able to produce the set of rules needed to transform a UML class diagram to relational schema. Unfortunately, the set of rules obtained is not published. The paper also lacks a validation step, which makes it very difficult to assess the usefulness of the proposed approach.

In 2009, Garcia-Magarino et al. [25] propose an algorithm capable of creating n-to-m transformation rules starting from interrelated model examples and their metamodels. The rules are created in a generic model transformation language, and they are, later, translated to ATL for evaluation proposes. The paper presents a validation step that mainly focuses on approach capability to generate n-to-m rules, but it does evaluate the quality of the produced rules. As for other MTBE approaches, the need of transformation traces limits the approach applicability.

In 2010, Kessentini et al. [35] proposed an approach for deriving 1-to-m transformation rules starting from transformation examples. The rule derivation process is made by producing 1-to-m mappings between elements in the source metamodel and elements in the target metamodel. This contribution does not use traces and also it actually produces rules. Although the rules produced are of type 1-to-m, which is an improvement compared to other contributions, the absence of n-to-m rules is still a limitation. The rule conditions produced are very simple. They only test the presence of source model elements and are not able, among other things, to use the property values. Like in his previous work, Kessentini et al. based his approach on a hybrid heuristic search that uses PSO and SA. The authors also perform a validation step but this has the same threats to validity as the previous approach, i.e., correctness scores derived manually by the authors.

In 2012, Saada et al. [47], extend the work of Dolques et al. [15] by proposing a two-step rule derivation process. In the first step, Dolques' approach is used to learn transformation patterns from transformation examples and transformation traces. In the second step, the learned patterns are analyzed and those considered as pertinent are selected. Selected patterns are, then, translated into transformation rules in JESS language. As in [15] the produced rules are n-to-m. The approach is finally tested in a ten-fold experimental setting where precision and recall are calculated.

#### 2.2.1.2 Mappings and other forms of transformations

In 2008, Kessentini et al. [34] propose a model transformation approach that, starting from transformation examples and their traces, transforms a source model into a target model. This contribution differs from the previous ones because it does not produce transformation rules. It instead directly derives the needed target model by analogy with the existing examples. The transformation process is made by producing n-to-m mappings between elements in the source model and elements in the target model. The mappings are derived by an hybrid heuristic search that first finds an initial solution by performing a global search (using the meta-heuristic Particle Swarm Optimization (PSO) [16]) and then improves this solution by performing a local search (using the meta-heuristic SA [37]). These heuristic searches are guided by an objective function that measures the similarity between the solution and the base of examples. The fact that this approach does not produce rules could be a limitation. Indeed, when rules are produced, it is possible to manually modify and improve them in an iterative process that ends with a satisfactory set of rules. These rules can be, later, reused as many times as the user needs. In the case of this approach, when obtaining an incomplete transformation for a given model, the user has to manually correct and complete the target model, employing his time in a task that only solves a specific case without any potential of reuse. Although this approach was validated on industrial data, the evaluation process has threats to validity. Indeed, the authors calculated two correctness scores, one by automatically comparing the obtained model with the expected one, and the other by checking manually the correctness of the obtained model. The manual assessment was done by the authors themselves and not by independent subjects.

In 2010, Dolques et al. [15], propose an MTBE approach based of Relational Concept Analysis (RCA) [44]. The RCA based derivation process analyses a unique transformation example and its mappings, along with the source and target meta-models. It produces sets of recurrent n-to-m transformation mappings organized in a lattice. When the mappings (or transformation traces) are not available, Dolques et al. propose a technique to produce an alignment between the source and target example models. This alignment is based on the identifiers, which compromises the quality of the resulting mappings, as mentioned by the authors. Because this rule derivation approach was not validated on concrete data, it is difficult to assess to which extent it is efficient. Moreover, the need of transformation traces is also a limit, as it is the case for all the approaches except for [35].

### 2.2.2 MTBD

In 2009, Sun et al. [55] proposed a new perspective for the derivation of transformations, namely, Model Transformation By Demonstration (MTBD). In this work, the objective is to generalize model transformation cases. However, instead of using examples, the users are asked to demonstrate how the model transformation should be done by directly editing (e.g., add, delete, connect, update) the model instance to simulate the model transformation process step by step. The user editing actions are recorded and serve as matching and transformation patterns that can be later applied on a similar (or the same) model by performing a pattern-matching process. From the point of view of the expressiveness, the transformation patterns produced can contain sophisticated transformation actions like string concatenation and mathematical operations on numeric properties which is an improvement to the previous approaches. This approach is intended to perform an endogenous transformation, i.e., both source and target models conform to the same metamodel. Moreover, the transformation consists of changing the source model itself progressively to end with the target one. These two facts are important limitations of this approach. Another primordial limitation is that the generated transformation patterns are not really rules. Thus, they can hardly be reused in a state-of-the-art transformation languages such as ATL. Because Sun et al. do not perform a validation, we do not have enough elements to judge the performance of their approach on a realistic scenario.

In 2010, Langer et al. [39] propose an MTBD approach, very similar to the one of Sun et al., with the improvement of handling exogenous transformations. These are transformations in which the source and target models belong to two different metamodels. The main limitation of this approach is the same as the one of Sun et al., since it does not produce transformation rules but instead transformation patterns. As for many approaches, the absence of a rigorous validation makes it difficult to assess the performance of this approach in realistic scenarios.

### 2.2.3 Summary

Table 2.II summarizes the main aspects of the related work on general model transformation. In the table the tags "X" and "-" are used to assert the presence, respectively absence, of a property. The table is divided into two groups, approaches that perform MTBE and approaches that perform MTBD.

The first group is separated between contributions that aims to derive transformation rules and those that derive mappings and other kind of equivalence between the source and the target language.

All the approaches that derive transformation rules use transformation example pairs and, with the exception of one work, all of them use transformation traces. They use several derivation processes e.g. pattern matching and heuristic searches. Less than a half of these approaches derive full operational rules, the others do not. With the exception of two works, none of them derives n-to-m rules. None of them, neither, are able to derive the target model by performing advanced operations like concatenating two source model identifiers. They do not implement any kind of rule execution control different than the default (*i.e.* every rule triggers when matching the source pattern) and with the exception of two contributions, none of them presents a comprehensive validation step. The approaches that derive mappings and other kind of equivalences, also use transformation examples with traces. One of them does not use metamodels since it derives a transformation equivalence at a model level. They also use several derivation processes. Both of them derive n-to-m matchings between source and target models and none of them allows advanced operations or implements execution control. One of them performs a comprehensive validation step.

The second group of approaches, those that transform a model by demonstration, does not use transformation examples or transformation traces, nor metamodels. They use demonstrative editing actions. Both are based on a pattern matching derivation process that derives n-to-m transformation patterns that state how to modify the source model in order to obtain the target model. Both of them allow operations to enlarge the target model identifier space by concatenating source model identifiers. Sadly, as in the first group, none of them performs a comprehensive evaluation step.

## 2.3 Related work of improving modeling language

In this section, we present the existing work related to the third contribution [21] presented in this thesis, entitled "Automatically searching for metamodel well-formedness rules in examples and counter-examples".

To the best of our knowledge, this is the first contribution with the objective of deriving automatically well-formedness rules in the context of metamodeling. Thus, we cannot discuss contributions that target the same problem within the same context.

Instead, we discuss in this section three families of contributions to which our work is related. The first family concerns the metamodel definition and the resulting modeling space. Another type of contributions that we consider is the learning of invariants or rules from low level artifacts. Finally, we discuss a closely and complementary domain which is the automatic recovery or reconstruction of metamodel structures from examples or low level artifacts.

### 2.3.1 Metamodel definition and modeling space

In 2012, Cadavid et al. [8] perform an empirical study, which analyzes the current state of practice in metamodels that actually use logical expressions to constrain the structure (well-formedness rules). They analyzed dozens of metamodels coming from industry, academia and the Object Management Group, to understand how metamodelers articulate metamodel structures with well-formedness rules. They implement a set of metrics in the OCLMetrics tool to evaluate the complexity of both parts, as well as their mutual coupling. They observe that all the metamodels tend to have a small core subset of concepts, which are constrained by most of the rules. And they also observe that, in general, the rules are loosely coupled to the structure. However, the most interesting conclusion, from our perspective, is that there is a limited set of well-formedness rule patterns (22) that are the most used by metamodelers. As we will see in Chapters 3 and 5, we use these patterns as a starting point when searching for well-formedness rules in a metamodel.

With respect to the modeling space resulting from a metamodel definition, Cadavid et al. [26] studied the automatic selection of a set of models that better cover this modeling space. The selected set should both cover as many representative situations as possible and be kept as small as possible for further manual analysis. They use a meta-heuristic algorithm, SA, to select a set of models that satisfies those two objectives. The approach was positively evaluated using two metamodels from two different domains. In this thesis, we reuse the idea of model set selection to create the base of model examples. As it will be shown in Chapters 3 and 5, the base of examples includes a set of valid and a set of invalid models for a given metamodel. These examples allow to assess if a learned set of well-formedness rules discriminate well between the two sets of models.

### 2.3.2 Derivation of invariants

In 2007, Ernst et al. [17] proposed the Daikon system. This system allows the dynamic detection of likely invariants in C/C++, Java and Perl programs. This contribution, which is a precursor to the domain, is based on a derivation process that works as follows. First, a program is automatically instrumented in order to report the value of the program variables during the executions. Then the program is run many times in order to produce the execution traces. Finally, an inference engine reads the execution traces and produces likely invariants using a *generate-and-check algorithm* to test a set of potential invariants against the traces.

In 2011, Ratcliff et al. [45] complemented Ernst et al. work [17]. This contribution proposes a method, based on a GP algorithm, in which a population of randomly created invariants is improved by comparing their performance against the program execution traces. Since an evolutionary search can consider a very large amount of invariants, Rafcliff et al. propose a mechanism to filter the interesting invariants from those that are uninteresting. Sadly, this mechanism is not explicitly documented. Finally, the authors perform a case study that allows them to conclude that their approach can find more invariants than Daikon [17].

In 2011, Zeller [62] gave an overview of specification mining. The idea is to extract specifications from existing systems by effectively leveraging the knowledge encoded into billions of code lines. These specifications are models of the software behavior that can act as specifications for building, verifying, and synthesizing new or revised systems. Rather than writing specifications from scratch, developers would, thus, rely on this existing knowledge base, overcoming specification inertia. Among the specifications that Zeller identifies we can mention pre-conditions, post-conditions and dynamic invariants.

During the same year, Dallmeir et al. [14] pointed out that dynamic specification mining effectiveness entirely depends on the observed executions. If not enough tests are available, the resulting specification may be too incomplete to be useful. To address this problem, they use test case generation to systematically enrich dynamically mined specifications. The test case generation covers all possible transitions between all observed states, and thus extracts additional transitions from their executions.

### 2.3.3 Metamodel reconstruction

In 2008, Javed et al. [31] proposed an approach to infer a metamodel from a collection of instance models. This contribution is motivated by the fact that, in most metamodeling environments, the instance models cannot be loaded properly into the modeling tool without the metamodel. The inferring process is based on grammar inference algorithms. Finally, the approach feasibility is shown by performing a case study.

Later, in 2012, Sanchez-Cuadrado et al. [50] proposed an interactive and iterative approach to the metamodel construction enabling the specification of model fragments by domain experts, with the possibility of using informal drawing tools like *Dia*. These fragments can be annotated with hints about the intention or needs for certain elements. Given these informations, a meta-model is automatically induced, which can be refactored in an interactive way, and then compiled into an implementation meta-model using profiles and patterns for different platforms and purposes.

| | | MTBE | | | | | | | | | MTBD | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Transformation rules | | | | | | | Mappingas and other | | | |
| | | Varro 2006 | Wimmer et al. 2007 | Strommer et al. 2008 | Balogh et al. 2009 | Garcia-Magalino et al. | Kessentini et al. 2010 | Saada et al. 2012 | Kessentini et al. 2008 | Dolques et al. 2010 | Sun et al. | Langer et al. |
| **Type of Transformation** | | Exogenous | Exogenous | Exogenous | Exogenous | Exogenous | Exogenous | Exogenous | Exogenous | Exogenous | Endogenous | Exogenous |
| **Input** | Transformation examples pairs | X | X | X | X | X | X | X | X | X | — | — |
| | Transformation traces (links source and target model elements) | X | X | X | X | X | — | X | X | X | — | — |
| | Source and target Meta-model | X | X | X | X | X | X | X | — | X | — | — |
| | Transformation traces (Demonstrative editing actions) | — | — | — | — | — | — | — | — | — | X | X |
| **Derivation process** | | Ad-hod heuristic | Ad-hod heuristic | Pattern matching | IPL | Ad-hoc algorithm | Hybrid heuristic search | Hybrid heuristic search | Hybrid heuristic search | RCA | Pattern matching | Pattern matching |
| **Output** | Rules | X | X | X | X | X | X | X | — | X | — | — |
| | Executable rules | — | X | X | — | X | — | — | — | — | — | — |
| | n-m rules | 1-1 | 1-1 | 2-1 | n-m | n-m | 1-m | 1-m | — | — | — | — |
| | other | — | — | — | — | — | — | — | n-m matching betwe en source and model elements | n-m matching between source and model elements | n-m transformation patterns | n-m transformation patterns |
| | Advanced operations | — | — | — | — | — | — | — | — | — | Allows string concatenation and math ope. | Allows string concatenation and math ope. |
| | Rule execution control | — | — | — | — | — | — | — | — | — | — | basic |
| | Validation step | — | — | — | — | — | X | X | X | — | — | — |

Table 2.II: Summary of Model Transformation By Example related work

# CHAPTER 3

# IMPROVING AUTOMATION MDE BY EXAMPLES AND THE THREE APPROACHES

## 3.1 Chapter Objectives

The objective of this chapter is to draw a big picture of our contributions and to show how the three approaches contribute in improving automation in Model Driven Engineering (MDE).

In order to do that, we first describe the context of this thesis by giving a summary of the main aspects of MDE. We then discuss the role of automation in MDE and the techniques that, depending on the context, can be used to improve the degree of automation in MDE tasks.

A special attention is paid to Search Based and *by examples* approaches since they form the basis of our contributions.

After giving the context, we present the three MDE problems we target and the three approaches proposed in this thesis to improve automation on these problems.

Finalizing, the common aspects of the approaches are discussed and a summary of the chapter is given.

## 3.2 Model Driven Engineering (MDE)

MDE is a paradigm that promises reducing software system complexity by the mean of the intensive use of models and automatic model transformations. The main idea of MDE is that software development can be easied by using several high-level abstraction models (domain models) to represent the software system and that, by the mean of model transformation, automatically produce low-level abstraction models (implementation models) like source code and test sets [33]. The assumption here is that dealing with several domain models is considerably easier than directly dealing with implementation artifacts.

In the following paragraphs we recall the main concepts of MDE. Our definitions are based mainly on the ideas of [33] and [7]. These concepts will be used in the rest of this chapter.

**Models.** Models are software artifacts used to represent software systems. Models reduce the effort of understanding the system because they contain less details, focus on a specific perspective, and are intended for a specific audience. Still, for a given perspective, models give the same answers than the modeled system. Thereby, models are easier to deal with than the modeled system while covering the same information.

**Metamodels.** Several models are used to represent a software system, each one focuses on a specific perspective. Examples of these perspectives are the level of abstractions at which the model represents the system, the aspect that the model describes and the audience to whom the model is intended. The formal definition of these perspectives are provided by Metamodels. Metamodels are software artifacts that define the elements and relationship that can be used and the rules that must be satisfied to produce a valid model. Metamodels are also complemented with well-formedness rules [9] that bring additional constraints in order to make metamodels more precise in their definitions and thus safer in their utilization.

**Model Transformations (MT).** Producing and maintaining several up-to-date models can become a hard task. To alleviate such a task, MDE activities are thought to be performed automatically (when possible) by mean of model transformations.

MT is a mechanism that receives as input one or several source models and creates as output one or several target models by applying, for example, a set of transformation rules (TR). TR are declarative programs that look for a pattern of elements in the source model(s) and that instantiates a pattern of elements in the target model(s).

MT and TR are defined at a metamodel level. That allows them to receive as input any model instance of the source metamodel and create the corresponding model instance of the target metamodel. Consequently, the source and the target metamodels are inputs for a MT and for TR.

Defining a MT is a difficult task. The specific transformation knowledge has to be gathered and implemented into a set of TRs. More over, this set of TRs have to be correct, coherent, and general enough. However, once defined, a MT becomes a repetitive task and thus, subject of automation and reuse.

**Chain of Model Transformations (CMT).** Frequently a single MT is not enough to cover the conceptual distance between the source models and the target models. In this case a succession of MTs is used. We call this succession a chain of model transformations (CMT). CMT transforms source models into target models by applying successive MT in a similar way that an assembly line does.

**SE tasks and MDE tasks.** Under MDE paradigm, every Software Engineering (SE) task (that are called MDE tasks) relies at some point on a CMT. For example, forward engineering [23] is seen as a CMT that starts with the domain expert and the SE practitioner providing domain models, and that finishes by producing implementation models like source code. As another example, model driven testing engineering [29] is also seen as a CMT. In this case the CMT starts with domain models representing the system and finishes by producing implementation models like test cases.

## 3.3 Improving Automation in MDE

In this thesis we present three approaches that increase the degree of automation in MDE. Before describing these approaches, let us answer the question "What automation is?". Dictionary.com defines automation as:

1. The technique, method, or system of operating or controlling a process by highly automatic means, as by electronic devices, reducing human intervention to a minimum;
2. a mechanical device, operated electronically, that functions automatically, without continuous input from an operator;

3. by extension, the use of electronic or mechanical devices to replace human labor.

Thus automation happens when human activities (specially productive ones) are replaced by electronic devices automatically controlled.

According to Sheridan [52], the degree of automation, is the relation between the tasks of a process that are allocated to machines and those allocated to humans. Wei et al. [59] extend this definition to consider the difficulty of the task. For these authors, the tasks can be the process performed to transform an input into an output or the control activities needed to assure that the process works properly.

Parasuraman et al. [43] define a scale of 10 levels to measure the degree of automation in information processing that depends on how less human interaction is needed to perform a process or a task. At level 1 (lowest) the computer offers no assistance and thus the human must take all decisions and actions. At level 10 (highest) the computer decides everything, acts autonomously and ignore the human. But the less quality an automated process achieves the more human interaction is needed in controlling and validating. Thus, the better is the quality of a process, the better is the level of automation. By quality we mean correctness, coherence, safety, rapidity and generality.

Therefore, in the case of MDE, improving the degree of automation is about replacing MDE tasks, that are performed manually by automatic (computer based) processes. In the case of MDE tasks that are already performed automatically, improving automation is about replacing them by new processes that need less human interactions.

Of course MDE is not meant to be a fully automated approach. Manual tasks are impossible to avoid [3]. At some point, a human being has to validate the automatically produced result or to organize the overall process. At the end, it is up to humans to define, for example, which are the model transformations that are interesting to perform or the systems that are interesting to be modeled. In this sense automation in MDE do not pretend to replace the human role but to avoid performing manually all those tasks that, because of their nature (complexity, time consuming, error prone and repetitive), are better done automatically.

## 3.4 Automation Techniques and By Example approaches

To improve automation, the proper techniques have to be chosen. This choice depends on the context. If the automation algorithm is known and performant, a program implementing it is enough, like in the case of sorting [30].

But there are situations in which the algorithm is unknown or not scalable. In these cases techniques like metaheuristics, search based algorithms and machine learning algorithms can be used, *e.g* [11], where a search algorithm is used to solve the vehicle routing problem.

These techniques can be combined with examples to solve specific problems (*e.g.* [36]) by analogy with examples or, even, to learn a good general subobtimal solution out from examples (*e.g.* [5]).

