Université de Montréal

Transformation by Example

par

Marouane Kessentini

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 informatique

Décembre, 2010

Université de Montréal

Faculté des arts et des sciences

Cette thèse intitulée:

Transformation by Example

Présentée par :

Marouane Kessentini

a été évaluée par un jury composé des personnes suivantes :

Résumé

La transformation de modèles consiste à transformer un modèle source en un modèle cible conformément à des méta-modèles source et cible. Nous distinguons deux types de transformations. La première est *exogène* où les méta-modèles source et cible représentent des formalismes différents et où tous les éléments du modèle source sont transformés. Quand elle concerne un même formalisme, la transformation est *endogène*. Ce type de transformation nécessite généralement deux étapes : l'identification des éléments du modèle source à transformer, puis la transformation de ces éléments. Dans le cadre de cette thèse, nous proposons trois principales contributions liées à ces problèmes de transformation. La première contribution est l'**automatisation des transformations des modèles**. Nous proposons de considérer le problème de transformation comme un problème d'optimisation combinatoire où un modèle cible peut être automatiquement généré à partir d'un nombre réduit d'exemples de transformations. Cette première contribution peut être appliquée aux transformations exogènes ou endogènes (après la détection des éléments à transformer). La deuxième contribution est liée à la transformation endogène où les éléments à transformer du modèle source doivent être détectés. Nous proposons une approche pour la **détection des défauts de conception** comme étape préalable au refactoring. Cette approche est inspirée du principe de la détection des virus par le système immunitaire humain, appelée sélection négative. L'idée consiste à utiliser de bonnes pratiques d'implémentation pour détecter les parties du code à risque. La troisième contribution vise à **tester un mécanisme de transformation** en utilisant une fonction oracle pour détecter les erreurs. Nous avons adapté le mécanisme de sélection négative qui consiste à considérer comme une erreur toute déviation entre les traces de transformation à évaluer et une base d'exemples contenant des traces de transformation de bonne qualité. La fonction oracle calcule cette dissimilarité et les erreurs sont ordonnées selon ce score. Les différentes contributions ont été évaluées sur d'importants projets et les résultats obtenus montrent leurs efficacités.

Mots-clés : Transformation de modèle, par l'exemple, défauts de conception, test des transformations, recherche heuristique, système immunitaire artificiel

Abstract

Model transformations take as input a source model and generate as output a target model. The source and target models conform to given meta-models. We distinguish between two transformation categories. *Exogenous* transformations are transformations between models expressed using different languages, and the whole source model is transformed. *Endogenous* transformations are transformations between models expressed in the same language. For endogenous transformations, two steps are needed: identifying the source model elements to transform and then applying the transformation on them. In this thesis, we propose three principal contributions. The first contribution aims to **automate model transformations.** The process is seen as an optimization problem where different transformation possibilities are evaluated and, for each possibility, a quality is associated depending on its conformity with a reference set of examples. This first contribution can be applied to exogenous as well as endogenous transformation (after determining the source model elements to transform). The second contribution is related precisely to the detection of elements concerned with endogenous transformations. In this context, we present a new technique for **design defect detection**. The detection is based on the notion that the more a code deviates from good practice, the more likely it is bad. Taking inspiration from artificial immune systems, we generate a set of detectors that characterize the ways in which a code can diverge from good practices. We then use these detectors to determine how far the code in the assessed systems deviates from normality. The third contribution concerns **transformation mechanism testing**. The proposed oracle function compares target test cases with a base of examples containing good quality transformation traces, and assigns a risk level based on the dissimilarity between the two. The traces help the tester understand the origin of an error. The three contributions are evaluated with real software projects and the obtained results confirm their efficiencies.