This idea has been largely applied in machine learning to automate objects classification (*e.g.* [63]) since the classification algorithm is, in general, unknown. In these applications, images of objects with their corresponding classification are given as examples to a machine learning process. The process analyzes the examples to determine what properties are important for the classification as well as the parameter values of the classifier. Once the properties determined and the parameters set, the application is able to automatically classify a new object (not present in the example set).

Examples have also been used in MDE to learn model transformations. Certainly, for some specific kind of models, the transformation algorithm is well known (*e.g.* Sequence diagrams to Petri nets models [6]). But, in general, the algorithm to transform a model of an arbitrary kind to its corresponding model of another arbitrary kind is unknown [12]. For this general case, approaches have been proposed that use examples models (pairs of source and target models). The examples are analyzed to set some learning or deriving artifact (*e.g.* [60]). Once the examples processed, the approaches are able to transform a new source model into its corresponding target model.

Whether the approach is intended to solve a specific problem following examples or to abstract a general solution out from examples, a mechanism to process the examples and to obtain information or knowledge from them, is needed. The choice of this mechanism depends on the problem's nature, on how general the solution is expected to be and also on how much information about the solution is known beforehand.

## 3.5 Targeted automation problems

Among all the tasks and aspects that can be improved when increasing automation in MDE, we are interested in three specific activities: software migration (SM) [24], model transformation (MT) [57] and precise metamodeling (PM) [46].

SM can be seen as a three steps process: (1) reverse engineering of models from the legacy code, (2) model transformation from the legacy paradigm to the new one, and (3), code generation from the model. The second step is a model transformation problem with the particular property that constructs of the new paradigm are obtained by clustering the ones of the legacy paradigm.

MT, in turn, is the classical problem of capturing the semantic equivalence between two metamodels.

Finally, PM aims to having precise metamodel definitions by adding well-formedness rules to the structure definition. Since metamodels are inputs for MT, PM is an important element in MDE.

In the next three sections we discuss the three approaches proposed in this thesis to improve automation in these three aspects of MDE. We finish the chapter by discussing the common elements to the three approaches and by connecting these ideas with the central theme of this thesis.

## 3.6 First approach : Deriving High-Level Abstractions from Legacy Software Using Example-Driven Clustering

The first approach [18] presented in this thesis, called "Deriving High-Level Abstractions from Legacy Software Using Example-Driven Clustering", introduces improvements in automation by modeling transformation in the context of migration as a Software Clustering (SC) problem.

This new approach, instead of creating clusters by optimizing SE metrics (*e.g.* [41]) or by grouping together elements that share common properties (*e.g.* [1]), creates clusters by similarity with cluster examples.

We argue that SE metrics and common properties do not necessary lead to reify high level software artifacts of good quality and that, by creating clusters using examples the quality of the derived constructs improves.

As for other SC approaches (e.g. [41]), we represent the SC problem as a graph partitioning problem and, since a general solution for this problem is unknown, we use a combination search based method that finds a suboptimal solution. In particular we use Particle Swarm Optimization (PSO) [16] and Simulated Annealing (SA) [37].

## 3.7 Second approach: Genetic-Programming Approach to Learn Model Transformation Rules from Examples

The second approach [22], called "Genetic-Programming Approach to Learn Model Transformation Rules from Examples" introduces improvements in automation for the production of Transformation Rules.

This approach proposes a method to automatically derive (generalize) the set of TRs that implement a MT by using as input examples of the transformation (pairs of source and target models).

Since in general, the algorithm to produce a coherent set of TRs that properly implement a MT is unknown and that TRs are programs, we decided to use Genetic Programming (GP) [38].

GP is a evolutionary algorithm and also a search based algorithm. It is used to automatically search for a program that approximate a behavior. GP is used in a context in which manually producing the program is difficult, but in which, there exist examples based on what GP can derive an approximation.

The idea of automatically deriving a MT from examples is not new (*e.g.* [57]) but our approach needs less human interaction and proposes more general solutions than the ones of the state of the art.

## 3.8 Third approach: Automatically searching for metamodel well-formedness rules in examples and counter-examples

The third approach [21], called "Automatically searching for metamodel well-formedness rules in examples and counter-examples", introduces improvements in the automation of well-formedness rule derivation for metamodels.

WFRs help in defining more precise metamodels by constraining the models that a metamodel can accept as valid, and, therefore, avoiding poorly constructed models [9].

The approach uses examples of valid models and invalid models as inputs and uses these examples to derive general well-formedness rules for the metamodel.

In this approach, we also propose a derivation method based on GP and examples, and the justification is similar to the one articulated for the second approach. Indeed, WFRs are programs and the algorithm to produce them is unknown. Still, there are examples that can be used to derive an approximation of them.

To our knowledge metamodel WFRs are currently created manually. Therefore, this method is the first attempt to automate this task.

## 3.9 Common aspects on the three approaches

In this thesis, the three approaches have in common that they all improve important parts of MDE.

The first one improves the migration process by formalizing the paradigm shift as a model transformation problem. Unlike general transformation, paradigm shift is seen as a clustering task. The second approach improves the CMT by automating the derivation of transformation rules. Finally, the third approach improves CMT by proposing a method for the automatic derivation of well-formedness rules for metamodels.

The three approaches have in common the fact that they all use examples. The first one creates clusters by conformance with cluster examples. The second one derives transformation rules by using examples of past transformations (pairs of source and target models) as input, and the third one uses examples of valid and invalid models to derive general well-formedness rules.

The third common aspect is that the three approaches solves a problem for which the algorithm is unknown, and each one proposes a solution based in search based algorithm like GP, PSO and SA. The first approach uses a combination of PSO and SA to perform clustering by analogy. In contrast, the two other approaches adapt GP to abstract knowledge (transformation rules and well-formedness rules) from examples.

## 3.10 Chapter summary

MDE reduces software system complexity easing its development and maintenance. MDE relies on the use of models to represent systems and uses CMTs to derive various software artifacts from these models.

Even if human interaction is unavoidable, improving the degree of automation in MDE tasks remains an important challenge.

The technique used to improve MDE tasks depends on the nature of the specific problem. For problems where efficient algorithmic solutions are known, a program implementing the algorithm is good enough. But when the algorithm is unknown or not scalable, other techniques like search based and machine learning algorithms can be combined with examples to solve the problem in specific cases or even to find general solutions.

In this thesis we proposes three approaches that improves software migration, model transformation and precise metamodeling, three important activities of MDE. The three approaches use examples and solve a problem in which a scalable algorithm is unknown by applying search based algorithms.

In the next three chapters we present in detail each approach and support our proposal with empirical validations.

# CHAPTER 4

# DERIVING HIGH-LEVEL ABSTRACTIONS FROM LEGACY SOFTWARE USING EXAMPLE-DRIVEN CLUSTERING

In this chapter we present the first contribution [18] entitled "Deriving High-Level Abstractions from Legacy Software Using Example-Driven Clustering" published at "Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research", 2011, pages 188-199. The contribution aims to improve automation in MDE. Specifically, the approach proposes a method to perform model transformation in the context of migration. The approach allows reifying a high-level of abstraction model from low level artifacts (source code) by performing a software clustering process based on a hybrid optimization search. The search, instead of being guided by structural metrics, is guided by conformance with known good clusters used as examples. Based on our empirical results, we argue that driving the search by conformance to examples leads to better results than current approaches. This contribution is the follow-up of [19], which was presented in GECCO 2011. In the remainder of this chapter we present the paper.

# Deriving High-Level Abstractions from Legacy Software Using Example-Driven Clustering

Martin Faunes, Marouane Kessentini and Houari Sahraoui

DIRO, Université de Montréal, Montréal, Canada

## Abstract

Much research in the past two decades has focused on automatic generation of abstractions from low-level software elements using clustering algorithms. This research is generally motivated by comprehension improvement through more abstract constructs, re-architecture of existing systems to improve their maintenance, or migration to new paradigms. In this paper, we start from a formulation of software clustering problems in a setting, where elements of a software system form a graph to be partitioned in order to derive high-level abstractions. We then propose a novel formulation where the graph partitioning solution is evaluated by the degree of its conformance with past clustering cases given as examples. We provide a concrete illustration of this formulation with the problem of object identification in procedural code.

## 1 Introduction

For more than 20 years, researchers have been proposing approaches and algorithms to automatically derive more abstract constructs from existing low-level software elements. This research work is generally motivated by program comprehension, re-architecture of existing systems to improve their maintenance, or migration to new paradigms. In general, deriving abstract constructs is treated as a clustering problem. Indeed, the proposed work tries to obtain abstractions by clustering basic elements. Early examples are abstract data type identification as groups of variables and procedures (e.g., [1]), remodularization as groups of procedures (e.g., [11]), and classes identification as groups of procedures and variable (e.g., [5]). Recent work concentrates, among others, on package restructuring as groups of classes (e.g., [3]), and components identification as groups of classes (e.g., [14]).

In modern software engineering, abstract elements such as classes and components generally reify actual entities or functions of the application domain. Consequently, their existence obeys the application semantics. If we want to obtain them by clustering existing low-level constructs in the code, we should have a means to approximate the application semantics from the code. The more a group of basic elements conforms to an application entity according to the approximation method, the more it is considered as an acceptable abstraction.

The most commonly used approximation heuristic in software clustering (SC) problems is that elements that are close structurally are also close semantically, i.e., they form an abstraction that represents an application entity or function. For example in a C program, if a set of procedures accesses the same set of variables, chances are that both sets define respectively the behavior and structure of a class corresponding to an application entity. We call this the structure-semantics-equivalence hypothesis. Another approximation method is based on the hypothesis that elements that are changed together frequently are semantically close (simultaneous-change hypothesis).

Although the existing approximation methods give good results in many clustering problems, they generate a lot of false positives. This makes it difficult to rely on them for automated processes. In this paper, we propose a new formulation of software-clustering problems. In this formulation, we view the clustering as a grouping process guided by the similarity with past clustering examples. We illustrate our proposal with the well-know clustering problem of object identification in procedural code. Our evaluation showed that the majority of identified objects are correct. Furthermore, our "by-example" approach outperforms a classical one [4].

The paper is organized as follows: Section 2 shows the classical formulation of software-clustering problems. This is done by synthesizing the existing work in a formal framework. At the end of this section, we highlight the limitations of this classical formulation. To alleviate these limitations, we propose a new formulation that is described in Section 3. To illustrate this new formulation, we present in Section 4 an application to the problem of object identification (OI) in procedural code. The first part of this section is dedicated to the mapping between the OI problem and our formulation. In the second part, we evaluate the derived solution on existing legacy code. Concluding remarks are finally given in Section 5.

## 2 SC as a Graph Partitioning Problem

This section has two goals. First, it presents a good sample of the exiting work on software clustering problems. The second objective is to propose a uniform formulation of the SC problems to better analyze the existing research contributions and highlight their limitations.

In the majority SC contributions, three important elements are usually specified: (1) the definition of the clustering problem, (2) the definition of an objective function to evaluate the quality of a solution, and (3) the concrete algorithm used to perform the clustering, i.e., to derive the solution. In this section, we propose a uniform formulation for the three above-mentioned elements and present the existing work accordingly.

### 2.1 Problem Definition

The majority of software clustering contributions is based on the structure-semantics-equivalence hypothesis. In this context, groups of elements that are structurally dependent are viewed as abstractions corresponding to application domain entities. In this setting, the contributions can be summarized as: given a set of elements composing a software and the dependency relationships between them, find groups (clusters) of elements that maximize an objective function and/or satisfy a set of constraints (e.g., [3]). There are many ways to formally represent the problem of software clustering. The most common way is to view it as a graph partitioning problem [16]. The notion of graph could be explicit (e.g., [17] and [9]), or implicit (e.g., [21] and [2]) where a matrix representation is used.

#### 2.1.1 Graph-based Representation

The software to be clustered defines a graph $G(V, E)$. The set of vertexes $V$ represents the elements of interest in the software and the set of edges $E \subseteq V \times V$ represent the dependencies between these elements. The nature of the elements in $V$ and their dependencies in $E$ are problem specific. For example, when clustering classes into packages, $V$ is the set of classes in the system and $E$ the relationships between classes such as method invocations (e.g., [3]).

The graph $G$ is typically a directed graph (e.g., [5] and [17]). In directed graphs each edge $e \in E$ is an ordered un pair $\langle u, v \rangle$ where an element $u \in V$ depends on an element $v \in V$, but the inverse is not necessarily true. For example, when deriving packages by grouping classes, $E$ could represent the method invocations between classes. In that case, the graph is directed. In the case of undirected graphs, each $e \in E$ is an unordered pair $(u, v)$ stating that there is a dependency between u and v without a specific direction. In the rest of this paper, we will consider only directed graph as undirected graphs could be easily transformed into a directed one.

For a clustering problem, different types of elements in a software could be of interest. Consequently, the vertexes in $V$ and edges in $E$ are typed. This can be represented by two functions (1) $t_V: V \rightarrow T_V$ associating each element $v \in V$ with its type $t_v \in T_V$ where $T_V$ is the set of possible types of $V$ and (2) $t_E: E \rightarrow T_E$ associating each element $e \in E$ with its type $t_e \in T_E$ where $T_E$ is the set of possible types of $E$. For example in object identification, procedures *call* procedures and *use* variables. Formally

$T_V = \{procedure, variable\}$

$T_E = \{call, use\}$

The definitions of types for vertexes and edges could be explicit (*e.g.,* [5] and [4]), or implicit (*e.g.,* [9]). Depending on the type of vertexes an edge connects, there are restrictions on the type it can take. For example, a variable can't *call* or *use* a procedure. This type of restriction can be represented formally as a function

$$p_E: T_V \times T_V \rightarrow P_E \subseteq T_E$$

that specifies for each pair of types of vertexes in $T_V$ the subset $P_E$ of allowed types of edges in $T_E$.

As stated by Mitchell in [18], for some clustering problems, edges in the graph could be weighted. The weight is defined as a function $W: E \rightarrow \mathbb{R}$. This function is used to measures the strength of the dependency between two elements. For instance, the weight could indicate the number of method invocations between two classes.

#### 2.1.2 Graph Partitioning

Now that we have introduced the graph based representation of a program, the next step is to describe how the graphs are partitioned to achieve the clustering goal. As described in [17], the partition of a graph G into m clusters is formalized as

$$P_G = \{K_1, K_2, \dots, K_m\}$$

where each $K_i$ is a cluster corresponding to a sub-graph of $G$ such that:

- Each cluster $K_i$ is a non empty sub-graph of $G$:

  $$K_i = (V_i, E_i) | V_i \subseteq V, V_i \neq \emptyset,$$

  $$1 \leq i \leq m, \quad m \leq |V|$$

  $$E_i = \{\langle u, v\rangle \in E \times E | u \in V_i \wedge v \in V_i\} \qquad (1)$$

- Each vertex in $G$ belongs to at least one cluster:

  Constraint 1: $\bigcup_{i=1}^{m} V_i = V$ (2)

- Each vertex in $G$ belongs to no more than one cluster:

  Constraint 2: $V_i \cap V_j = \emptyset \quad \forall\, i \neq j$ (3)

The above mentioned constraints are sometimes ignored to cover a broader range of clustering problems. In the case of component identification (e.g., [6] and [14]), all the classes could not be assigned to components, which violates the first constraint. Moreover, when components are identified with the perspective of reuse, a same class could be considered for more than one component (violation of the second constraint). For other problems, additional constraints are added. This is particularly the case for the remodularization where part of the initial architecture should be preserved like in [3]. Another possible constraint is that some elements should be in the same cluster (e.g., [17]).

## 2.2 Evaluating Clustering Solutions

An objective function is necessary to evaluate the quality of a partition and to guide the partitioning process. In general, the objective function involves a vector $M$ of metrics to be measured on a clustering solution. Each metric defines an objective to reach. To combine this multiple objectives into a single value, the most common technique is to use a vector $W$ of weights that define the importance of each metric. The metrics in $M$ depend on the software structure as it appears in the graph $G$ and the partition solution $P_G$. The most frequently used metrics are cohesion and coupling (e.g., [13]). Formally, the objective function $f$ is defined as follows.

$$f: (M, W) \rightarrow [0,1]$$

where each metric $M_i$ and weight $W_i$ are defined as functions:

$$M_i: (G, P_G) \rightarrow [0,1] \text{ and } W_i: i \rightarrow [0,1]$$

Metrics that have in general different definition domain are normalized ([0,1]) to ease their combination. As a consequence, the objective function is generally defined in the interval [0,1] , which eases the interpretation and the comparison (*e.g.,* [9]). Sometimes, other definition domains are used (*e.g.,* [16]).

When the dependencies of the graph $G$ are obtained by static analysis, static metrics are used (*e.g.,* [17] and [13]). Conversely, when $G$ is derived by dynamic analysis (case of execution traces), dynamic metrics are used (*e.g.,* [6] and [14]).

### 2.2.1 Cohesion and Coupling as metrics

Cohesion and coupling are the most used metrics for clustering evaluation. This is due to two reasons. First, for graph partitioning problems, we usually seek for solutions that maximize intra-group dependencies (cohesion) and minimize inter-group dependencies (coupling) (see for example [18] and [9]). Second, from the quality point of view, cohesion and coupling are two important properties.

Cohesion (call it $COH$) measures the mutual proximity of a group of elements in terms of dependencies. Accordingly, the clustering process should place together elements that are interdependent. When the used graph contains unweighted edges like in [16], the cohesion of a partition $COH(P_G)$ is usually calculated as the average of cohesions of its clusters $COH(K_i)$.

$$COH(P_G) = \frac{1}{k}\sum_{i=1}^{n} COH(K_i) \in [0,1]$$

This is evaluated as the number of internal edges $|IntEdges|$ between elements of $K_i$ normalized by the total number of possible edges $PIE(K_i)$ between the elements of a cluster $K_i$.

$$COH(K_i) = \frac{|IntEdges(K_i)|}{PIE(K_i)} \in [0,1]$$

$$IntEdges(K_i) = \left\{ \langle u, v\rangle \in E \,\middle|\, \begin{matrix} (u \in K_i) \\ \wedge\ (v \in K_i) \end{matrix} \right\}$$

Coupling (call it $COU$) measures the dependencies between elements belonging to different clusters. The clustering process tries to minimize coupling between those elements. When the used graph contains unweighted edges like in [9] and [3], the Coupling $COU(K_i, K_j)$ between pairs of clusters is calculated as the number of edges inter-clusters $|ExtEdges(K_i, K_j)|$. Coupling is usually normalized by the number of possible edges inter-clusters $PEE(K_i, K_j) = V_i \times V_j$.

$$COU(K_i, K_j) = \frac{|ExtEdges(K_i, K_j)|}{PEE(K_i, K_j)} \in [0,1]$$

where

$$ExtEdges(K_i, K_j) =$$

$$\left\{ \langle u, v\rangle \in E \,\middle|\, \begin{matrix} \big((u \in V_i) \wedge (v \in V_j)\big) \\ \vee \\ \big((u \in V_j) \wedge (v \in V_i)\big) \end{matrix} \right\}$$

The coupling of a partition $P_G$ is calculated as the average of the couplings between pairs of the $m$ clusters.

$$COU(P_G) =$$

$$\begin{cases} \dfrac{\frac{1}{m(m-1)} \sum_{\substack{i,j=1 \\ i<j}}^{m} COU(K_i, K_j)}{2} & m > 1 \\ 0 & m = 1 \end{cases}$$

A good example on how cohesion and coupling are combined into a single objective function is the one given in [18] where the objective function, to be maximized, is BasicMQ. BasicMQ is calculated as the difference between cohesion and coupling:

$$BasicMQ(P_G) = \begin{cases} COH(P_G) - COU(P_G) & m > 1 \\ COH(P_G) & m = 1 \end{cases}$$

Both cohesion and coupling could be calculated taking into account the weights of the edges as in [18], where another objective function, called TurboMQ, is defined. Finally, other approaches are used to combine cohesion and coupling. Examples are threshold-based metric aggregation [4] and Pareto optimal analysis [8].

### 2.2.2 Other Metrics and other Objective Functions

In addition to coupling and cohesion, other metrics are used in clustering problems. An important one is the number of cyclic-dependencies between the clusters (e.g., [9] and [12]). In both contributions the number of cyclic-dependencies has to be minimized too conform to the principle of client-supplier that eases the maintenance. Bottleneck is another metric that counts the number of incoming and outgoing dependencies of the clusters (e.g., [9]). Clustering algorithms aim at minimizing this metric to facilitate the maintenance of the obtained software.

*Size-Complexity* metrics are also commonly used (*e.g.*, [9]). For many clustering problems, the size and/or complexity of the clusters should be controlled for maintenance considerations (*e.g.*, [5]).

In addition to metrics, there are many other clustering evaluation techniques. Some are still related to structural dependencies and others involve information that cannot be extracted only from the code. Examples of these techniques are similarity-based clustering (e.g., [3]), and co-change-based clustering (e.g., [15]).

An interesting approach is the one of [22] where the clustering process uses structural (dependencies) and non structural information (ownership, location). To this end, the authors define a clustering approach based on Information Theory.

As it is conjectured for our work, examples could be used to evaluate the quality of clustering solutions. The idea is to compare the inputs and outputs of the clustering with those of known cases. This was done in [10] where a refactoring process is performed to improve the quality of an OO program. In this work, an objective function measures a distance between the OO metrics of the refactored program and those of an example program. As we will see in Section 3, our approach is different as we do not use metrics to assess the similarity with examples. We rather use the structural similarity between the program to cluster and the examples. Moreover, we do not use a single example but a set of examples.

### 2.3 Algorithms

As stated in Section 2.1, the majority of software clustering problems is modeled as a graph-partitioning problem. Consequently, it is difficult to obtain an optimal solution using an exhaustive search method. This observation motivates the use of heuristic-search methods. When looking to the existing literature, a wide variety of algorithms were used: Hill climbing and multi-hill climbing, Genetic algorithms, Simulated Annealing, Tabu search, etc.

In some contributions, deterministic clustering algorithms were used such as hierarchical clustering (*e.g.,* [4]), and formal concept analysis (*e.g.,*[5] and [21]). Hierarchical clustering algorithm usually is sensitive to the order in which the elements are processed. Some heuristics are used to reduce this sensitivity. Formal concept analysis produces a set of clusters but not a partition. A post-processing using heuristics is necessary to derive a partition.

### 2.4 Summary

As mentioned previously, the main limitations of existing approaches to software clustering is the definition of a function that evaluates the quality of the obtained abstractions. In many problems, the clusters (abstractions) should have a meaning that is beyond the structural proximity of the basic elements that compose them. For example, a class implements a family of objects of the real world. Similarly, a component implements a function or a set of functions required in an application. Relying only on structure proximity when defining the clusters, cannot guarantee that the obtained classes or components are meaningful from the application domain standpoint. In the following section, we propose a complementary way to guide the clustering process. Rather than the structural proximity alone, we use the similarity with past valid clustering examples.

## 3 Software Clustering by Example

In Section 2 we have shown that, following the structural-semantics-equivalence hypothesis, almost all the approaches aims at minimizing or maximizing an objective function defined to measure structural dependencies inside and between clusters starting from a dependency graph.

In this section we present our approach which is based on the hypothesis that examples can help recovering part of the semantic proximity. The idea behind our clustering process is to compare configurations of groups of elements to configurations in an example base that led to abstract entities. Roughly speaking, suppose in a system $A$ known as having a good modularization, a group of classes in a module $M_A$ are connected, by different types of relationships, following a certain configuration. If in a system $B$ to modularize, we find a group of classes $M_B$ that follows (almost) the same configuration as $M_A$, then chances are that $M_B$ forms a good module. Classes in both configurations do not need to have the same names.

### 3.1 Modeling the Base of Examples

We define a base of examples as a set $BE$ of $n$ pairs $s_e = \{G_e, P_{Ge}\}$ of already clustered software represented by a dependency graph $G_e$ and its corresponding partition $P_{Ge}$, formally:

$$BE = \{s_e | s_e = \{G_e, P_{Ge}\}, 1 \leq e \leq n\}$$

In opposition to case-based reasoning approaches, our goal is not to find a complete system similar to the one we try to cluster. Therefore, the example base is used as set of clusters $EK$ coming from different systems.

$$EK = \left\{Q_j \in \bigcup P_{Ge} \middle| s_e \in BE\right\}$$

For the sake of clarity we refer to clusters in the example base by $Q$ to dissociate them from those we seek to define, and for which we use $K$.

## 3.2 Deriving a Partition

Software elements represented by vertexes in $V$ of $G(V,E)$ should be grouped together by similarity with cluster examples $Q_j \in EK$. To this end, we define a function $ave: V \rightarrow KE$ that assigns an example cluster to each element in the graph to partition. For example, $ave(v_1) = Q_{13}$ means that the vertex $v_1 \in V$ was assigned the cluster $Q_{13} \in EK$. The nature of the graph $G$ and the function $ave$ depend on the clustering problem.

The clustering process produces a partition $P_G$ of the graph $G(V,E)$ with the principle that vertexes with the same assigned cluster $Q_i \in EK$ according to $ave$ form a cluster $K_i \in P_G$. Formally:

$$K_i = \{v_j \in V | \, ave(v_j) = Q_i\}$$

## 3.3 Evaluating a Partition

The objective function $f$ is defined in terms of similarity between the groups of elements of the system to partition with the associated cluster examples. It could be calculated as the weighted average of similarities of these groups. To give equal chances to each group, the similarity is normalized by the size of the groups.

$$f = \frac{\sum_{i=1}^{k} |K_i| Sim(K_i, Q_j)}{|V|} \in [0,1]$$

The similarity $Sim$ between a candidate cluster $K_i$ and an example cluster $Q_j$ is defined as a function of the similarities between their respective elements. It is a variation of the graph matching function defined in [19]. Formally,

$$Sim(K_i, Q_j) = \frac{1}{|K_i|} \sum_{v \in K_i} \max_{q \in Q_j} vSim(v,q) \in [0,1]$$

The function $vSim(v,q)$ compares a vertex $v \in V_i$ of $K_i$ to a vertex $q \in V_j$ of $Q_j$. $vSim(v,q)$ equals zero if the types of $v$ and $q$ are different, that is, if $v$ and $q$ are not comparables. Otherwise, $vSim(v,q)$ will match the edges $EV(v)$ of $v$ and $EV(q)$ of $q$ and will return the ratio of matched edges over the total edges of $v$ and $q$. In directed graphs, $EV(v)$ includes both incoming and outgoing edges.

$$EV(v) =$$
$$\{e \in E | e = (u,v) \vee e = (v,u), u \in V, v \in V\}$$

The edge $e \in EV(v)$ and an edge $g \in EV(q)$ match if they satisfy all the following conditions:

- both are of the same type
- both $e = (u,v)$ and $g = (p,q)$ are internal (respectively external) edges.
- both $e$ and $g$ are incoming (respectively outgoing) edges of respectively $v$ and $q$.
- Neither $e$ nor $g$ have been matched yet with another edge.

To illustrate $vSim(v,q)$ let us consider the vertexes $v$ and $q$ in Figure 1. For simplicity reasons, we suppose that $v$ and $q$ are of the same type and that the edges of $v$ and $q$ (the ones in bold) are of the same type as well.

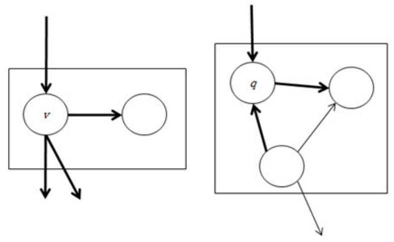


Figure 1: Example of vertexes to calculate vSim(v, q)

In this example, $v$ has 4 edges ($EV(v) = 4$), $q$ has 3 edges ($EV(q) = 3$). Both $v$ and $q$ have one incoming edge from the outside of their clusters, call them $e_{v1}$ and $e_{q1}$. Then, $match(e_{v1}, e_{q1}) = match(e_{q1}, e_{v1}) = 1$. Both $v$ and $q$ have one outgoing edge inside their respective clusters ($e_{v2}$ and $e_{q2}$). This gives $match(e_{v2}, e_{q2}) = match(e_{q2}, e_{v2}) = 1$. Consequently,

$$vSim(v,q) = \frac{2+2}{4+3} = \frac{4}{7}$$

To calculate $Sim(K_i, Q_j)$, we create a matrix that contains $vSim$ values for all pairs of vertexes. Then, we match the pairs to maximize the global similarity as shown in the example of Table 1.

| | | $Q_2$ | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **q3** | **q4** | **q5** | **q11** | **q13** | **q16** | **Max** |
| $K_1$ | **v1** | 0.83 | 0.40 | 0.90 | 0.00 | 0.00 | 0.00 | 0.90 |
| | **v3** | 0.50 | 0.58 | 0.25 | 0.00 | 0.00 | 0.00 | 0.50 |
| | **v4** | 0.25 | 0.83 | 0.50 | 0.00 | 0.00 | 0.00 | 0.83 |
| | **v6** | 0.00 | 0.00 | 0.00 | 0.67 | 0.50 | 0.33 | 0.67 |
| | **v9** | 0.00 | 0.00 | 0.00 | 0.67 | 0.50 | 0.33 | 0.50 |
| | **v11** | 0.00 | 0.00 | 0.00 | 0.67 | 0.50 | 0.33 | 0.33 |
| | | | | | | | **Sum** | 3.73 |

Table 1: Vertex-Similarity Matrix of $K_1$ and $Q_2$.

This matrix was build to calculate $Sim(K_1, Q_2)$. As the number possibilities is small, all the possible mappings are evaluated. In our case, the best mapping is the one that matches $v_1$ to $q_5$ ($vSim$ 0.9), $v_3$ to $q_3$ ($vSim$ 0.5), $v_4$ to $q_4$ ($vSim$ 0.83), etc. This gives

$$Sim(K_1, Q_2) = \frac{1}{|K_1|} \sum_{v \in K_1} \max_{q \in Q_2} vSim(v,q) = \frac{3.73}{6}$$
$$= 0.62$$

In the next section, we present an illustrative application of our approach for the well-known problem of object identification in procedural code.

## 4 Application to Object Identification

In Section 3 we have presented our new formulation of software clustering problems. In this section, we show how this formulation could apply to a specific clustering problem, namely, object identification in procedural code. The intuition behind many solutions to this problem is that if a subset of procedures accesses the same variables, this is an indication that the variables define the state of an object and the procedures its behavior (e.g., [5]). We still use the same intuition, but we change the way we evaluate the nature and the strength of the dependencies between variables and procedures.

### 4.1 Modeling Procedural Code as a Graph

The software to be clustered is defined as a graph $G(V,E)$ where the vertexes are procedures and variables and the edges procedure calls and variable accesses. According to the framework of Section 2, vertex and edge types are respectively

$$T_V = \{procedure, variable\}$$
$$\text{and } T_E = \{call, use\}$$

In this illustration, we use unweighted edges, although call and use edges could be weighted by respectively the number of calls between two procedures and the number of accesses between a procedure and a variable.

The base of example $BE$ is defined according to the framework of Section 3. It contains a set of procedural programs $\{G_e, P_{Ge}\}$ represented as graphs and the corresponding partitions. Each sub-graph of a partition $P_{Ge}$ of a program graph $G_e$ has been previously tagged as an actual object of the application domain of the considered program.

When using the example base $BE$ to identify the set of objects $P_{Gc}$ of a program $c$ represented by a graph $G_c$, the quality of $P_{Ge}$ is also evaluated according to the framework of Section 3. Functions $Sim(K_i, Q_j)$ and $vSim(v,q)$ match separately variables and procedures one the one hand, and consider both types of edges for $call$ and $use$.

### 4.2 Finding Objects by Partitioning the Graph

The space of all the possible partitions is very large for average procedural programs. To explore this space, a heuristic search is suited. In this illustration we use a hybrid method that combines Particle Swarm Optimization (PSO) [7] and Simulated Annealing (SA) [15]. First, we perform a global heuristic search by PSO to reduce the search space and select an initial solution. Then, to refine this solution, a local heuristic search is done using SA [20]. In the remainder of this subsection we describe briefly PSO and SA, and then, detail our adaptations to the specific problem of object identification.

As usual, the adaptations concern the encoding of solutions, the solution-change operators, and the definition of the fitness function.

### 4.2.1 Particle Swarm Optimization (PSO)

PSO [7] is a parallel population-based computation technique. The PSO swarm (population) is represented by a set of $n$ particles (possible solutions to the problem). A particle $i$ is defined by a position coordinate vector $X_i$, in the solution space. Particles improve themselves by changing positions according to a velocity function. The improvement is assessed by a fitness function. The best position of each particle ($pbest$) is stored and so is stored de best position ever found by the particles in the swarm ($gbest$). At each iteration, all particles are moved according to their velocities (Equation 4). The velocity $V_i'$ of a particle $i$, given by Equation (5), depends on its inertia, i.e., previous velocity $V_i$, its $pbest_i$, and the $gbest$. These factors are weighted respectively by parameters $W$, $C_1$, and $C_2$. $pbest_i$ and $gbest$ are not systematically considered. Random numbers (between 0 and 1) are uniformly generated to determine to which extended a particles will take into account these two positions.

$$X_i' = X_i + V_i' \qquad (4)$$

$$V_i' = W \times V_i + C_1 \times rand_1(\,) \times (pbest_i - X_i) + C_2 \times rand_2(\,) \times (gbest - X_i) \qquad (5)$$

The algorithm iterates until the particles converge towards a unique position that determines a suboptimal solution to the problem.

### 4.2.2 Simulated Annealing (SA)

SA [15] is a local search algorithm that gradually transforms a solution following the annealing principle used in metallurgy. Starting from an initial solution, SA uses a pseudo-cooling process where a pseudo temperature is decreased gradually. For each temperature step, three operations are repeated for a fixed number of iterations: (1) determine a new neighboring solution, (2) evaluate the fitness of the new solution and (3) decide on whether to accept the new solution in place of the current one based on the fitness function and the temperature value. Solutions are accepted if they improve quality. When the quality is degraded, they still can be accepted, but with a certain probability. The acceptance probability is high when the temperature is high and the quality degradation is low. As a consequence, quality-degrading solutions are easily accepted in the beginning of process when the temperatures are high, but with less probability as the temperature decreases. This mechanism prevents from reaching a local optimum.

### 4.2.3 Solution-Coding Adaptation

The efficiency of applying a search-based method to a particular problem relies heavily on how potential problem solutions are coded into an appropriate representation that can be manipulated by the method. We model the search space as an n-dimensional space where each dimension corresponds to one of the composing element of the software to partition into objects. A solution is then a point in that space defined by a coordinate vector whose elements are the numbers of the example clusters in $BE$. Each solution vector is the result of the assignment function $ave$ defined in Section 3.2. When generating the initial population, for each solution, $ave$ assign randomly a cluster in the example base to each element (vector coordinate) to each element of the program to cluster. Later in the clustering process, this assignments are modified using solution-change operators.

To show how a possible solution is coded, let us consider a program with eight composing elements (four variables and four procedures). This program defines an 8-dimensional space where each possible solution is apposition vector of eight coordinates. One solution in this space, shown in Figure 2, proposes to group the composing elements into three objects. It suggests, for example, that the two variables ($v_0$ and $v_2$), and two procedures ($p_4$ and $p_6$) should be grouped according to example cluster $Q_2$.

This solutions coding is used by both search methods PSO and SA.

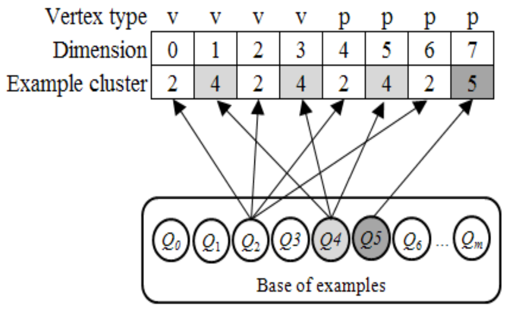


Figure 2: Example of a Solution Representation.

### 4.2.4 Change Operators

Modifying solutions to produce new ones is the second important aspect of the heuristic search. Unlike coding, solution change is implemented differently for PSO and SA. While PSO sees a change as a movement in the search space driven by a velocity function, SA considers it as random coordinate modifications.

In the case of PSO, a translation (velocity) vector is derived according to Equation (5) and added to the position vector (see the example of Figure 3). Each dimension of the position vector is added to the corresponding dimension of the velocity vector. These additions consider two constraints. First, the velocity vector contains real values. When added to the coordinate values, they produce real coordinates that do not correspond to the numbers of the cluster examples. In this case, the obtained real value is rounded to the closest integer value (see for example dimension 1 where velocity 2.9, added to coordinate 2, gives a value of 4.9 changed to coordinate 5). The second constraint is related to the number of clusters in the example base $n$. To derive coordinates corresponding to valid cluster numbers, the obtained coordinates are transformed according to arithmetic modulo $n$.

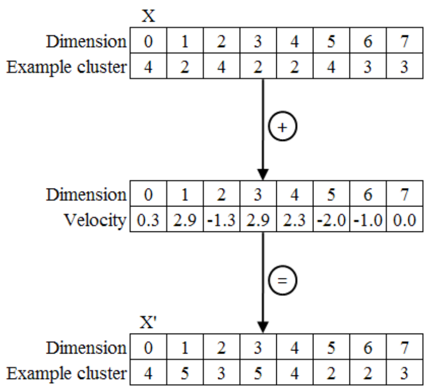

X
Dimension 0 1 2 3 4 5 6 7
Example cluster 4 2 4 2 2 4 3 3
+
Dimension 0 1 2 3 4 5 6 7
Velocity 0.3 2.9 -1.3 2.9 2.3 -2.0 -1.0 0.0
=
X'
Dimension 0 1 2 3 4 5 6 7
Example cluster 4 5 3 5 4 2 2 3

Figure 3: An Illustration of the Change Operator in PSO.

For SA, the change operator involves randomly choosing $y$ dimensions $(y < n)$ and replacing their assigned clusters by randomly selected ones form the base of examples. For instance, Figure 4 shows an example of this change. Elements of dimensions 2, 4, and 7 are selected to be changed. They are assigned to $Q_3$ in place of, respectively, $Q_2$, $Q_4$, and $Q_5$. The other elements keep their assigned clusters. The number of dimensions to change is a parameter of the SA algorithm (three in our example). In our validation, this parameter is randomly chosen at each step between 1 and $\frac{n}{2}$.

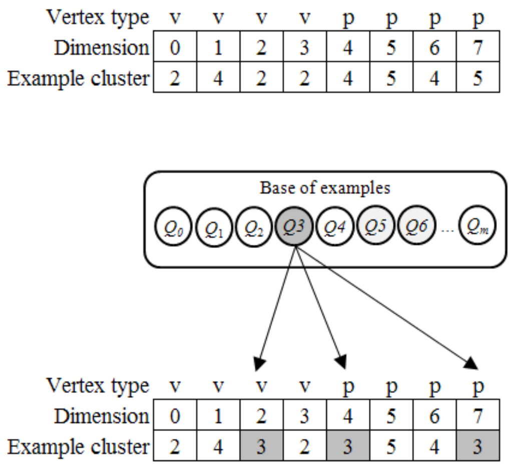

Vertex type v v v v p p p p
Dimension 0 1 2 3 4 5 6 7
Example cluster 2 4 2 2 4 5 4 5
Base of examples
$Q_0$ $Q_1$ $Q_2$ $Q_3$ $Q_4$ $Q_5$ $Q_6$ ... $Q_m$
Vertex type v v v v p p p p
Dimension 0 1 2 3 4 5 6 7
Example cluster 2 4 3 2 3 5 4 3

Figure 4: An Illustration of the Change Operator in SA.

## 4.3 A Case Study

To evaluate the efficiency of our formulation of the SC problems in the particular case of object identification, we conducted a case study.