Keywords : Model-driven engineering, by example, design defects, search-based software engineering, artificial immune-system

Contents

Chapter 1: Introduction 16

1.1 Research Context 16

1.1.1 Automated Model Transformation.. 17

1.1.2 Automated Testing Transformation 21

1.2 Problem Statement 22

1.3 Contributions.. 25

1.4 Roadmap 27

Chapter 2: Related Work...29

2.1 Model Transformation 29

2.1.1 Exogenous Transformation 30

2.1.2 Endogenous Transformation 39

2.2 Correctness of Model Transformation 43

2.3 By-Example Software Engineering.. 48

2.4 Search-based Software Engineering 49

2.5 Summary 50

Part 1: Exogenous Transformation by Example.......................53

Chapter 3: Static Model Transformation by Example.............55

3.1 Introduction.. 55

3.2 Class Diagram to Relational Schema Transformation by Example..................... 55

Chapter 4: Dynamic Model Transformation by Example 100

4.1 Introduction.. 100

4.2 Sequence Diagrams to Colored Petri Nets Transformation by Example........... 100

REFERENCES...122

Part 2: Endogenous Transformation by Example 124

Chapter 5: An Immune-Inspired Approach for Design Defects Detection 127

5.1 Introduction 127

5.2 Design Defects Detection by Example: An Immune System Metaphor 127

Chapter 6: Design Defects Detection Rules Generation by Example 138

6.1 Introduction 138

6.2 Design Defects Detection Rules Generation by Example 139

1. Introduction 139

2. Problem Statement 140

2.1 Defintions 140

2.2 Problem Statement 140

3. Approach Overview 140

3.1 Overview 141

3.2 Problem Complexity 141

4. Search-based Rules Generation 141

4.1 Harmony Search Algorithm 141

4.2 Solution Representation 142

4.3 Evaluating Solutions 142

5. Validation 143

5.1 Goals and Objectives 143

5.2 System Studied 143

5.2 Experimental Setting 143

5.3 Results 144

5.4 Discussion 145

6. Related Work 146

7. Conclusion 146

References 147

Part 3: Model Transformation Correctness 148

Chapter 7: Testing Transformation by Example 150

7.1 Introduction 150

7.2 Testing Transformation by Example 150

Chapter 8: Conclusions 188

8.1 Contributions 188

8.2 Limitations and Future Research Directions 190

Related Publications 193

Bibliography 196

List of Tables

Table 1 By-example Approaches 51

Search-Based Model Transformation by Example

Table 1. Solution Representation 67

Table 2. 12-fold cross validation with PSO 80

Table 3. 12-fold cross validation with PSO-SA 81

Example-Based Sequence Diagrams to Colored Petri Nets Transformation Using Heuristic Search.

Table 1. Solution representation 107

Table 2. CPN size comparison 113

Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code.

Table 1 Program Statistics 129

Table 2 Results for Gantt 129

Table 3 Results for Xerces 131

Design Defects Detection Rules Generation: A Music Metaphor

Table 1 Program Statistics 140

Example-based Model Transformation Testing

Table 1. 12-fold cross validation........ 170

Table 2. Errors detected in SM5 171

List of Figures

Figure 1 Model Transformation Process 18
Figure 2 Automated Model-driven Engineering 21

Search-Based Model Transformation by Example

Fig 1. MOTOE overview 59
Fig 2. Illustration of the proposed transformation process 60
Fig 3. Class diagram metamodel 61
Fig 4. Relational schema metamodel 61
Fig 5. Example of a CLD source model 62
Fig 6. Equivalent RS target model to the CLD source model of Figure 5 63
Fig 7. Base of transformation examples and blocks generation in source model of TE4.. 64
Fig 8. Example of source model (UML-class diagram) 67
Fig 9. Change Operator in PSO 68
Fig. 10. Change Operator in SA 69
Fig 11. Fitness improvement with SA after PSO initial pass 82
Fig 12. Example-size variation with PSO 83
Fig 13. Example-size variation with PSO-SA 83
Fig 14. Execution time 84

Example-Based Sequence Diagrams to Colored Petri Nets Transformation Using Heuristic Search.

Fig. 1. (a) Example of SD (source model) and (b) his equivalent CPN (target model) .. 100
Fig. 2. Transformation solution as blocks-to-constructs mapping 103
Fig.3. Example of source model 106
Fig. 4. Change Operator in PSO 107
Fig. 5. Change Operator in SA 108

Fig. 6. Temporal coherence 110
Fig. 7. Correctness of the transformations 112

Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code.