We selected 10 C programs from the website Planet Source Code[1]. The size of these programs in terms of composing elements (variables and procedures) varies from 18 to 105, with an average of 57. For each program, after reading the documentation and exploring the code, we identified manually the objects in terms of groups of variables and procedures. Each program was parsed and a corresponding graph was constructed. We grouped for the 10 programs the graphs with their corresponding partitions, *i.e.,* manually identified objects, defining 10 identification examples. All together, the 10 examples define 49 objects (cluster examples).

To measure the correctness of our object identification, we used a 10-fold cross-validation procedure. For each fold, we identify the objects in one program and use the 9 other programs as the base of examples. The objects identified au-

[1] www.planet-source-code.com

tomatically are compared to those found manually. The correctness for each fold is calculated as the proportion of composing elements that are assigned to good objects. The global correctness is derived as the average of the 10 fold's correctness values.

To set the parameters of PSO and SA algorithm, we took values commonly found in the literature [12].

Table 2 shows the correctness values obtained for each of the 10 folds when using the hybrid PSO-SA research. Correctness varies between 66% and 100% depending on the programs. The identification in small programs is correctly done (100%). It was good for large programs (around 80%). Programs with less good values are the average ones. 7 over the 10 programs have identification correctness greater than 80%.

| PC | Num. of elements | Fitness | Correctness |
|---|---|---|---|
| PC 1 | 18 | 0,92 | 100% |
| PC 2 | 22 | 0,89 | 100% |
| PC 3 | 25 | 0,91 | 100% |
| PC 4 | 44 | 0,86 | 86% |
| PC 5 | 59 | 0,89 | 78% |
| PC 6 | 62 | 0,74 | 68% |
| PC 7 | 65 | 0,78 | 66% |
| PC 8 | 76 | 0,89 | 79% |
| PC 9 | 90 | 0,85 | 83% |
| PC 10 | 105 | 0,88 | 89% |
| **Avg.** | **57** | **0.82** | **85%** |

Table 2: 10-Fold Cross Validation Results.

In addition to the PSO-SA, we performed the identification with only PSO (with more iterations and a larger population) to evaluate if hybrid search is useful compared to single-method search. As shown in Figure 5, hybrid search with PSO-SA gives better results for larger programs compared to search with PSO only. The difference for the larger program was 0.34 (0.89 compared to 0.55). For small programs both strategies give the same results. For the average ones PSO alone was a better option. This indicates that hybrid strategies that combine global and local search are useful when the search space is large.

After evaluating the efficiency of our formulation, the second question is to compare it to methods using the classical coupling-cohesion formulation of the clustering problems. Starting from the approach proposed in [4], we implemented a genetic-based detection algorithm that aims at minimizing coupling and maximizing cohesion in the partition solutions. In this algorithm we used two variants of the fitness function corresponding to two ways of combining coupling and cohesion criteria. In the first variant, call it ACf1, the quality of a solution is measured as the ratio between normalized cohesion and coupling. Formally

$$fitness_1(P_G) = \frac{COH(P_G)}{COU(P_G)}$$

In the second variant, ACf2, the fitness is calculated as the average of the cohesion and the inverse of coupling. Formally

$$fitness_2(P_G) = \frac{\left(COH(P_G) + \frac{1}{COU(P_G)}\right)}{2}$$

Both variants give a high fitness when the cohesion is high and the coupling low.

Table 2 shows the comparative results between our example-based identification (PSO-SA) and the coupling-cohesion-based identification (ACf1 and ACf2). For all the 10 programs, the correctness is significantly better with our new formulation than with the classical one. The difference is even more important for larger programs. This is a clear indication that similarity with previous examples, combined with the structural dependencies, could improve the quality of the clustering approaches in SC problems.

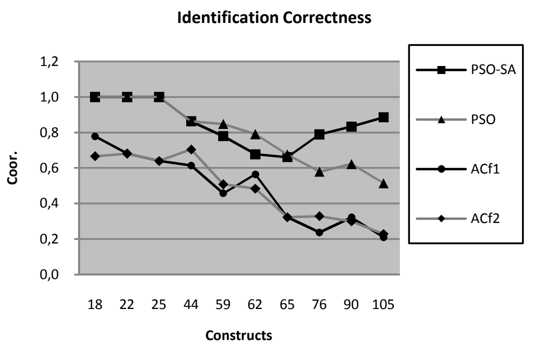


Figure 5: Effect of Search Strategy, Program size, and Alternative clustering Methods on Correctness.

## 5 Conclusions

Software clustering offers promising solutions to many maintenance problems such as software re-architecture and migration. Software clustering consists in identify groups of software elements that may correspond to more abstract artifacts. To guide this identification, objective functions are proposed with the principle of approximating semantic information from the structural dependencies between elements.

In this paper, we first propose a general formulation of the SC problems. Then, we propose a novel formulation in which the clustering objective functions are defined in terms of similarity with previous clustering examples rather than only the structural proximity. Our formulation is general and could be used for various SC problems. In this paper, we illustrate it with the well-known problem of object identification.

In this context, we performed a case study to evaluate the example-based object identification. The study targeted 10 C programs taken from *Planet Source Code* website. In an initial phase, we identified manually the objects in those programs and created a base of examples accordingly. Then, we used a 10-fold cross-validation procedure where each program was clustered using the 9 others.

Although our experimental results are very encouraging and indicate a clear trend in favor of our proposal, some limitations are worth noting. One important problem that limits the applicability of our approach is the need for an example base. This base could be difficult to obtain for migration problems that involve two paradigms (procedural to object or object to components). This is because two versions (one for each paradigm) of the same program should be aligned. Hopefully, for single-paradigm problems such as re-architeture and modularization, there are many programs with refactored architectures that can be used. In this context, we are adapting and evaluating our approach on the re-packaging problem.

## References

[1] G. Canfora, A. Cimitile, M. Munro and M. Tortorella, "Experiments in identifying reusable abstract data types in program code", in *Workshop on Program Comprehension*, pp. 36-45, 1993.

[2] G. Czibula and G. Şerban, "Improving systems design using a clustering approach", *International Journal of Computer Science and Network Security*, vol. 6, no. 12, pp. 40-49, 2006.

[3] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, "Automatic package coupling and cycle minimization", in *Working Conference on Reverse Engineering*, pp. 103-112, 2009.

[4] H. Sahraoui, P. Valtchev, I. Konkobo, and S. Shen, "Object identification in legacy code as a grouping problem", in *International Computer Software and Applications Conference*, pp. 689-696, 2002.

[5] H. Sahraoui, W. Melo, H. Lounis, and F. Dumont, "Applying concept formation methods to object identification in procedural code", in *International Conference on Automated Software Engineering*, pp. 210-218, 1997.

[6] H. Washizaki and Y. Fukazawa. "A technique for automatic component extraction from object-oriented programs by refactoring", in *Science of Computer Programming*, vol. 56, no.1-2, pp. 99-116, 2005.

[7] J. Kennedy and R. Eberhart, "Particle swarm optimization", in *International Conference on Neural Networks*, vol. 4, pp. 1942-1948, 1995.

[8] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level", in *Conference on Genetic and Evolutionary Computation*, pp. 1106-1113, 2007.

[9] M. Harman, R. Hierons, and M. Proctor, "A new representation and crossover operator for search-based optimization of software modularization", in *Genetic and Evolutionary Computation Conference*, pp. 1351-1358, 2002.

[10] M. O'Keefee and Mel Ó Cinnéide. "Automated design improvement by example", in *Conference on New Trends in Software Methodologies, Tools and Techniques*, pp. 315 - 329, 2007.

[11] M. Siff and T. Reps, "Identifying modules via concept analysis", in *Transactions on Software Engineering*, vol. 25, no. 6, pp. 749-768, 1999.

[12] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler, "Towards model transformation generation by-example", in *International Conference on System Sciences*, pp. 285b, 2007.

[13] O. Seng, M. Bauer, M. Biehl, and G. Pache, "Search-based improvement of subsystem decompositions", in *Conference on Genetic and Evolutionary Computation*, pp. 1045–1051, 2005.

[14] S. Allier, H. Sahraoui, S. Sadou, and S. Vaucher, "Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces", in *International Symposium on Component Based Software Engineering*, pp. 216-231, 2010.

[15] S. Kirkpatrick, "Optimization by simulated annealing: quantitative studies", in *Journal Of Statistical Physics*, vol. 34, no. 5, pp. 975-986, 1984.

[16] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using automatic clustering to produce high-level system organizations of source code", in *International Workshop on Program Comprehension*, pp. 45-53, 1998.

[17] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures", in *International Conference on Software Maintenance*, pp. 50-59, 1999.

[18] S. Mitchell, *"A heuristic search approach to solving the software clustering problem"*, Dissertation, Drexel University, Philadelphia, PA, United States, 2002.

[19] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. Van Dooren, "A measure of similarity between graph vertexes: Applications to synonym extraction and web searching", *Society for Industrial and Applied Mathematics*, vol. 46, no. 4, pp. 647-666, 2004.

[20] V. Kelner, F. Capitanescu, O. Léonard, and L. Wehenkel, "A hybrid optimization technique coupling an evolutionary and a local search algorithm", *in Journal Of Computational And Applied Mathematics*, vol. 215, no.2, pp. 448-456, 2008.

[21] V. Van Deursen and T. Kuipers, "Identifying objects using cluster and concept analysis", in *International Conference on Software Engineering*, pp. 246–255, 1999.

[22] P. Andritsos and V. Tzerpos, "Software clustering based on information loss minimization.", *in 10th Working Conference on Reverse Engineering*, pp. 334-344, 2003.

# CHAPTER 5

# GENETIC-PROGRAMMING APPROACH TO LEARN MODEL TRANSFORMATION RULES FROM EXAMPLES

In this chapter we present our second contribution [22] entitled "Genetic-Programming Approach to Learn Model Transformation Rules from Examples" published at "Theory and practice of Model Transformations", 2013, pages 17-32. The contribution aims to improve automation in MDE. Specifically, the approach derives a model transformation. The derivation process is based on optimization search (GP) guided by conformance with examples of past transformations (pairs of source and target models). As opposed to other approaches, ours does not need transformation traces, and thus, is easier to use. Our approach can derive full operational n-to-m transformation rules that can have complex conditions. The validation on two transformation problems showed that non-trivial rules could be derived. This contribution is the follow-up of [20], which was presented in ASE 2012.

# Genetic-Programming Approach to Learn Model Transformation Rules from Examples

Martin Faunes[1], Houari Sahraoui[1], and Mounir Boukadoum[2]

[1] DIRO, Université de Montréal, Canada
[2] Université du Québec à Montréal, Canada

**Abstract.** We propose a genetic programming-based approach to automatically learn model transformation rules from prior transformation pairs of source-target models used as examples. Unlike current approaches, ours does not need fine-grained transformation traces to produce many-to-many rules. This makes it applicable to a wider spectrum of transformation problems. Since the learned rules are produced directly in an actual transformation language, they can be easily tested, improved and reused. The proposed approach was successfully evaluated on well-known transformation problems that highlight three modeling aspects: structure, time constraints, and nesting.

## 1 Introduction

The adoption of new technologies generally follows a recurrent cycle described by Moore in [16]. In this cycle, user categories adopt a technology at different moments depending on their profiles and the technology's maturity. Moore identified the move from the *early adopters* category to the *early majority* category as the gap that is the most difficult to cross and in which many technologies spend a long time or just fail. Model Driven Engineering (MDE), as a new technology that changes considerably the way we develop software, does not escape this observation. MDE received much attention in recent years due to its promise to reduce the complexity of the development and maintenance of software applications. However, and notwithstanding the success stories reported in the past decade, MDE is still at the early-adopters stage [15]. As mentioned by Selic[1], in addition to the economic and cultural factors, the technical factors, particularly the difficulty of automation, represent major obstacles for MDE's adoption.

Automation is a keystone and a founding principle of the MDE paradigm. According to Schmidt, MDE technologies combine domain-specific modeling languages with transformation engines and generators to produce various software artifacts [21]. By automating model-to-model and model-to-code transformations, MDE fills the conceptual gap between source code and models, and ensures that models are up to date with regards to the code and other models. In recent years, considerable advances have been made in modeling environments

[1] Bran Selic, "The Embarrassing Truth About Software and Automation and What Should be Done About It", Keynote talk, ASE 2007.

and tools. However, in practice, automated model transformation and code generation has been restricted to niche areas such as database mapping and data-intensive-application generation [15]. To address this limitation, a large effort has been made to define languages for expressing transformation rules (e.g., ATL [9]) to make the writing of transformation programs easier.

Having a good transformation language is only one part of the solution; the most important part is to define/gather knowledge about how to transform any model conforming to a particular metamodel into a model conforming to another metamodel. For many problems, this knowledge is incomplete or not available. The difficulty of writing transformation rules is the main motivation behind the research on learning transformation rules from examples. Although the idea goes back to the early nineties, the first concrete work on Model Transformation by Example (MTBE) was proposed by Varro in 2006 [24]. MTBE's objective was to derive transformation programs by generalizing concrete transformations found in a set of prototypical examples of source and target models. Since then, many approaches have been proposed to derive the transformation rules (*e.g.,* [22,1,6,4,12,20]) or to transform a model by analogy with transformed examples [10].

Still, the existing MTBE approaches only solve the problem of rule derivation partially. Most of them require detailed mappings (transformation traces) between the source and target model examples [1], which are difficult to provide in some situations; others cannot derive rules that test many constructs in the source model and/or produce many construct in the target model, many-to-many rules [22], a requirement in complex transformation problems. A third limitation is the inability of some approaches to automatically produce complex rule conditions to define precise patterns to search for in the source model [20]. Finally, some approaches produce abstract, non-executable rules that have to be completed and mapped manually to an executable language [4].

In a previous work [5], we proposed a preliminary approach for the derivation of complex and executable rules from examples without the need of transformation traces. The approach was inspired from genetic programming (GP) and exploits GP's ability to evolve programs in order to improve their capacity to approximate a behavior defined by a set of valid pairs of inputs/outputs. The approach was quantitatively evaluated on the transformation of class diagrams to relational schemas. Although 75% of the model constructs were correctly transformed, many key transformation rules were not derived or only derived partially. In this paper, we propose an improved version of the algorithm with new ways of solution initialization, new program derivation from existing ones, and program evaluation. This new version is evaluated on two transformation problems that cover three important software modeling characteristics: structure, time constraints, and nesting. In the first problem, the transformation of class diagrams to relational schemas, we test the ability of our approach to handle the transformation of structural models. Time-constrained-model transformation is considered in the second case study through the problem of sequence diagrams to state charts. In this problem, the derived transformation should preserve the time constraints between the constructs. Our second case study also handles the complex problem of nested-sequence-diagrams to state-charts transformation. In this case, the transformation control is non trivial as the rules should transform the nested elements before those that contain them. The obtained quantitative and qualitative results show that our approach allows the derivation the correct transformation rules for both problems.

## 2 Learning Rules from Examples

Our goal is to define a transformation-rule derivation process that may apply to a wide range of transformation problems. To this end, our approach should work even if fine-grained transformation traces are not available. Additionally, constraints on the shape or size of the rules should be as limited as possible. This includes the numbers of source and target-construct types and the nature of rule conditions. Finally, the produced rule sets must be executable without a manual refinement step.

### 2.1 Rule Derivation as an Evolutionary Process

Transformation rules are programs that analyze certain aspects of source models given as input and synthesize the corresponding target models as output [21]. Learning complex and dynamic structures such as programs is not an easy task [2]. Of the possible tools that can be used for automatic programs generation, Genetic Programming (GP) [13] is a strong contender for supremacy as it was originally created for the purpose. This motivated our investigation of GP to automatically derive rule sets, *i.e.*, declarative programs, using examples of models transformations, *i.e.*, complex inputs/outputs. GP draws inspiration from Darwinian evolution and aims to automatically derive a program to solve a given problem, starting from some indications about how the problem should be solved. These usually take the form of input and output examples, and the derivation process is done by iteratively improving an initial population of randomly-created programs, *i.e.*, by keeping the fittest programs for reproduction at each step, the reproduction being made by means of *genetic operators* similar to those observed in nature. The typical GP cycle is sketched in Figure 1.

Before, starting the evolution process, the user must have a set of example pairs describing the expected program behavior in the form of <input, output>. The user must also define a way to encode and create the initial population of random programs. Finally, a mechanism is needed to run the programs on the provided inputs and compare the execution results with the expected outputs. This is typically done by defining a fitness function that evaluates the closeness between the produced and expected outputs.

To apply GP to the MTBE problem, we have to consider two issues. First, transformation rules are not imperative programs and cannot be encoded as trees as usually done in GP [13]; second, the outputs of transformations are models (usually graphs) that are not easy to compare for evaluating the correctness of

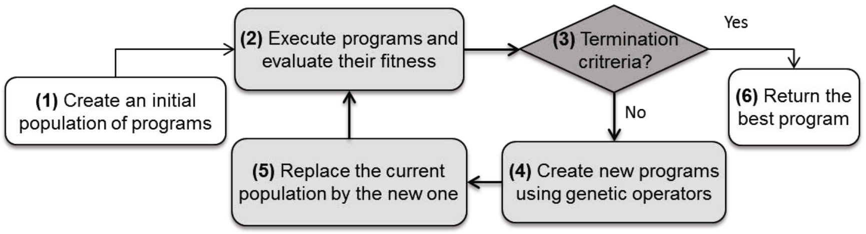

(1) Create an initial population of programs
(2) Execute programs and evaluate their fitness
(3) Termination critreria?
Yes
No
(4) Create new programs using genetic operators
(5) Replace the current population by the new one
(6) Return the best program

**Fig. 1.** A typical GP cycle

a program. In the following subsections, we detail our adaptation of the GP algorithm to the specific problem of MTBE. Note that, for our investigation, we decided to use a simple metamodeling language to describe the metamodels and a generic rule language/engine JESS [8] for the writing and execution of transformation rules. This decision was made to separate, in a first phase of this research project, the intrinsic complexity of MTBE from the accidental complexity of conformance to standards and interoperability concerns. The mapping between JESS and a transformation language such as ATL is pretty easy to perform since both languages offer similar features such as declarative and imperative structures as well as control mechanisms.

## 2.2 Encoding Rule Sets

Typical transformation problems require a set of transformation rules to cover all the patterns in the source models. A program $p$ is accordingly encoded as a set of transformation rules, $p = \{r_1, r_2, ..., r_n\}$. Each transformation rule $r_i$ is in turn encoded as a pair $r_i = (SP, TP)$, where $SP$ is the pattern to search for in the source model and $TP$ is the pattern to instantiate when producing the target model.

*Source Pattern.* A source pattern $SP$ is a pair $SP = (SGC, G)$, in which $SGC$ is a set of generic source constructs and $G$ is a guard. A generic source construct is the specification of an instance of a construct type that has to be matched with concrete constructs in the source model. For example, in the rule of Listing 1.1, $SGC = \{C, A, S\}$, where $C$, $A$ and $S$ represent respectively a class, an attribute, and an association. $SGC$ could include more than one generic construct from the same construct type, *e.g.*, two classes and an association. Each generic construct has the properties of its construct type in the source metamodel. When matched with a concrete construct from the source model, these properties take the values of the latter. For instance, an attribute $A$ has its name (descriptive property) and the name of the class it belongs to (join property) as properties. During execution, the value of a property can be accessed as shown in Listing 1.1, *e.g.*, $A.name$ and $A.class$.

The guard $G$ contains two types of conditions: join conditions and state conditions. Join properties are used to define the set of join conditions, which allow to specify a source pattern as a model fragment, *i.e.*, a set of interrelated constructs according to the metamodel. For example, in the rule of Listing 1.1, the join condition $A.class = C.name$ states that $A$ should be an attribute of class $C$ whereas $S.classFrom = C.name$ restricts the pattern to only classes that are at the origin of associations.

**Listing 1.1.** Rule encoding example

```
Source pattern:
//  Generic source element
     Class C, Attribute A, Association S
//  Guard: Join condition
     (and (A.class = C.name) (S.classFrom = C.name))
//  Guard: State condition
     (and (S.maxCardFrom < 1) (S.maxCardTo   > 1))
Target pattern:
//  Generic target element
     Table T, Column O
//  Bindings
     T.name := C.name
     O.name := A.name
//  Join-statement
     O.table = T.name
```

State conditions involve the properties of the generic source constructs (both join and descriptive ones). They are encoded as a binary tree containing elements from terminal ($T$) and primitive ($I$) sets. $T$ is the union of the properties of the constructs in $SGC$ and a set of constants $C$. For the rule of Listing 1.1, the properties are *C.name*, *A.name*, *A.class*, *S.classFrom*, $S.classTo$, $S.MaxCardFr$, $S.MaxCardTo$, etc. As the properties are numbers and strings, numeric and string constants such as $\{0, 1, Empty, ...\}$ are added to the terminals. As conditions are manipulated, the Boolean constants *true* and $false$ are also added. The set of primitives $I$ is composed minimally of logical operators and comparators ($I = \{and, or, not, =, >, <, ...\}$). Other operators, such as arithmetic or string operators, could be added to test values derived from the basic properties. Since this work uses the concrete rule language JESS [8], the conceptual distinction between join and state conditions is not reflected in the actual code. Both types of conditions form the condition tree with terminals as leaf nodes and primitives as the other nodes. A rule without any condition will be represented by a tree with the single node “true”. A rule is fired for any combination of instances for which the condition tree is true.

*Target Pattern.* The target pattern $TP$ is a triple $TP = (TGC, B, TJ)$, where $TGC$, $B$ and $TJ$ represent respectively a set of generic target constructs, a set of binding statements, and a set of join statements. A generic target construct specifies a concrete construct to create in the target model when the rule is fired. In the example of Listing 1.1, two constructs are created: a table $T$ and a column $O$. The set of bindings $B$ determines how to set the property values of the created constructs with the property values of the constructs that match the source pattern. In Listing 1.1, the created table and column will respectively have the same names as the selected class and attribute. Finally, the join statements $TJ$ allow to connect the created constructs to form a fragment in the target model.

In the example provided, column $O$ is assigned to table $T$. The join statements must conform to the target metamodel.

## 2.3 Creating Rule Sets

As stated in Section 2.1, deriving transformation rules using genetic programming requires the creation of an initial population of random rule sets. Each rule set has to be syntactically correct with respect to the rule language (JESS in this work). Moreover, a rule set should be consistent with the source and target metamodels. In this respect, rules should describe valid source and target patterns. For the initial population, a number of rule sets $nrs$ is created ($nrs$ is a parameter of the approach). The number of rules to create for each rule set is selected randomly from a given interval. For each rule, we use a random combination of elementary model fragments (building blocks) to create the source and target patterns. The random combination of building blocks is intended to reduce the size of the search space by considering connected model fragments rather than arbitrary subsets of constructs. For each rule, two combinations are performed respectively over the graphs of the source and target metamodels to create the source and target patterns of the rule, $SP$ and $TP$.

A building block is a minimal model fragment which is self-contained, *i.e.*, its existence does not depend upon the existence of other constructs. For example, in a UML class diagram, a single class could form a building block. However, an attribute should be associated to its class to form a block. Similarly, an inheritance relationship forms with two classes (superclass and subclass) a building block. The determination of the building block for a given metamodel depends only on this latter and not on the transformation of its models.

To create random patterns (source or target), a maximal number of generic constructs $nc$ is first determined randomly. Then, a first building block is randomly selected and included within the pattern. If $nc$ is not reached yet, another building block is selected among those that could be connected to the blocks in the current fragment. Two blocks could be connected if they share at least one generic construct. The connection is made by considering both constructs to connect as the same generic construct. The procedure is repeated until $nc$ is reached. To illustrate the pattern creation procedure, consider the following example. Imagine that the maximum number of constructs is set to four. A first random selection could add to the pattern the block $(ClassC_1, AttributA_1, A_1.class = C_1.name)$ containing two connected generic constructs $C_1$ and $A_1$. As the size of the pattern is less than four, another random selection could add an inheritance block with constructs $InheritanceI_1$, $ClassC_3$, and $ClassC_4$, and links $I_1.class = C_3.name$ and $I_1.super = C_4.name$. One of the two possibilities of connections $((C_1, C_3)$ or $(C_1, C_4))$ is selected, let us say $(C_1, C_4)$. $C_4$ is then replaced by $C_1$ in the pattern including the links.

The last step toward the pattern creation is the random generation of the state conditions (for a source pattern) or the binding statements (for a target pattern). For a source pattern, a tree is created by randomly mixing elements from the terminal set $T$, *i.e.*, properties of the selected constructs and constants consistent with their types, and elements from the primitive set $P$ of operators. The creation of the tree is done using a variation of the“grow” method defined in [13]. In the case of a target pattern, the binding statements are generated by randomly assigning elements in the terminal set $T$ of the source pattern to the properties of the generic constructs of the target pattern that were not set by the join statements (links). The random property-value assignments are done according to the property types.

## 2.4 Deriving New Rule Sets

In GP, a population of programs is evolved and improved by applying genetic operators (mutation and crossover). These operators are specific to the problem to solve. As with the initial-population creation, the genetic operators should guarantee that the derived programs are syntactically and semantically valid. Before applying the genetic operators to produce new programs, programs from the current generation are selected for reproduction depending on their fitness values. For the derivation of transformation rule sets, *roulette-wheel* selection is used. This technique assigns to each rule set a probability of being selected that is proportional to its fitness. This selection strategy favors the fittest rule sets while still giving a chance of being selected to the others. Note that some program could be included directly into the new population, *i.e.*, elitist strategy.

*Crossover.* The crossover operation consists of producing new rule sets by combining the existing genetic material. It is applied with a given probability to each pair of selected rule sets. After selecting two-parent rule sets for reproduction, two new rule sets are created by exchanging parts of the parents, *i.e.*, subsets of rules. For instance, consider the two rule sets $p_1 = \{r_{11}, r_{12}, r_{13}, r_{14}\}$ having four rules and $p_2 = \{r_{21}, r_{22}, r_{23}, r_{24}, r_{25}\}$ with five rules. If two cut-points are randomly set to 2 for $p_1$ and 3 for $p_2$, the offspring obtained are rule sets $o_1 = \{r_{11}, r_{12}, r_{24}, r_{25}\}$ and $o_2 = \{r_{21}, r_{22}, r_{23}, r_{13}, r_{14}\}$. Because each rule is syntactically and semantically correct before the crossover, this correctness is not altered for the offspring.

*Mutation.* After the crossover, the obtained offspring could be mutated with a given probability. Mutation allows the introduction of new genetic material while the population evolves. This is done by randomly altering existing rules or adding newly-generated ones. Mutation could occur at the rule set level or at the single rule level. Each time, a rule set is randomly selected for mutation, a mutation strategy is also randomly selected. Two mutation strategies are defined at the rule-set level: (1) *adding a randomly-created rule* to the rule set and (2) *deleting a randomly-selected rule.* To avoid empty rule sets, deletion could not be performed if the rule set has only one rule.

At the rule level, many strategies are possible. For a randomly-selected rule, one could *replace the target pattern* by a new one, randomly created. One could also *rebind one or more target pattern properties* by picking a random number of properties in the target pattern and randomly bind them to properties in the source pattern and constants. These modifications, when done as in Section 2.2, preserve the rule's validity, both syntactically and semantically. For the source pattern, it is also possible to introduce random modifications as for the target pattern. However, the target pattern has to be modified accordingly to avoid semantical and syntactical errors.

## 2.5 Evaluating Rule Sets

For the initial population and during the evolution, each generated rule set is evaluated to assess its ability to perform correct transformations. This evaluation is performed in two steps: (1) rule set execution on the examples and (2) comparison of produced vs. expected target models. Rule sets are translated into the JESS, and executed on the examples using the JESS rule engine. Metamodels are represented as sets of fact templates and models as fact sets. The rule translation is straightforward with the particularities that generic-target-construct declaration, join statements and bindings are merged into fact-assertion clauses. Listing 1.2 shows the JESS translation of the rule in Listing 1.1.

**Listing 1.2.** An example of a JESS Rule

```
(defrule RuleListing1
 (class (name ?C1))
 (attribute (name ?A1)(class ?A2))
 (association (maxCardFrom ?S1) (maxCardTo ?S2)(classFrom ?S3))
 (test (and (and (eq ?A2 ?C1)(eq ?S3 ?C1))
            (and (< ?S1 1)(> ?S2 1))))
=>
 (assert (table(name ?C1)))
 (assert (column(name ?A1)(table ?C1))))
```

Our fitness function measures the similarity between the target models produced by a rule set and the expected ones as given in the example model pairs. Consider $E$ the set of examples $e_i$ composed each of a pair of a source and a target model $(ms_i, mt_i)$. The fitness $F(E, p)$ of a rule set $p$ is defined as the average of the transformation correctness $f(mt_i, p(ms_i))$ of all examples $e_i$. The transformation correctness $f(mt_i, p(ms_i))$ measures to which extent the target model $p(ms_i)$, obtained by executing $p$ on the source model $ms_i$, is similar to the expected target model $mt_i$ of $e_i$.

Comparing two models, *i.e.*, two graphs with typed nodes, is a difficult problem (graph isomorphism). Considering that in the proposed GP-based rule derivation, the fitness function is evaluated for each rule set, on each example, and at each iteration, this cannot afford exhaustive graph comparisons. Instead, a quick an efficient graph kernel $f$ is used. $f$, which is a model similarity measure, calculates the weighted average of the transformation correctness per construct type $t \in T_{mt_i}$ in the expected model $mt_i$. This is done to give the same importance to all construct types regardless of their frequencies. Formally:

$$f(mt_i, p(ms_i)) = \sum_{t \in T_{mt_i}} \frac{f_t(mt_i, p(ms_i))}{|T_{mt_i}|} \tag{1}$$

$f_t$ is defined as the weighted sum of percentages of the constructs of type $t$ that are respectively fully ($fm_t$), partially ($pm_t$), or non($nm_t$) matched:

$$ft(mt_i, p(ms_i)) = \alpha fm_t + \beta pm_t + \gamma nm_t,\ \alpha + \beta + \gamma = 1 \tag{2}$$

For each construct of type $t$ in the expected model, we first determine if it is fully matched by a construct in the produced model, *i.e.*, it exists in the produced model a construct of the same type that have the same property values. For the constructs in the expected model that are not matched yet, we determine, in a second step, if they can be partially matched. A construct is partially matched if it exists in the produced model a construct of the same type that was not matched in the first step. Finally, the last step is to classify all the remaining constructs as not matched.

Coefficients $\alpha$, $\beta$, and $\gamma$ have each a different impact on the derivation process during the evolution. $\alpha$, which should be set to a high value (typically 0.6), is used to favor rules that correctly produce the expected constructs. As mentioned earlier, $\beta$, with an average value ($\approx 0.3$), allows to give more chances to rules producing the right types of the expected constructs and helps converging towards the optimal solution. Finally, $\gamma$ has to be set to a small value ($\approx 0.1$). The idea of giving a small weight to the not-matched constructs seems counterintuitive. However, our experience shows that this promotes diversity, particularly during the early generations, and this helps avoid local solution optima.

The calculation of the transformation correctness assesses whether the constructs of the expected model are present in the produced model. As a consequence, a good solution could include the correct rules that generate the right constructs, but it could also contain redundant rules or rules that generate unnecessary constructs. To handle this situation, we consider the size of the rule set when selecting the best solution. Consequently, even if an optimal solution is found in terms of correctness, the evolution process continues to search for equally-optimal solutions, but with fewer rules.

## 3 Evaluation

We evaluate our approach from two perspectives. First, a quantitative evaluation allows to answer the question: To which extent our approach generates rules that correctly transform the set of provided examples? In a second phase, a qualitative evaluation will help answering the question: If the examples are correctly transformed, are the produced rules those that are expected? In this context, we constructed a semi-real environment where the transformation solutions are known and where the examples models are simulated by creating prototypical source models and by deriving the corresponding target models using the known transformations. We were aware of the limitations of this setting, but it helps investigate more problems and it clearly defines the reference rule sets that the approach should derive. Additionally, it reasonably simulates situations where the examples have been manually created over a long period of time by experts.

The preliminary version of our approach was evaluated on the transformation of class diagrams to relational schemas [5]. This transformation, call it case $A$, illustrates well the problem of transforming structural models. Its complexity resides, among others, in the multiple possibilities of transforming the same construct according to the values of its properties. In the evaluation of the improved version presented in this paper, we also studied the transformation of UML2 sequence diagrams to state machines (Case $B1$ for basic sequence diagrams and Case $B2$ for advanced ones). Such a transformation is difficult because, in addition to considering the transformation of single model fragments and ensuring the structural coherence, it introduces two important modelling characteristics: time constraints and nesting. In this transformation, the coherence in terms of time constraints and weak sequencing should be guaranteed. On the other hand, nesting is tested because this transformation have to deal with combined fragments (alternatives, loops, and sequences) that can be nested at different levels, and thus, this transformation has to manage the recursive compositions in addition to handling the structural and time coherence. For case $A$, we used the transformation described in [3], whereas for cases $B1$ and $B2$, we rewrote, as rules, the graph-based transformations given in [7]. As GP-based algorithms are probabilistic in nature, five runs were performed in parallel for each case. For each run, we set the number of iterations to 3000, the population size to 200 and elitism to 20 programs. Crossover probability was set to 0.9 and mutation probability to 0.9. Unlike classical genetic algorithms, having a high mutation probability is not unusual for GP algorithms (e.g. [18]). The weighs $(\alpha, \beta, \gamma)$ of the fitness function were set to $(0.6, 0.3, 0.1)$, as explained in Section 2.5.

## 3.1 Quantitative Results

For each case, an optimal solution was found in at least one of the five runs. This is an indication that the search process has a good probability of convergence. The charts with sampled data that are shown in figures 2 and 3 illustrate the evolution of a successful run for cases $A$ and $B2$[2]. Three curves are displayed in each plot: the fitness function value ($F$) and the proportion of full matches ($FM$) (vertical axis on the left), and the rule set size ($PS$) (vertical axis on the right). The curves correspond to the fittest rule set at each generation (iteration) identified in the horizontal axis.

The solution evolutions for both cases follow the same pattern and differ only in the number of generations needed to converge toward a solution and to reach a minimal rule-set size. As expected, case $A$, with structural constraints only, is the one with the fastest convergence. At the initial generation, which is considered as a random transformation generation, half of the constructs are correctly transformed ($FM = 0.5$). These are simple one-to-one transformations (class-to-table or attribute-to-column) that have high chances of being generated randomly. The optimal solution in terms of $FM$ is found at the 59th generation

[2] The complete data can be downloaded at `http://geodes.iro.umontreal.ca/en/projects/MOTOE/ICMT13`

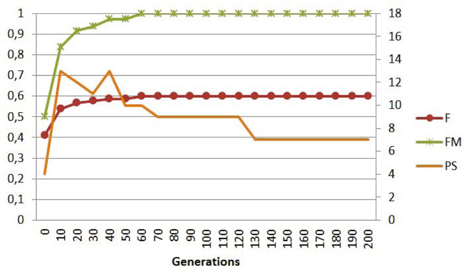


**Fig. 2.** Search evolution for case $A$

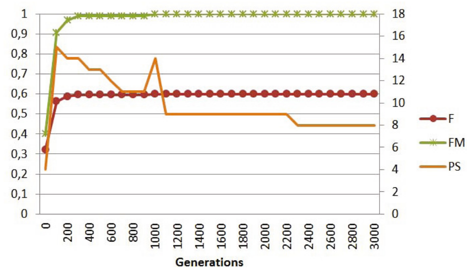


**Fig. 3.** Search evolution for case $B2$

with 10 rules. Once a solution with $FM = 1$ is found, the search process continues so that the current solution is replaced if another one with fewer rules is found. This happened three times for case $A$, with the last occurrence at the $129^{th}$ generation where the number of rules dropped to 7. No further improvement was observed during the rest of the evolution in terms of number of rules. Compared to the results obtained on this case with our previous work [5], a significant improvement was observed (100% *vs.* 75% for FM).

Case $B2$, for which structural, time and nesting constraints are involved, took many more generations (991) to converge to 100% of full match, with minimal rules achieved at generation 2280. The complexity of the transformation and the increase in the size of the search space also reduced the proportion of correct transformations obtained randomly in the initial population ($FM = 0.4$ for the initial generation compared to $FM = 0.5$ in case $A$). Case $B1$ has similar results as $B2$, but with a faster convergence curve. From the computational perspective, the parallel five runs took collectively between one hour for case $A$ and three hours for case $B2$ on a standard workstation (CPU @ 3.40GHz with 16 Go of RAM). Although this time could be reduced by optimizing the code, it is not considered excessive knowing that the process of learning new transformations is not intended to be executed frequently.

### 3.2 Qualitative Results

Obtaining 100% correct transformations of examples does not necessarily mean that we have derived the expected rules. In theory, for a limited sample of test cases, the same output values could be produced by different programs. Thus, to assess our results qualitatively, we need to compare the produced rules with those used to generate the examples (expected ones).

For cases $A$, we were searching for rules to transform classes, associations with various cardinalities and inheritance relationships. The expected rule set was found with a slight difference in one rule. Indeed, as all the classes in our examples contain at least one attribute, the rule that creates a table from a class has an unnecessary condition on the presence of an attribute. This kind of situations cannot be detected automatically because there is no counterexample. In the case of $B1$, the expected rules to create a state machine for every object in the sequence diagram, considering messages as events, were perfectly recovered.

Finally for $B2$, rules have to be found for every combined fragment (sequences, loops, and alts) and for managing the nesting at different levels. Here again, the best solution contains all the expected rules with an additional one. The extra rule is subsumed by a correct rule that creates a *start* state from the initial message of a combined fragment. Both rules have the same target pattern, whereas the extra rule has additional conditions. This situation (subsumption) could be easily detected by an automatic rule-set cleaning phase.

### 3.3 Discussion

During the development and evaluation of our approach, we faced several challenges to address or circumvent. This section discusses the most important ones.

**Rule Execution Control.** In the existing MTBE approaches, including ours, rules are defined to search for model fragments in the source model following a source pattern, and instantiate corresponding model fragments in the target model according to a target pattern. The target model fragments are usually not independent and have to be properly connected to form a coherent target model. Connecting target model fragments is difficult because, in most transformation languages, rules cannot check if a construct is present in the target model to connect to the produced fragment. In most MTBE approaches, the connection is achieved implicitly by using the same naming space both for source and target models. In our work, we circumvent partially the connection problem by recreating the target constructs. This technique was sufficient to handle the studied transformation cases, but it may be of limited use for other complex transformation problems. A good solution to handle the connection problem may be an explicit approach that uses global variables and meta-rules (execution control) as explained in [17]. In such an approach, the derivation process would learn the control separately or along with the transformation rules. We plan to explore this idea in a future work.

**Complex Value Derivation.** In our experimental setting, rule conditions and binding statements consider property values as data elements that cannot be combined to create new data elements. For example, for a construct $C_1$ in the source model with two numeric properties $p_1$ and $p_2$ and a string property $p_3$, a condition like $C_1.p_1 + C_1.p_2 \leq 2$ could not be created. Similarly, for a construct $C_2$ to create in the target model with a string property $p_4$, we cannot derive the binding statement $C_2.p4 = \text{“}Der - \text{”} + C_1.p3$. In our approach, such conditions and binding statements could be recovered by adding value-derivation operators such as arithmetic and string operators in the primitive set $I$ (see Section 2.2). However, this can be done only at the cost of increasing the search-space size, with an impact on convergence. We plan to consider these new operators in the future after a code optimization phase to handle the extra computational cost.