Figure 1 Approach Overview 125
Figure 2 Global alignment 125
Figure 3 Encoding 125
Figure 4 Best alignment sequence 125
Figure 4 Effect of the number of detectors 130

Design Defects Detection Rules Generation: A Music Metaphor

Fig 1. Approach overview 137
Fig 2. Base of examples 137
Fig 3. An example of the ith harmony 140
Fig. 4. Results for Gantt 141
Fig. 5. Results for Quick UML 141
Fig. 6. Results for Xerces-J 142

Example-based Model Transformation Testing

Figure 1 Model transformation mechanism 149
Figure 2 Class diagram metamodel 151
Figure 3 Relational schema metamodel 151
Figure 4 Model transformation testing process 151
Figure 5 Overall process of our approach 155
Figure 6 Test case 157
Figure 7 Transformation unit coding 157
Figure 8 AIS-based algorithm overview 160
Figure 9 Global alignment algorithm 160

Figure 10 Best sequence alignment between U5 and T15 162
Figure 11 Transformation input: class diagram 167
Figure 12 Example-size variation 172
Figure 13 Execution time 172
Figure 14 Interactive transformation errors detection using our tool: (a) all traces and (b) only risky traces 175
Figure 15 Detectors variation vs solution quality (precision) 176
Figure 16 Detectors variation vs. execution time... 177

Acronyms

MT: Model Transformation

UML: Unified Modeling Language

MDE: Model-Driven Engineering

OCL: Object Constraints Language

MOTOE: MOdel Transformation as Optimization by Example

CPN: Colored Petri Nets

AIS: Artificial Immune System

RS: Relational Schema

API: Application Programming Interface

BON: Builder Object Network

ECL: Embedded Constraint Language

LHS: Left-Hand Side

RHS: Right-Hand Side

MTBE: Model Transformation by Example

ILP: Inductive Logic Programming

MTBD: Model Transformation by Demonstration

QVT: Query/View/Transformation

OO: Object-Oriented

DECOR: Defects dEtection CORrection

BBNs: Bayesian Belief Networks

MDA: Model-Driven Architecture

QBE: Query By Example

SBSE: Search-Based Software Engineering

SA: Simulated Annealing

GA: Genetic Algorithm

PSO: Particle Swarm Optimization

HS: Harmony Search

Je dédie cette thèse à :

Mon père **Lassâad**

Ma mère **Monia**

Pour l'incontestable soutien, le respect et toute l'affection qu'ils ont témoignés à mon égard. Qu'ils puissent trouver dans ce modeste travail la récompense de leurs énormes sacrifices.

A mes frères **Mohamed** *et* **Wael.**

A la mémoire de mon grand-père **Abdelkader**.

A tous ceux qui me sont chers.

En témoignage de ma gratitude et de ma reconnaissance.

Remerciements

Je débute cette page en remerciant le ministère de l'enseignement supérieur tunisien et l'université de Montréal pour avoir co-financé ce travail de recherche à travers plusieurs bourses d'excellence.

Je veux remercier également les membres du jury qui ont bien voulu juger cette thèse.

Je commence la liste de remerciements par l'indescriptible et hyper-motivé Professeur Houari Sahraoui qui m'a distingué, m'a accueilli et intégré dans son équipe. Il m'a guidé avec un grand dévouement, une attention de chaque jour, un suivi sans faille. Il m'a permis par son autorité respectueuse de progresser, de prendre conscience de mes responsabilités pour parvenir à réaliser ce travail en me communiquant sa passion au quotidien pour la recherche. Il a investi mon « mail » d'informations sans cesse renouvelées afin de faire progresser mon travail et maintenir un lien étroit avec les chercheurs du domaine. Les mots sont faibles pour lui exprimer ma reconnaissance.