**Transformation Examples.** In the evaluation of our initial approach [5], we used examples collected from the literature, whereas in the evaluation of the improved version, we used prototypical examples. Using prototypical examples helped to find the correct solution faster, because a reduced number of examples was necessary to cover the modeling space. However, these could be difficult to define in real situations. The choice of using prototypical or existing examples depend on the context: availability of expertise *vs.* availability of examples.

**Model Comparison.** The search for a solution is guided by the transformation correctness (fitness function). As mentioned in Section 2.5, an exhaustive comparison between the produced and expected models is costly. A trade-off is necessary between the comparison precision and the computational constraints. From our experience, sophisticated comparisons such as the one described in [5] do not impact the search much, when contrasted against the simple comparison described in this paper. We plan to conduct a rigorous cost benefit study to compare different alternatives of model-comparison functions.

## 4 Related Work

Learning transformations from examples takes inspiration from other domains such as programming by example [19]. Existing work could be grouped into two categories model transformation by example and model transformation by demonstration. In the first categories, the majority of approaches takes as input a set of pairs of source and target models. Models in each pair are generally manually aligned through fine-grained mapping between constructs of the two models [22]. Rule derivation is performed using *Ad hoc* algorithms [6,22], machine learning such as inductive logic programming in [1], or association rule mining with formal concept analysis [4]. In our approach, we use a different category of derivation algorithm, *i.e.*, genetic programming. In this algorithm, candidate transformation programs are evolved with the objective of better matching the provided transformation examples. The derivation does not require the user alignment/mapping of models that could be difficult to formalize in many cases. Indeed, once a candidate program is derived, it is executed on the example source models and its output is compared to the example target models. One positive side effect of our approach is that the obtained rules are executed and tested during the derivation process, which helps assessing each rule individually and the rule set globally. In some of the above-mentioned approached, the rules are not executable or are mapped in a subsequent step to an executable language. For example, the work in [4] is extended by mapping the derived association rules into executable ones in JESS [20]. In the same category of contributions, the work by Kessentini et al. [11] brings a different perspective to the MTBE problem. Rather than deriving a reusable transformation program, it defines a technique that automatically transforms a source model by analogy with existing transformation examples. Although this could be useful for some situations, the inability to derive transformation rules/knowledge could be seen as a limitation.

The second category of contributions in transformation rule learning is the model transformation by demonstration (MTBD). The goal here is to derive transformation patterns starting from step by step recorded actions on past transformations. In [23], Sun et al. propose an approach to generalize model editing actions (e.g., add, delete, update) that a user performs to refactor a model. The user editing actions are recorded and serve as patterns that can be later applied on a similar model by performing a pattern-matching process. This approach is intended to perform endogenous transformations (refactoring) and its generalization to exogenous transformation is not trivial. in [14], Langer et al. proposes an MTBD approach, very similar to the previous one, with the improvement of handling exogenous transformations. MTBD solves many problems of MTBE, as complex transformation could be abstracted. However, transformation patterns are derived individually and there is no guarantee that patterns could be applied together to derive consistent target models. In our case, the fact that rule sets are evaluated by executing them on the example source models, helps assessing the consistency of the produced models.

In addition to the differences highlighted in the previous paragraphs, our approach allows generating many-to-many rules that search for non trivial patterns in the source models and instantiate non trivial patterns in the target models. In contrast with the state-of-the-art approaches, we do not try to derive patterns by explicitly generalizing situations found among the examples. We instead use an evolutionary approach that evolves transformation programs, guided by their ability to correctly transform the example at hand. Finally, it is difficult to compare quantitatively and qualitatively with the other approaches. The validations of most of these are not or only partially reported.

## 5 Conclusion

Prior work has demonstrated that model transformation rules could be derived from examples. However, these contributions require fine-grained examples of model mapping or need a manual refinement phase to produce operational rules. In this paper, we propose a novel approach based on genetic programming to learn operational rules from pairs of unrelated models, given as examples. This approach was evaluated on structural and time-constrained model transformations. We found that in virtually all the cases, the produced rule sets are operational and correct. Our approach is a new stone in the resolution of the MTBE problem, and our evaluation provides a compelling evidence that MTBE could be an efficient solution to many transformation problems. However, some limitations are worth noting. Although the approach worked well for the addressed problem, the evaluation showed that convergence is difficult to reach for complex transformations. Future work should therefore include the explicit reasoning on rule execution control to simplify the transformation rules. It should also better consider transformations with complex conditions and bindings. In particular, we consider dealing with source and target models that do not share the same naming space using natural-language processing techniques.

## References

1. Balogh, Z., Varrò, D.: Model transformation by example using inductive logic programming. Soft. and Syst. Modeling 8 (2009)
2. Banzhaf, W.: Genetic Programming: An Introduction on the Automatic Evolution of Computer Programs and Its Applications. Morgan Kaufmann Publishers (1998)
3. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3) (2006)
4. Dolques, X., Huchard, M., Nebut, C., Reitz, P.: Learning transformation rules from transformation examples: An approach based on relational concept analysis. In: Int. Enterprise Distributed Object Computing Workshops (2010)
5. Faunes, M., Sahraoui, H., Boukadoum, M.: Generating model transformation rules from examples using an evolutionary algorithm. In: Aut. Soft. Engineering (ASE) (2012)
6. García-Magariño, I., Gómez-Sanz, J.J., Fuentes-Fernández, R.: Model transformation by-example: An algorithm for generating many-to-many transformation rules in several model transformation languages. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 52–66. Springer, Heidelberg (2009)
7. Grønmo, R., Møller-Pedersen, B.: From UML 2 sequence diagrams to state machines by graph transformation. Journal of Object Technology 10 (2011)
8. Hill, E.F.: Jess in Action: Java Rule-Based Systems (2003)
9. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
10. Kessentini, M., Sahraoui, H.A., Boukadoum, M.: Model transformation as an optimization problem. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Heidelberg (2008)
11. Kessentini, M., Sahraoui, H.A., Boukadoum, M., Omar, O.B.: Search-based model transformation by example. Soft. and Syst. Modeling 11(2) (2012)
12. Kessentini, M., Wimmer, M., Sahraoui, H., Boukadoum, M.: Generating transformation rules from examples for behavioral models. In: Proc. of the 2nd Int. WS on Behaviour Modelling: Foundation and Applications (2010)
13. Koza, J., Poli, R.: Genetic programming. In: Search Methodologies (2005)
14. Langer, P., Wimmer, M., Kappel, G.: Model-to-model transformations by demonstration. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 153–167. Springer, Heidelberg (2010)
15. Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M.: An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. In: Empirical Software Engineering
16. Moore, G.: Crossing the Chasm: Marketing and Selling Disruptive Products to Mainstream Customers. HarperCollins (2002)
17. Pachet, F., Perrot, J.: Rule firing with metarules. In: SEKE (1994)
18. Ratcliff, S., White, D.R., Clark, J.A.: Searching for invariants using genetic programming and mutation testing. In: GECCO (2011)
19. Repenning, A., Perrone, C.: Programming by example: programming by analogous examples. Commun. ACM 43(3) (2000)
20. Saada, H., Dolques, X., Huchard, M., Nebut, C., Sahraoui, H.: Generation of operational transformation rules from examples of model transformations. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 546–561. Springer, Heidelberg (2012)

21. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2) (2006)
22. Strommer, M., Wimmer, M.: A framework for model transformation by-example: Concepts and tool support. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 372–391. Springer, Heidelberg (2008)
23. Sun, Y., White, J., Gray, J.: Model transformation by demonstration. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 712–726. Springer, Heidelberg (2009)
24. Varró, D.: Model transformation by example. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)

# CHAPTER 6

# AUTOMATICALLY SEARCHING FOR METAMODEL WELL-FORMEDNESS RULES IN EXAMPLES AND COUNTER-EXAMPLES

In this chapter we present the third contribution [21] entitled "Automatically searching for metamodel well-formedness rules in examples and counter-examples" published at "Model-Driven Engineering Languages and Systems", 2013, pages 187-202. The contribution aims to improve automation in MDE. Specifically, the approach proposes a method to improve modeling language precision. In particular, the approach proposes a method to derive well-formedness rules for metamodels, based on a optimization search (genetic programming) that is guided by examples of valid and invalid models. We validate our approach over two metamodels for which our approach can derive most of the relevant well-formedness rules. In the remainder of this chapter we present the paper.

# Automatically searching for metamodel well-formedness rules in examples and counter-examples

Martin Faunes[1], Juan Cadavid[2], Benoit Baudry[2], Houari Sahraoui[1], and Benoit Combemale[2]

[1] Université de Montréal, Montreal, Canada

[2] IRISA/INRIA, Rennes, France

**Abstract.** Current metamodeling formalisms support the definition of a metamodel with two views: classes and relations, that form the core of the metamodel, and well-formedness rules, that constraints the set of valid models. While a safe application of automatic operations on models requires a precise definition of the domain using the two views, most metamodels currently present in repositories have only the first one part. In this paper, we propose to start from valid and invalid model examples in order to automatically retrieve well-formedness rules in OCL using Genetic Programming. The approach is evaluated on metamodels for state machines and features diagrams. The experiments aim at demonstrating the feasibility of the approach and at illustrating some important design decisions that must be considered when using this technique.

## 1 Introduction

Metamodeling is a key activity for capitalizing domain knowledge. A metamodel formally defines the essential concepts of an engineering domain, providing the basis for the automation of many operations on models in this domain (*e.g.,* analysis, simulation, refactoring, transformation, visualization). However, domain engineers can benefit from the full power of automatic model operations only if the metamodel is precise enough to effectively specify and implement these operations, as well as to ensure a safe application. Current metamodeling techniques, such as EMF[3], GME [13] or MetaEdit+[4], impose to define a metamodel as two parts: a *domain structure*, which captures the concepts and relationships that can be used to build models in a specific domain, and *well-formedness rules*, that impose further constraints that must be satisfied by all

[3] Eclipse Modeling Framework, cf. `http://www.eclipse.org/modeling/emf/`
[4] cf. `http://www.metacase.com`

models in the domain. The domain structure is usually modeled as a class diagram, while well-formedness rules are expressed as logical formula.

When looking at the most popular metamodel repositories (*e.g. [1]*, we find hundreds of metamodels which include only the domain structure, with no well-formedness rules. The major issue with this is that it is possible to build models that conform to the metamodel (*i.e.*, satisfy the structural constraints imposed by concepts and relationships of the domain structure), but are invalid with respect to the domain. For example, considering the class diagram metamodel without well-formedness rules, it is possible to build a class diagram in which there is a cyclic dependency in the inheritance tree (this model would be valid with respect to the domain structure but invalid with respect to the domain of object-oriented classes). From an engineering and metamodel exploitation perspective, the absence of well-formedness rules is a problem because it can introduce errors in operations that are defined on the basis of the domain structure. For example, operations that rely on automatic model generation might generate wrong models or compatibility analysis (*e.g.* to build model transformation chains) can be wrong if the input model is considered as conforming to the domain structure while it does not fully conform to the domain.

The intuition of this work is that domain experts know the well-formedness rules, but do not explicitly model them and some operations may consider them as assumptions (i.e., hidden contract). We believe that experts know them in the sense that, if we show them a set of models that conform to the domain structure, they are able to discriminate between those that are valid with respect to the domain and those that are not. However, we can only speculate about why they do not formalize them. Given the importance of well-formedness rules, we would like to have an explicit model of these rules to get a metamodel as precise as possible and get the greatest value out of automatic operations on models.

In this work, we leverage domain expertise to automatically generate well-formedness rules in the form of OCL (*Object Constraint Language*) invariants over a domain structure modeled as a class diagram with MOF. We gather domain expertise in the initial domain structure and a set of models that conform to the domain structure, in which some models are valid with respect to the domain and some models are invalid. Starting from this input, our technique relies on Genetic Programming [12] to automatically generate well-formedness rules that are able to discriminate between the valid and invalid models.

We validate our approach on two metamodels: a state machine metamodel and a feature diagrams metamodel. For the first metamodel our approach finds 10 out of 12 well-formedness rules, with $precision = recall = 0.83$. For the second metamodel we retrieve seven out of 11 well-formedness rules with a $precision = 0.78$ and $recall = 0.64$.

The contributions of this paper are the following:

– formalizing the synthesis of well-formedness rules as a search problem;
– a set of operators to automatically synthesize and mutate OCL expressions;
– a series of experiments that demonstrate the effectiveness of the approach and provide a set of lessons learned for automatic model search and mutation.

The paper is organized as follows. Section 2 provides the background and, defines and illustrates the problem addressed. Section 3 details the proposed approach using Genetic Programming to derive well-formedness rules, and Section 4 reports our experiments to evaluate the approach. Section 5 surveys related work. Finally, we conclude and outline our perspectives in Section 6.

## 2 Problem definition

This section precisely defines what we mean by metamodeling and illustrates how both the domain structure and well-formedness rules are necessary to completely specify a metamodel. Then we illustrate how the absence of well-formedness rules can lead to situations where models conform to the domain structure but are invalid with respect to the domain.

### 2.1 Definitions

**Definition 1.** ***Metamodel****. A metamodel is defined as the composition of:*

– ***Domain structure****. This part of the metamodel specifies the core concepts and attributes that define the domain, as well as the relationships that specify how the concepts can be bound together in a model.*
– ***Well-formedness rules****. Additional properties that restrict the way concepts can be assembled to form a valid model.*

The method we introduce in this work can be applied to any metamodel that is specified according to this definition. Nevertheless, for this work we had to choose concrete formalisms to implement both parts. Thus, here, we experiment with domain structures formalized with MOF and well-formedness rules formalized with the Object Constraint Language (OCL).

### 2.2 Illustration of precise metamodeling

Here we illustrate why both parts of a metamodel are necessary to have a specification as precise as possible and avoid models that conform to the metamodel but are invalid with respect to the domain. The model in Fig. 1 specifies a simplified domain structure for state machines. A `StateMachine` is composed of several `Vertex`s and several `Transitions`. `Transition`s have a source and a target `Vertex`, while `Vertex`s can have several incoming and outgoing `Transition`s. The model distinguishes between several different types of `Vertex`s.

The domain structure in Fig. 1 accurately captures all the concepts that are necessary to build state machines, as well as all the valid relationships that can exist between these concepts. However, valid models can also exist, of this structure, that are not valid state machines. For example, the metamodel does not prevent the construction of a state machine in which a join pseudostate has only one incoming transition (when it should have at least 2). Thus, the sole

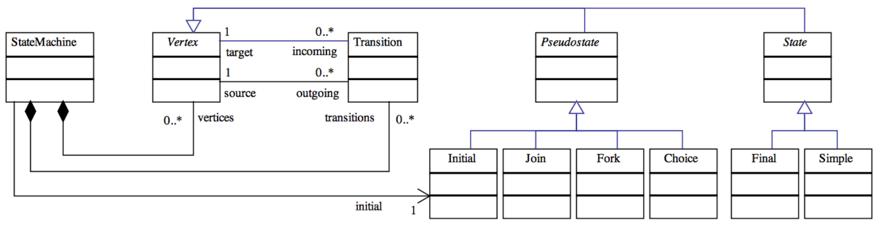


Fig. 1: State machine metamodel

domain structure of Fig. 1 is not sufficient to precisely model the specific domain of state machines.

The domain structure needs to be enhanced with additional properties to capture the domain more precisely. The following well-formedness rules, expressed in OCL, show some mandatory properties.

1. $WFR1$: Join pseudostates have one outgoing transition

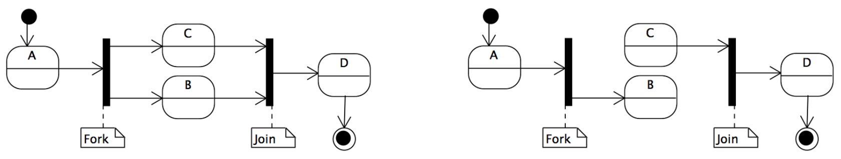
   ```
   (context Join inv : self.outgoing->size() = 1))
   ```

2. $WFR2$: Fork pseudostates have at least two outgoing transitions

   ```
   (context Fork inv : self.outgoing->size() > 1)
   ```

## 2.3 Problem definition

The initial observation of this work is that most metamodelers build the domain structure, but do not specify the well-formedness rules. The absence of these rules allows the creation of models that conform to the metamodel (only domain structure) but are not valid with respect to the domain. For example, if we ignore the well-formedness rules illustrated previously, it is possible to build the two models of Fig. 2a and Fig. 2b. Both models conform to the structure of Fig. 1, but the model of Fig. 2b is an invalid state machine.



(a) Valid (b) Invalid

Fig. 2: Example of state machines

The intuition of this work is that, given a domain structure without well-formedness rules, it is possible (i) to generate models (*e.g.*, using test model generation techniques [2]) and (ii) to ask domain experts to sort these models between valid and invalid. Then, our objective is to automatically retrieve a set of well-formedness rules. The retrieved well-formedness rules are not meant to be exactly those sought (that are unknown), but shall be a good approximation. In particular, they should be able to properly discriminate models beyond those provided in the learning process, *i.e.*, they should generalize the examples.

## 3 Approach description

### 3.1 Approach overview

The problem, as described in Section 2, is complex to solve. The only inputs to our derivation mechanism are the sets of examples of valid (positive) and invalid (negative) models. Hence, our goal is to retrieve the minimal set of well-formedness rules that better discriminate between the two sets of models.

From a certain perspective, well-formedness rule sets could be viewed as declarative programs that take as input a model and produce as output a decision about the validity of this model with respect to the domain. This observation motivates the use Genetic Programming (GP) as a technique to derive such rule sets. Indeed, GP is a popular evolutionary algorithm which aims at automatically deriving a *program* that approximates a *behaviour* from examples of inputs and outputs. It is used in a scenario where manually writing the program is difficult. In our work, the examples of inputs are the models and the outputs are their validity. As we will show later in this section, to guide the derivation process, well-formedness rules should be evaluated on the example models. To this end, the rules to search for are implemented as OCL invariants[5][6].

The boundaries of our derivation process are summarized in Fig. 3. In addition to example models, the derivation process takes as input a metamodel for which the invariants are sought. It produces as output fully operational OCL invariants that represent an approximation to the sought invariants.

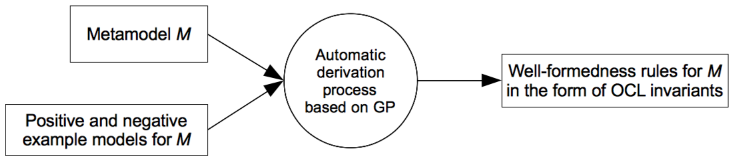

Metamodel *M*
Positive and negative example models for *M*
Automatic derivation process based on GP
Well-formedness rules for *M* in the form of OCL invariants

Fig. 3: Approach overview

In the next two sub-sections, first, a brief introduction to the GP technique is given and then its use to solve specifically the problem of well-formedness rule derivation is described.

[5] http://projects.eclipse.org/projects/modeling.mdt.ocl.

[6] In the remainder of this section, we use the term "invariant" (resp. "invariant set") to designate a well-formedness rule (resp. rule set)

## 3.2 Genetic Programming

The most effective way to understand GP is to look to the typical GP process (cycle), sketched in Fig. 4. Step 1 of a GP cycle consists of creating an initial population of randomly-created programs. Then, in step 2, the fitness of each program in the current population is calculated. This is typically done by executing the programs over the example inputs and comparing the execution results with the expected outputs (those given as example). If the current population satisfies termination criteria in step 3, *e.g.*, a predefined number of iterations or a target fitness value, the fittest program met during the evolution is returned (step 7); otherwise, in step 4, a new population is created (it is also called *evolving* the current population). This is done by selecting the fittest programs of the current population and reproducing them. Although, the selection process favors the programs with the highest fitness values, it still gives a chance to any program to avoid local optima. Reproduction involves three families of genetic operations: (i) *elitism* to directly add top-ranked programs to the new population, (ii) *crossover* to create new programs by combining *genetic material* of the old ones, and (iii) *mutation* to alter an existing program by randomly adding new *genetic material*. Once a new population is created, it replaces the current one (step 5) and the next iteration of the GP cycle takes place, *i.e.*, steps 2 to 5. Thus, programs progressively change to better approximate the behaviour as specified by the inputs/outputs.

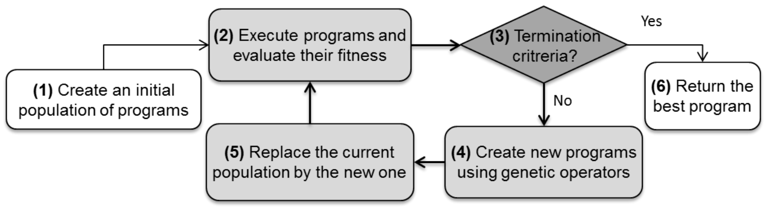

(1) Create an initial population of programs
(2) Execute programs and evaluate their fitness
(3) Termination critreria?
Yes
No
(6) Return the best program
(4) Create new programs using genetic operators
(5) Replace the current population by the new one

Fig. 4: A typical GP cycle

## 3.3 Using GP to Derive Well-Formedness Rules

To adapt GP to our problem, we have to produce a set of positive and negative models (base of examples). Then, we need to define a way to encode a set of invariants and to create the initial population of them. Another action consists in selecting a mechanism to execute sets of invariant on the provided models to calculate their fitness. Finally, proper genetic operators should be defined to evolve the population of candidate sets. In the rest of this section, these adaptations are described in details.

*Input/output encoding:* The base of examples $E$ is a set of pairs $e = (m, v)$ where $m$ is a model (conforming to the considered metamodel $M$) and $v$, a *boolean*, is the model validity stating if $m$ satisfies the invariants or not. We refer to the example model as $e_m$ and to the example model validity as $e_v$. Each model $m$ conforms to the ECORE [16] metamodel $M$.

*Invariant set encoding:* In GP, a population of programs is initially created and evolved to search for the one which better approximates the behavior specified by the examples of inputs and outputs. In our adaptation, a program is a set $p$ that contains OCL invariants $i_j$, $p = \{i_1, i_2, ..., i_n\}$. A model $m$, to be valid given an invariant set $p$, has to satisfy each invariant $i_j \in p$. To encode an OCL invariant $i_j$, we use the format provided by the Eclipse OCL framework. An OCL invariant is seen as a tuple $(c, t)$ where $c$ is the context, *i.e.*, a main metamodel class, and $t$ is a tree that combines logical operators, comparison operators, functions, metamodel elements, and constants according to OCL syntax. Metamodel elements can be class attributes or class relationships (called references). In such a tree, the leave nodes are metamodel elements and constants, and the leave-node parents are comparison operators and functions. Any node on top of these two levels is a logical operator. In our implementation, we use the logical operators $\{and, or, not, implies\}$, comparison operators $\{>, <, =, \geq, \leq, \neq\}$, and other operations like $\{isKindOf, forAll, includesAll, size, allInstances, etc.\}$. These operations are generally enough to encode a wide range of OCL invariants.

*Random invariant set creation:* The first phase of the well-formedness rule derivation process is the random generation of the initial population, consisting of $n$ invariant sets. In theory, there is an infinity of possible invariants that can be generated for a given metamodel. However, Cadavid et al. [3] showed empirically, *i.e.*, by analyzing dozens of metamodels from the standard community, academia, and industry, that there is a limited number of recurrent invariant patterns (20), whose instances are used individually or combined to create complex invariants. A pattern example is *CollectionSizeEqualsOne*, which states that the size of a collection *col*, contained in a class $A$, should be equal to 1:

```
context A inv : col->size() = 1
```

Such a pattern could be instantiated for any collection that can be found in a class, regardless of its type. Two possible instantiations for the state-machine metamodel in Fig. 1, could be the following:

```
context Fork inv : self.incoming-> size() = 1
context Fork inv : self.outgoing-> size() = 1
```

In our random generation process, we first automatically produce all the possible instances of the above-mentioned 20 basic patterns for the considered metamodel. This results in a large number of rules, lots of them are wrong, some of them are too simple or with wrong parameter values and thus it is still necessary to explore, combine and mutate this initial space of rules in order to produce the right set. To this end, for each invariant set to create, we randomly pick some of of the generated instances to produce simple invariants or complex ones by

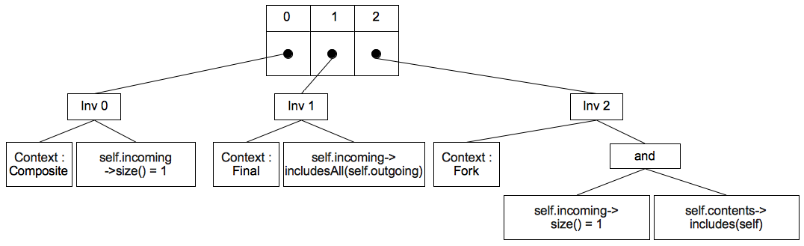

0
1
2
Inv 0
Inv 1
Inv 2
Context : Composite
self.incoming ->size() = 1
Context : Final
self.incoming-> includesAll(self.outgoing)
Context : Fork
and
self.incoming-> size() = 1
self.contents-> includes(self)

Fig. 5: An example of a randomly-created invariant set

combining the chosen instances with logical operators. Simple invariants can be combined if they share the same context. Fig. 5 shows an example of a set with three invariants. The two first invariants are simple and contain respectively an instance of the pattern *CollectionSizeEqualsOne* and an instance of the pattern *CollectionIsSubset*, *i.e.*, a collection that shoud be included in another one. The third invariant is the conjuction of an instance of *CollectionSizeEqualsOne* with an instance of *CollectionIncludesSelf*, *i.e.*, if a class contains a collection typed with itself, an instance of this class also makes part of this contained collection.

The number of instances to select as well as the number of combinations to perform to produce complex invariants (tree depths) are decided randomly during the creation of each set. The pattern instances are syntactically (w.r.t the OCL syntax) and semantically (w.r.t the metamodel structure) correct as they are their combinations. However, this does not mean that they are good invariants. This is decided by the fitness function.

*Fitness calculation:* In our implementation, OCL invariants are evaluated on the example models using the Eclipse OCL engine. The fitness function $f$ assesses how well an invariant set $p$ discriminates the models contained in the base of examples $E$ with respect to the expert-based classification. $f$ is a weighed function of two sub-functions $f_1$ and $f_2$. The first component, $f_1$, measures the rate of example models in $E$ that are well classified by $p$. A model $e_m$ is well classified if $v(e_m, p)$, the evaluation of $p$ on $e_m$, is equal to $e_v$. $f_1$ is defined as:

$$f_1(p, E) = \frac{\sum_{e \in E} I(v(e_m, p) = e_v)}{|E|} \rightarrow [0, 1] \quad (1)$$

Function $I(a)$ returns 1 if $a = true$ and 0 otherwise. The evaluation of a set of invariants $p$ on a model $m$ , $v(m, p)$, is defined formally as:

$$v\ (m, p) = u(m, i_1) \wedge u(m, i_2) \wedge ... \wedge u(m, i_z) \rightarrow Boolean; \forall i_k \in p \quad (2)$$

Here, $u(m, i)$ is a boolean function that returns $true$ if $m$ satisfies the invariant $i$ and $false$ otherwise.

Component $f_1$ allows to evaluate the set of invariants as a whole. However, it could penalize candidate sets that include good invariants but a few ones. To reward good invariants individually, we defined a second component, $f_2$, of the fitness function. $f_2$ is calculated by counting the invariants $i \in p$ that are able to find at least $\alpha$ true positives $T_p$ and at least $\beta$ true negatives $T_p$. We then divide by the number of invariants $|p|$ to normalize the result between 0 and 1:

$$f_2\left(p, E\right) = \frac{\sum_{e \in E} I\left(T_p(i, E) \geq \alpha \ \wedge T_n(i, E) \geq \beta\right)}{|p|} \rightarrow [0, 1] \tag{3}$$

Here, a true positive (resp. negative) is a model $e \in E$ classified as valid (resp. invalid) and that satisfies (resp. not satisfies) the invariant $i \in p$:

$$T_p\left(p, E\right) = \sum_{e \in E; e.v} I\left(u\left(e, i\right)\right); T_n\left(p, E\right) = \sum_{e \in E; \neg e.v} I\left(\neg u\left(e, i\right)\right) \tag{4}$$

Now that we can generate an initial population and evaluate each of the invariant sets, the next step consists in selecting invariant sets to use them to produce a new population by applying crossover and mutation operators.

*Selection method:* To determine which sets of invariants will be reproduced to create the new population, the *Roulette-wheel* selection method is used in this work. This technique assigns to each invariant set in the current population a probability of being selected for reproduction that is proportional to its fitness. This selection strategy favours the fittest invariant sets while still giving a chance to the others.

*Genetic Operators* : The crossover operator consists of producing new invariant sets by combining the existing genetic material. After selecting two parent sets for reproduction, two new invariant sets are created by exchanging invariants of the parents. For instance, consider the two invariant sets $p_1 = \{i_{11}, i_{12}, i_{13}, i_{14}\}$ having four invariants and $p_2 = \{i_{21}, i_{22}, i_{23}, i_{24}, i_{25}\}$ with five invariants. If a cut-point is randomly set to 2 for $p_1$ and another to 3 for $p_2$, the offspring obtained are invariant sets $o_1 = \{i_{11}, i_{12}, i_{24}, i_{25}\}$ and i$o_2 = \{i_{21}, i_{22}, i_{23}, i_{13}, i_{14}\}$. Because each parent invariant is syntactically and semantically correct before the crossover, this correctness is not altered for the offspring. Crossover is applied with high probability.

Mutation allows to randomly inject new genetic materiel in the population. It is applied with a low priority to offsprings after a crossover or to the selected parents when the crossover is not applied. In our adaptation of GP, we implemented 10 mutation operators that modify an invariant set at many levels. Every operator preserves the sibling correctness, syntactically and semantically. The first three operators are defined at the set level. One allows to add a new invariant, produced randomly according to the procedure used in the initial population generation. The second operator simply picks one of the existing invariants in the set and removes it. If we consider the set of Fig. 5 , we could have, for instance, the following mutations, corresponding respectively to the two operators:

```
Add: context Orthogonal inv : self.outgoing->includesAll(self.incoming)
Remove: context Fork inv : self.incoming-> size() = 1
```

The third operator at the set level selects two invariants, simple or complex, having the same context, and combines them using the "implies" operator. The remaining operators are defined at the invariant level. For one invariant of the considered set, some mutations consist in replacing respectively a comparison or a logical operator by a new one. For example, "=" in "Inv 0" of Fig. 5 could be replaced by ">". Similarly, "and" in "Inv 2" could become "implies". Incrementing/decrementing a numerical constant and replacing an attribute or a reference by a new one that is of the same type and that belongs to the same context, also are possible mutations, *e.g.*, replacing 1 by 0 or "incoming" by "outgoing" in "Inv 0". Another used mutation is the replacement of an operand (sub-tree) of a logical operator or a comparator by a randomly generated one. For example, the operand "self.contents->includes(self)" in "Inv 2" could be replaced by "self.outgoing->size() = 0". The final mutation is the negation of a node that returns a boolean value (a logical operator, a comparison operator or a boolean function). For instance, "Inv 1" could be mutated to "not self.incoming ->includesAll(self.outgoing)".

All the decisions made during the mutation, including the selection of the mutation operator, the invariant to change, and the replacement elements, are determined randomly.

## 4 Evaluation

### 4.1 Research Questions

The evaluation of our approach addresses the two following research questions:

1. To which extent our approach is able to derive well-formedness rules that properly discriminate between valid and invalid models?
2. Are the produced well-formedness rules those that are expected?

The first questions aims at assessing the validity of the approach from the quantitative perspective while the second considers the qualitative perspective.

### 4.2 Experimental Setting

**Method**. To answer both research questions, we conduct an experiment in which we evaluate our approach over two different metamodels. The evaluation is performed in a semi-real environment in which we know a priori the well-formedness rules sought (OCL invariants provided with the metamodels). The example models are randomly created using Alloy [9]. The creation with Alloy takes into account the known invariants. The number of positive models that are created (those that satisfy all the invariants) is equal to five times the number of known invariants. An identical number of negative models is also created. To create negative models, we randomly negate one or more invariants to force Alloy to violate them. The positive and negative model examples are then given as input to the derivation process, but not the known invariants.

To answer the first question, we first calculate the classification correctness of the best found invariant set, *i.e.*, proportion of models in the example base that are correctly classified ($f_1$ in the fitness function). Then, considering the stochastic nature of our approach, *i.e.*, different executions may lead to different results, we take a sample of executions and compare it with another sample obtained by a random technique. To have a fair comparison, we defined the random technique as the selection of the best from $n \times m$ randomly–generated sets, where $n$ and $m$ are respectively the size of a population and the number of iterations in our approach. In other words, both our approach and the random technique explore the same number of invariant sets. The comparison of the two samples is done using an independent-sample t-test (or Mann-Whitney test if $f_1$ values are not normally distributed in the two execution samples). The tests are performed with a significance at the level of $\alpha = 0.05$, *i.e.*, a probability of less than 5% that the difference between the two samples is obtained by chance.

To answer the second research question, we analyzed the invariants of the best derived solution and compare them with the known invariants. The comparison produces four sets: invariants found that match the expected ones (FOU), invariant found that are subsumed (less general) by the expected ones (SUB), invariants that are not expected (INC), and expected invariants not found excluding the subsumptions (MIS). Ideally, all the found invariants should be in FOU and MIS should be empty. Solutions with all the invariants in FOU but a few in SUB are also acceptable. We defined two versions of precision and recall depending on the acceptance of subsumed invariants (relaxed) or not (strict), as follows:

$precision_{strict} = \frac{|FOU|}{|FOU|+|SUB|+|INC|}$ and $recall_{strict} = \frac{|FOU|}{|FOU|+|SUB|+|MIS|}$

$precision_{rel} = \frac{|FOU|+|SUB|}{|FOU|+|SUB|+|INC|}$ and $recall_{rel} = \frac{|FOU|+|SUB|}{|FOU|+|SUB|+|MIS|}$

**Data**.The first metamodel used is the one of state machines (see Fig. 1). We selected 12 OCL invariants related to the incoming and outgoing transitions depending on the state types. As mentioned earlier we created 60 positive and 60 negative models ($5 \times 12$ for each set).

The second metamodel that we consider represents the feature diagrams [11] (see Fig. 6). For this metamodel, we selected 11 OCL invariants covering the interdependencies between the feature types and the relation types. We created accordingly 55 positive and 55 negative example models.

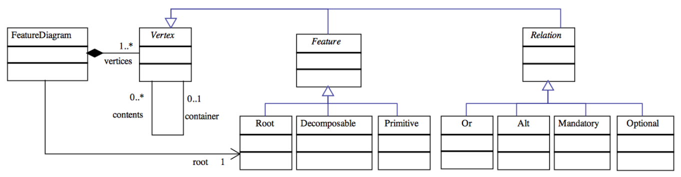


Fig. 6: Feature diagram metamodel

**Algorithmic parameters**. GP, being a meta-heuristic algorithm, it depends on many parameters. The population size was fixed to 100 invariant sets and the evolution was performed with a maximum of 1000 iterations. To ensure that the best invariant sets will be kept during the evolution, we used an elitism strategy that consists in automatically adding the 10 fittest sets of each generation to the next one. For the evolution operator, the crossover probability was set to 0.9. We used the same probability for mutation. Unlike classical genetic algorithms, having a high mutation probability is not unusual for GP algorithms (see, for instance, [14]). For the fitness function we give equal weights to $f_1$ and $f_2$ (0.5), and the parameter $\alpha$ of $f_2$ was set to 1. Finally, the probability of creating complex invariants vs. simple ones during the random creation is set to 0.1, *i.e.*, each time an invariant has to be generated, it has nine chances to be simple and one to be complex. This probability is recursively applied to the operands of the logical operators when a complex invariant is created.

### 4.3 Results

*Question 1.* Given the stochastic nature of the GP, we performed a sample of executions and took the best found set. For the state machine metamodel the optimal best set was found before reaching the maximum number of iterations (after 537 iterations). This set perfectly discrimnates the positive models from the negative ones ($f_1 = 1$). For the feature digram metamodel, the best set missclassified 10 from the 110 models ($f_1 = 0.91$). The second step was to assess if the GP-based derivation performs better, in terms of discrimination power, than random generation. We performed a Kolmogoriv-Smirnov test that revealed that the $f_1$ values are normally distributed in both GP-based and random execution samples. This allows us to perform an independent-samples t-test with the null hypothesis that there is no difference in $f_1$ between the two derivation techniques. As illustrated in Table 1, the GP-based derivation performs clearly better than the random technique ($\sim 0.9$ compared to $\sim 0.25$) and this difference in $f_1$ is statistically significant with $p < 0.001$ for both metamodels.

Table 1: Comparison with random generation (Question 1).

| Metamodel | Average $f_1$ for GP | Average $f_1$ for Random | Sig. |
|---|---|---|---|
| State machines | 0.96 | 0.22 | $< 0.001$ |
| Feature diagrams | 0.88 | 0.25 | $< 0.001$ |

*Question 2.* We manually analyzed the obtained invariants for each metamodel and compared them to the expected ones[7]. Table 2 summarizes the analysis results. For state machines, 12 invariants were found. 10 of them exactly matches

[7] Full results at `http://geodes.iro.umontreal.ca/en/projects/MOTOE/MODELS13`

Table 2: Precision and recall for invariant determination (Question 2).

| Metamodel | $precision_{strict}$ | $recall_{strict}$ | $precision_{rel}$ | $recall_{rel}$ |
|---|---|---|---|---|
| State machines | 0.83 | 0.83 | 0.83 | 0.83 |
| Feature diagrams | 0.78 | 0.64 | 0.89 | 0.73 |

expected invariants, 2 are incorrect and 2 are missing. This led to a precision and a recall (strict an relaxed) of 0.83. The missing and incorrect invariants are:

```
Missing invariants
  context Initial inv : self.incoming->size() = 0
  context Final inv : self.outgoing->size() = 0
Incorrect invariants
  context Initial inv : self.outgoing->includesAll(self.incoming))
  context Final inv : self.incoming->includesAll(self.outgoing))
```

We expected invariants enforcing that the set of incoming (respectively outgoing) transitions is empty for initial (respectively final) states. Our algorithm, based on the examples, finds invariants that evaluate to true, as empty sets are always included in other sets, but do not represent the correct semantic.

For the feature diagrams, the results were slightly worse. Indeed, 9 invariants were derived. 7 of them are good invariants whereas one is subsumed and one is incorrect. 3 expected invariants were not recovered. Consequently, the strict precision is 0.78 and the strict recall 0.64, whereas, the relaxed ones are increased respectively to 0.89 and 0.73. The concerned invariants are:

```
Missing invariants
 context Or inv : contents->forAll(v:Vertex | v.oclIsKindOf(Feature))
 context Optional inv : contents->forAll(v:Vertex | v.oclIsKindOf(Feature))
 context PrimitiveFeature inv : self.contents->size() = 0
Incorrect invariant
  context PrimitiveFeature inv : self.container->includesAll(self.contents))
Subsumed invariant
  Expected: context DecomposableFeature inv : self.contents->size() > 1
  Found: context DecomposableFeature inv : self.contents->size() > 0
```

The incorrect invariant correspond to the same case discussed for the state machines, *i.e.*, inclusion of an empty set. The subsumed invariant is explained by the fact that in all the positive models, the contents of a *DecomposableFeature* includes more than one element with lead to the condition "$> 1$" instead of the expected "$> 0$". Finally, two invariants with the iterator $forAll$ were not found.