Je tiens à remercier professeur Mounir Boukadoum pour avoir co-dirigé ma recherche et qui n'a jamais épargné l'effort de m'aider, pour ses grandes qualités humaines, pour la pertinence de ses orientations ainsi que pour la grande disponibilité dont il a fait preuve tout au long du déroulement de cette thèse. Merci mille fois pour l'énorme soutien, pour toutes les heures passées à relire les articles, ses encouragements y sont pour beaucoup dans l'aboutissement de ce travail. J'espère que je serai toujours à la hauteur de sa confiance. Qu'il trouve dans ces quelques lignes l'expression de mon profond respect et de ma réelle gratitude.

Je remercie les membres de l'équipe GEODES pour leur sympathie et leur gentillesse. Merci à Stéphane pour son humanité et ses conseils, merci à Guillaume le roi de la 3D, merci à Martin à qui je souhaite encore du courage pour sa thèse. Egalement, je remercie spécialement Jamel, Aymen, Ahmed, Dorsaf, Omar, Fleur et Hajer pour leurs bonnes humeurs.

J'exprime également ma gratitude à tous mes enseignants, qui ont contribué chacun dans son domaine, à ma formation universitaire, sans laquelle je n'aurai jamais arrivé à réaliser ce travail.

Chapter 1: Introduction

1.1 Research Context

Software engineering is concerned with the development and evolution of large and complex software-intensive systems. It covers theories, methods and tools for the specification, architecture, design, testing, and maintenance of software systems. Today's software systems are significantly large, complex and critical. Such systems cannot be developed and evolved in an economic and timely manner without automation.

Automated software engineering applies computation to software engineering activities. The goal is to partially or fully automate software engineering activities, thereby significantly increasing both quality and productivity. This includes the study of techniques for constructing, understanding, adapting and modelling both software artefacts and processes. Automatic and collaborative systems are both important areas of automated software engineering, as are computational models of human software engineering activities. Knowledge representations and artificial intelligence techniques that can be applied in this field are of particular interest; they represent formal and semi-formal techniques that provide or support theoretical foundations.

Automated software engineering approaches have been applied in many areas of software engineering. These include requirements engineering, specification, architecture, design and synthesis, implementation, modelling, testing and quality assurance, verification and validation, maintenance and evolution, reengineering, and visualisation [40], [64]. This thesis is concerned with two important fields of automated software engineering: (1) model driven engineering and (2) maintenance. The contributions to the first field consist of model transformation automation and testing using different techniques; those to the second field include two tasks, of which the improvement of code quality by automating the detection and correction of bad programming practices. This task can be viewed as a special kind of transformation, called endogenous transformation, where the source and target models are the same. The second task is the validation of a transformation mechanism in order to detect potential errors.

1.1.1 Automated Model Transformation

A first distinction concerns the kinds of software artefacts being transformed. If they are programs (i.e., source code, bytecode, or machine code), we use the term program transformation; if they are models, we use the term model transformation (MT). In our view, the latter term encompasses the former, since a model can range from abstract analysis models to very concrete models of source code. Hence, model transformations also include transformations from a more abstract to a more concrete model (e.g., from design to code) and vice versa (e.g., in a reverse engineering context). Model transformations are obviously needed in common tools such as code generators and parsers.

Kleppe et al. [5] provide the following definition of model transformation, as illustrated in Figure 17: a transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