## 4.4 Threats to Validity and Performence Issues

As for any experimental evaluation, some threats could affect the validity of our findings. Conclusion validity could be affected by the stochastic nature of our approach. To address this threat, we conducted statistical tests on a sample of executions to show that the difference in correctness between our approach and random generation is large and statistically significant. Another related threat concerns the influence of the algorithmic parameters on the obtained results. We set some of the parameters to standard or consensual values (crossover probability, population size, and number of iterations). For the others, we tested different combinations (fitness function weights and mutation probability). Mutation probability, in particular, is certainly the parameter that has the most influence on the results. Indeed, when the initial population does not contain invariants that are close the ones sought, many mutations are necessary to converge towards the optimal invariant set (see for example, [14,7]).

We identified two potential threats to the external validity. First, the models used as examples were automatically generated taking into account the sought invariants rather than collected and classified by experts as valid/invalid. To ensure that the produced models cover well the modeling space, we forced Alloy to perform the generation with different parameter values such the number of class instances in each model. In the future, we plan to conduct new experiments with more real settings to circumvent this threat. The second threat concerns the used metamodels. Although these metamodels describe different domains, the investigation of more metamodels is necessary to draw better conclusions. The manual comparison made by the authors to answer $Question2$ could represent a threat to the internal validity. Deciding for the exact invariant matches and subsumptions could be error-prone and affected by the experimenter expectancies. To prevent this threat, we conducted this comparison rigorously and diligently. We expect to use independent subjects to write/classify the models and evaluate the invariants in our future experiments.

Several implementation iterations were necessary to obtain an efficient version of our algorithm. We reused many elements that affect the performance of our algorithm, Eclipse OCL engine, Alloy model generator, and Alloy to ECORE transformer. These elements are used for each invariant set in the population and repeated trough the different evolution iterations. To obtain an acceptable performance, we first parallelized the GP process to calculate the fitness function of each invariant set in a population in separated threads. After, many trials, we created one thread per invariant set when evaluating a population. A second change, which improved considerably the performance, is the pre-calculation of the component $u(e, i)$ that is used in $f_1$ and $f_2$, *i.e.*, we pre-calculate the validity of each example model for each invariant present in the population. As many invariants are shared by many sets, and their validity is used in $f_1$ and $f_2$, the improvement was considerable. The two optimizations allowed us to run the algorithm over a input size 20 time bigger.

## 5 RelatedWork

In this section we analyze the related works to our approach from two different perspectives. The first one is the derivation of invariants, as rules learned from an underlying artifact, either models or programs. In the second perspective, we cite other works using learning techniques to derive useful information for MDE stakeholders. For the first perspective, the main referent in the derivation of invariants in software engineering is Daikon [6]. Taking a program as input, it analyzes the computed values and detects likely invariants that can be used for program understanding and documentation and verification of formal specifications among other tasks. The machine learning technique used is an inference engine based on a generate-and-check algorithm. This approach was later notoriously complemented with Sam Ratcliff's work [14]. Demonstrating that evolutionary search can consider a very wide amount of program invariants, the need for a filtering mechanism was imposed. The given solution was the use of mutation testing, enabling thus the approach to sort out invariants that are not interesting for the user. Zeller investigates the idea of specification mining[17], where he intends to leverage on repositories of software specifications, in order to reuse this knowledge into actionable recommendations for today's developers of formal specifications. The main technique for achieving specification mining is the generation of test cases covering a wide range of possible program executions - the "execution space". Test cases which lead to undesired program executions, or so-called *illegal states*, are used to enrich specifications [4].

For the second perspective, in the field of Model-Driven Engineering, machine learning techniques have been used successfully. [5] uses formal concept analysis to learn patterns of model transformation rules from a set of examples. Another application is the reverse engineering of metamodels, also known as metamodel recovery. In [10] the authors propose a mechanism to learn a metamodel from a set of models, by using techniques inspired by grammar inference. In the same fashion, [8] proposes a process for pattern extraction from deployable artifacts in order to recover architecture models. Learning of metamodels has also been presented as *bottom-up metamodeling.* In [15], authors present an approach to build metamodels from partial object models, annotated with information to build abstractions. These abstractions are refined iteratively, in order to obtain an *implementation* metamodel ready to use for MDE activities. Although this approach does not actually use search-based techniques, it does highlight the importance of guiding domain experts in the difficult task of metamodeling.

## 6 Conclusions

In this paper, we propose an approach to automatically derive well-formedness rules for metamodels. Our approach uses positive and negative example models as input and it is based on a Genetic Programming that evolves a population of random created rules, guided by a fitness function that measures how well the rules discriminate the models used as example. Once finished, the process returns the best set of well-formedness rules ever created during the process. We validate the approach over two different metamodels coming from different domains: a state machines, and feature diagrams. As a result, our approach automatically derives most of the expected well-formedness rules. This results shows the feasibility of our approach and defines a starting point for our future works. Future work includes investigating the support of more complex invariants, and alternatives in the way to obtains model examples. We are also extending our experiments to address the threats to validity mentioned in this paper. In particular, we explore the application of the approach on other various metamodels, including ones coming from industry.

## References

1. Metamodel zoos. `http://www.emn.fr/z-info/atlanmod/index.php/Zoos`.
2. J. Cadavid, B. Baudry, and H. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *Proceedings of the International Conference on Software Testing, verification and validation (ICST)*, pages –, Apr. 2012.
3. J. Cadavid, B. Combemale, and B. Baudry. Ten years of Meta-Object Facility: an Analysis of Metamodeling Practices. Tech. report RR-7882, INRIA, 2012.
4. V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96. ACM, 2010.
5. X. Dolques, M. Huchard, C. Nebut, H. Saada, et al. Formal and relational concept analysis approaches in software engineering: an overview and an application to learn model transformation patterns in examples. 2011.
6. M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.
7. M. Faunes, H. Sahraoui, and M. Boukadoum. Generating model transformation rules from examples using an evolutionary algorithm. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 250–253. ACM, 2012.
8. J. Favre. Cacophony: Metamodel-driven software architecture reconstruction. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 204–213. IEEE, 2004.
9. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
10. F. Javed, M. Mernik, J. Gray, and B. Bryant. MARS: A metamodel recovery system using grammar inference. *Information and Software Technology*, 50(9-10):948–968, 2008.
11. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.
12. J. Koza and R. Poli. Genetic programming. In *Search Methodologies*. 2005.
13. A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, 2001.
14. S. Ratcliff, D. White, and J. A. Clark. Searching for invariants using genetic programming and mutation testing. 2011.
15. J. Sánchez-Cuadrado, J. de Lara, and E. Guerra. Bottom-up meta-modelling: An interactive approach. *Model Driven Engineering Languages and Systems*, pages 3–19, 2012.
16. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2008.
17. A. Zeller. Specifications for free. In *NASA Formal Methods*, pages 2–12. Springer, 2011.

CHAPTER 7

# CONCLUSION AND FUTURE PERSPECTIVE

## 7.1 Thesis contributions and impact on MDE

The main contribution of our thesis is proposing a general approach for improving automation in MDE.

In our vision, automation problems can be solved by means of an optimization process in which solutions are searched based on examples.

We apply this vision to three MDE activities, (1) model transformation in the context of migration, (2) general model transformation and (3) modeling language definition.

For the first problem, MT in the context of migration, we propose a software clustering approach that searches for a solution by conformance with example clusters. Our approach main contribution is that, instead of guiding the clustering process based on the classic structural metrics, the search is guided by conformance with known good clusters used as examples. Based on our empirical results, we argue that driving the search by conformance to examples leads to better results than current approaches.

In the second problem, general model transformation, we contribute to the state of the art by proposing an approach to derive model transformations based on a heuristic search (genetic programming) guided by conformance with the examples of source and target models. Our approach, unlike the existing ones, does not need transformation traces, but only example pairs of source and target models. It produces complex many-to-many rules with sophisticated conditions. These rules are fully operational at the end of the derivation process. The validation on two transformation problems showed that non-trivial rules could be derived.

For the problem of modeling language definition, our contribution is proposing a search based approach to automatically improve a modeling language definition where the search is based on examples. Roughly speaking, the approach derives well-formedness rules for a metamodel starting only with two sets of valid and invalid models. To the best of our knowledge, this is the first contribution in the domain. The initial evaluation revealed encouraging results.

We believe that our contributions are important steps towards the improvement of automation level offered by the existing MDE environments. Improving automation will result in a better adoption of this paradigm by a larger part of the software industry. More concretely, contributing to the migration towards new development/design paradigms will help organizations that are struggling to prolong the life and quality of their vital software.

The derivation of transformation rules from examples is, without any doubt, the contribution that has the most impact on MDE adoption. Although much effort is dedicated to writing transformations for well-known general formalisms, e.g., UML, there is no critical mass to deal with all the domain-specific languages. Our approach gives an additional option to derive transformations by asking the stakeholder to give transformation examples rather than fully-fledged transformations.

Another reason why MDE is not widely adopted is the lack of support to produce and to process models safely. In other words, the definition of modeling languages (metamodels) is a craft activity, with tremendous consequences on the correctness and reliability of the derived software. Introducing precision is often synonym of formalizations, which requires non-tivial skills from the stakeholders. Our approach to the learning of well-formedness rules offers the possibility of giving examples of valid and invalid models letting the formal rule writing to an automated process.

## 7.2 Going beyond

Our proposal is a general framework, which requires to be applied to concrete MDE problems, like the three that we address in the thesis.

In this respect, although we provide compelling evidence that our vision can be concretely applied, the proposed approaches do not pretend to fully solve the addressed problems. There is still room for improvement.

In the first approach, in model transformation in the context of migration, a faster mechanism to test a solution quality is needed to improve the performance of the clustering process. On the other hand, the availability of bigger example cluster databases is necessary to test our proposal in industrial scenarios. Moreover, this example base could be organized by application domain since the application low level structure tends to be similar in applications of similar domains. Furthermore, the proposal needs to be tested in other scenarios beyond procedural to OO, e.g., object to component applications. That would give us an important feedback on the approach feasibility.

Many improvements are also possible for the transformation rule learning approach. First of all, a validation in an industrial setting is needed to confirm the maturity of our approach. To do so, many more example bases are needed. These example bases should consider a wide range of transformation problems. Eventually, benchmarks should be created in order to compare different team results. These tests should lead to better/faster mechanisms to guide the search (fitness functions). The second improvement opportunity is that no one has still addressed the complex problem of producing transformations in which the constructs in the target model are created by complex operations on constructs of the source model. An example is when a field in the target model is created by concatenating two fields of the source model. This kind of transformation, rare in some transformation problems, happens often in others. From another perspective, up to now, model transformation approaches only consider transformation rules that ignore the problem of controlling the rules' execution (e.g., with metarules). Explicit transformation control is often needed in model transformation used in industrial contexts. This idea is currently explored by my followers in the project. In the same vein, existing approaches consider in general that transformations search for patterns in the source model and instantiate patterns in the target model. Once again, this is a simplification since in many problems, rules have to check for the existence of constructs in the target model to connect to them the target pattern instances. Finally, it should be possible to feed the search with rules known to be correct and ask the search to preserve and complete these rules. That would facilitate the work of practitioners, since they often know the main rules (those that cover 80% of the cases), but have difficulties to write the remaining 20%.

The third approach is in its early stages. It means that many ideas of improvement are on the agenda. First, a newer mechanism to create the initial population of rules is needed to ease the convergence towards promising solution. Second, more efficient mechanisms to test a solution quality are needed. Indeed, the simple discrimination between the two sets of valid and invalid models could lead to unwanted rules. As a third initiative, other derivation operators (combination and alteration of the current rules) should be defined and tested. Moreover, as our approach is probabilistic by nature, a comprehensive study should be conducted in order to determine the best parameters to support more complex rules. And finally, better mechanisms to test the approach should be designed in order to avoid the threat to validity intrinsic to our test method. Another important idea is to extend the scope of our approach to complex well-formedness rules. Up to now, we applied our approach on a subset of simple rule types. Considering deriving complex rules will bring a new level of complexity that we have to handle.

Potvin says: "valo de req" sic.

# BIBLIOGRAPHY

[1] Hani Abdeen, Stéphane Ducasse, Houari Sahraoui, and Ilham Alloui. Automatic package coupling and cycle minimization. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 103–112. IEEE, 2009.

[2] Simon Allier, Houari A Sahraoui, Salah Sadou, and Stéphane Vaucher. Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In *Component-Based Software Engineering*, pages 216–231. Springer, 2010.

[3] Lisanne Bainbridge. Ironies of automation. *Automatica*, 19(6):775–779, 1983.

[4] Zoltán Balogh and Dániel Varró. Model transformation by example using inductive logic programming. *Software & Systems Modeling*, 8(3):347–364, 2009.

[5] Adam Baumberg and David Hogg. *Learning flexible models from image sequences*. Springer, 1994.

[6] Simona Bernardi, Susanna Donatelli, and José Merseguer. From uml sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 35–45. ACM, 2002.

[7] Jean Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[8] Juan Cadavid, Benoît Combemale, Benoit Baudry, et al. Ten years of meta-object facility: an analysis of metamodeling practices. 2012.

[9] Juan Cadavid, Benoît Combemale, Benoit Baudry, et al. Ten years of meta-object facility: an analysis of metamodeling practices. 2012.

[10] G Canfora, A Cimitile, M Munro, and M Tortorella. Experiments in identifying reusable abstract data types in program code. In *Program Comprehension, 1993. Proceedings., IEEE Second Workshop on*, pages 36–45. IEEE, 1993.

[11] Ai-ling Chen, Gen-ke Yang, and Zhi-ming Wu. Hybrid discrete particle swarm optimization algorithm for capacitated vehicle routing problem. *Journal of Zhejiang University Science A*, 7(4):607–614, 2006.

[12] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003.

[13] István Gergely Czibula and G Serban. Improving systems design using a clustering approach. *IJCSNS International Journal of Computer Science and Network Security*, 6(12):40–49, 2006.

[14] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96. ACM, 2010.

[15] Xavier Dolques, Marianne Huchard, Clémentine Nebut, and Philippe Reitz. Learning transformation rules from transformation examples: An approach based on relational concept analysis. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2010 14th IEEE International*, pages 27–32. IEEE, 2010.

[16] Russell Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pages 39–43. IEEE, 1995.

[17] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[18] Martin Faunes, Marouane Kessentini, and Houari Sahraoui. Deriving high-level abstractions from legacy software using example-driven clustering. In *Proceed-*

*ings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 188–199. IBM Corp., 2011.

[19] Martin Faunes, Marouane Kessentini, and Houari Sahraoui. Software clustering by example. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 245–246. ACM, 2011.

[20] Martin Faunes, Houari Sahraoui, and Mounir Boukadoum. Generating model transformation rules from examples using an evolutionary algorithm. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 250–253. ACM, 2012.

[21] Martin Faunes, Juan Cadavid, Benoit Baudry, Houari Sahraoui, and Benoit Combemale. Automatically searching for metamodel well-formedness rules in examples and counter-examples. In *Model Driven Engineering Languages and Systems*. ACM/IEEE, 2013.

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

[23] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*. Citeseer, 2004.

[24] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-driven engineering for software migration in a large industrial context. In *Model Driven Engineering Languages and Systems*, pages 482–497. Springer, 2007.

[25] Iván García-Magariño, Jorge J Gómez-Sanz, and Rubén Fuentes-Fernández. Model transformation by-example: an algorithm for generating many-to-many transformation rules in several model transformation languages. In *Theory and Practice of Model Transformations*, pages 52–66. Springer, 2009.

[26] Juan José Cadavid Gómez, Benoit Baudry, and Houari Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 131–140. IEEE, 2012.

[27] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106–1113. ACM, 2007.

[28] Mark Harman, Robert M Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO*, volume 2, pages 1351–1358, 2002.

[29] Reiko Heckel and Marc Lohmann. Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 82(6):33–43, 2003.

[30] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[31] Faizan Javed, Marjan Mernik, Jeff Gray, and Barrett R Bryant. Mars: A metamodel recovery system using grammar inference. *Information and Software Technology*, 50(9):948–968, 2008.

[32] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[33] Stuart Kent. Model driven engineering. In *Integrated formal methods*, pages 286–298. Springer, 2002.

[34] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model transformation as an optimization problem. In *Model Driven Engineering Languages and Systems*, pages 159–173. Springer, 2008.

[35] Marouane Kessentini, Manuel Wimmer, Houari Sahraoui, and Mounir Boukadoum. Generating transformation rules from examples for behavioral mod-

els. In *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*, page 2. ACM, 2010.

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

[37] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5-6):975–986, 1984.

[38] John R Koza and Riccardo Poli. Genetic programming. In *Search Methodologies*, pages 127–164. Springer, 2005.

[39] Philip Langer, Manuel Wimmer, and Gerti Kappel. Model-to-model transformations by demonstration. In *Theory and Practice of Model Transformations*, pages 153–167. Springer, 2010.

[40] Spiros Mancoridis, Brian S Mitchell, Chris Rorres, Y Chen, and Emden R Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 45–52. IEEE, 1998.

[41] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 50–59. IEEE, 1999.

[42] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8 (4):295–318, 1991.

[43] Raja Parasuraman, Thomas B Sheridan, and Christopher D Wickens. A model for types and levels of human interaction with automation. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 30(3):286–297, 2000.

[44] Uta Priss. Relational concept analysis: Semantic structures in dictionaries and lexical databases. 1996.

[45] Sam Ratcliff, David Robert White, and John A Clark. Searching for invariants using genetic programming and mutation testing. 2011.

[46] Mark Richters and Martin Gogolla. Validating uml models and ocl constraints. In *UML 2000 - The Unified Modeling Language*.

[47] Hajer Saada, Xavier Dolques, Marianne Huchard, Clémentine Nebut, and Houari Sahraoui. Generation of operational transformation rules from examples of model transformations. In *Model Driven Engineering Languages and Systems*, pages 546–561. Springer, 2012.

[48] Houari Sahraoui, Petko Valtchev, Idrissa Konkobo, and Shiqiang Shen. Object identification in legacy code as a grouping problem. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 689–696. IEEE, 2002.

[49] Houari A Sahraoui, Walcélio Melo, Hakim Lounis, and François Dumont. Applying concept formation methods to object identification in procedural code. In *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, pages 210–218. IEEE, 1997.

[50] Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra. Bottom-up metamodelling: An interactive approach. In *Model Driven Engineering Languages and Systems*, pages 3–19. Springer, 2012.

[51] Olaf Seng, Markus Bauer, Matthias Biehl, and Gert Pache. Search-based improvement of subsystem decompositions. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1045–1051. ACM, 2005.

[52] Thomas B Sheridan. *Telerobotics, automation and human supervisory control*. The MIT press, 1992.

[53] Michael Siff and Thomas Reps. Identifying modules via concept analysis. *Software Engineering, IEEE Transactions on*, 25(6):749–768, 1999.

[54] Michael Strommer and Manuel Wimmer. A framework for model transformation by-example: Concepts and tool support. In *Objects, Components, Models and Patterns*, pages 372–391. Springer, 2008.

[55] Yu Sun, Jules White, and Jeff Gray. Model transformation by demonstration. In *Model Driven Engineering Languages and Systems*, pages 712–726. Springer, 2009.

[56] Arie Van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st international conference on Software engineering*, pages 246–255. ACM, 1999.

[57] Dániel Varró. Model transformation by example. In *Model Driven Engineering Languages and Systems*, pages 410–424. Springer, 2006.

[58] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, 56(1):99–116, 2005.

[59] Zhi-Gang Wei, Anil P Macwan, and Peter A Wieringa. A quantitative measure for degree of automation and its relation to system performance and mental load. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 40 (2):277–295, 1998.

[60] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards model transformation generation by-example. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 285b–285b. IEEE, 2007.

[61] Chenchen Xiao and Vassilios Tzerpos. Software clustering based on dynamic dependencies. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 124–133. IEEE, 2005.

[62] Andreas Zeller. Specifications for free. In *NASA Formal Methods*, pages 2–12. Springer, 2011.

[63] Mengjie Zhang and Will Smart. Multiclass object classification using genetic programming. In *Applications of Evolutionary Computing*, pages 369–378. Springer, 2004.