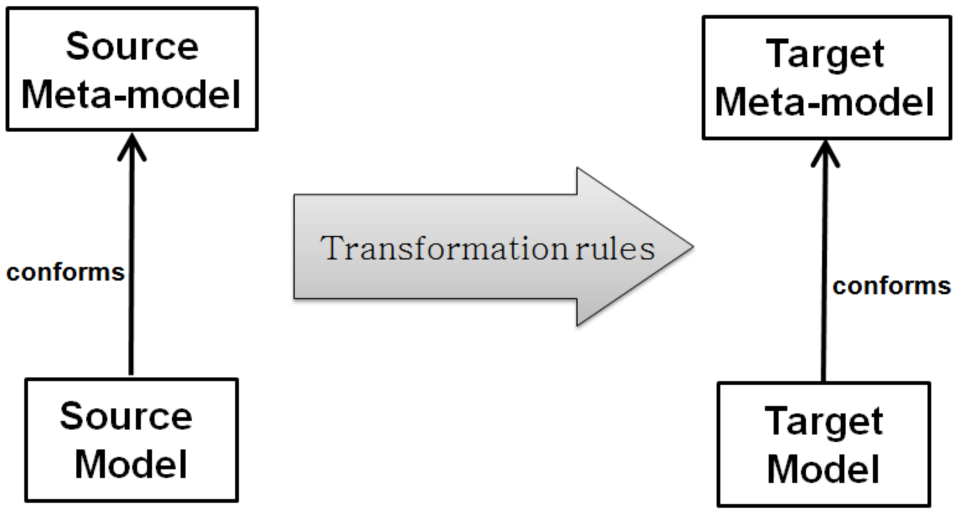


Figure 17 Model Transformation Process

In order to transform models, they need to be expressed in some modeling language (e.g., UML for design models, and programming languages for source code models). The syntax and semantics of the modeling language itself are expressed by a meta-model (e.g., the UML meta-model). Based on the language in which the source and target models of a transformation are expressed, a distinction can be made between endogenous and exogenous transformations.

Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages. This distinction is essentially the same as the one that is proposed in the "Taxonomy of Program Transformation" [113], but ported to a model transformation setting. In this taxonomy, the term rephrasing is used for an endogenous transformation, while the term translation is used for an exogenous transformation.

Typical examples of translation (i.e., exogenous transformation) are:

– Synthesis of a higher-level, more abstract, specification (e.g., an analysis or design model) into a lower-level, more concrete, one (e.g., a model of a Java program). A typical example of synthesis is code generation, where the source code is translated into byte-code (that runs on a virtual machine) or executable code, or where the design models are translated into source code.

– Reverse engineering is the inverse of synthesis and extracts a higher-level specification from a lower-level one.

– Migration from a program written in one language to another, but keeping the same level of abstraction.

Typical examples of rephrasing (i.e., endogenous transformation) are:

– Optimization, a transformation aimed to improve certain operational qualities (e.g., performance), while preserving the semantics of the software.

– Refactoring, a change to the internal structure of software to improve certain software quality characteristics (such as understandability, modifiability, reusability, modularity, adaptability) without changing its observable behaviour.

As shown in Figure 18, there is a principal difference between endogenous and exogenous transformation. In the first category, we transform the whole source model to this equivalent target model conforming to different meta-models. However, for the second category we have two steps. The first one consists of identifying the elements to transform in the source model and then the second step consists of transforming these elements.

The endogenous transformations are principally related to maintenance activities (refactoring, performance, etc). In modern software development, maintenance accounts for the majority of the total cost and effort in a software project. Especially burdensome are those tasks which require applying a new technology in order to adapt an application to changed requirements or a different environment. The high cost of software maintenance could be reduced by automatically improving the design of object-oriented programs without altering their behaviour [64].

The potential benefit of automated adaptive maintenance tools is not limited to a single domain, but rather spans a broad spectrum of modern software development. The primary concern of developers is to produce highly efficient and optimized code, capable of solving intense scientific and engineering problems in a minimal amount of time. One of the important issues in automated software maintenance is to propose automated tools that improve software quality. Indeed, in order to limit costs and improve the quality of their software systems, companies try to enforce good design development practices and similarly to avoid bad practices.

The underlying assumption is that good practices will produce good software. As a result, these practices have been studied by professionals and researchers alike with a special attention given to design-level problems. There has been much research focusing on the study of bad design practices sometimes called defects, antipatterns [61], smells [10], or anomalies [84] in the literature. Although bad practices are sometimes unavoidable, in most cases, a development team should try to prevent them and remove them from their code base as early as possible. Thus, we define code transformation as the process related to modifying the code in order to eliminate detected defects and improve the quality of the software. Hence, many fully-automated detection and correction techniques have been proposed [89]. Like in model transformation, the vast majority of existing work in design defects detection and correction is rule-based. Different rules identify key features that characterize anomalies using combinations of techniques like metrics, structural analysis, and/or lexical information.

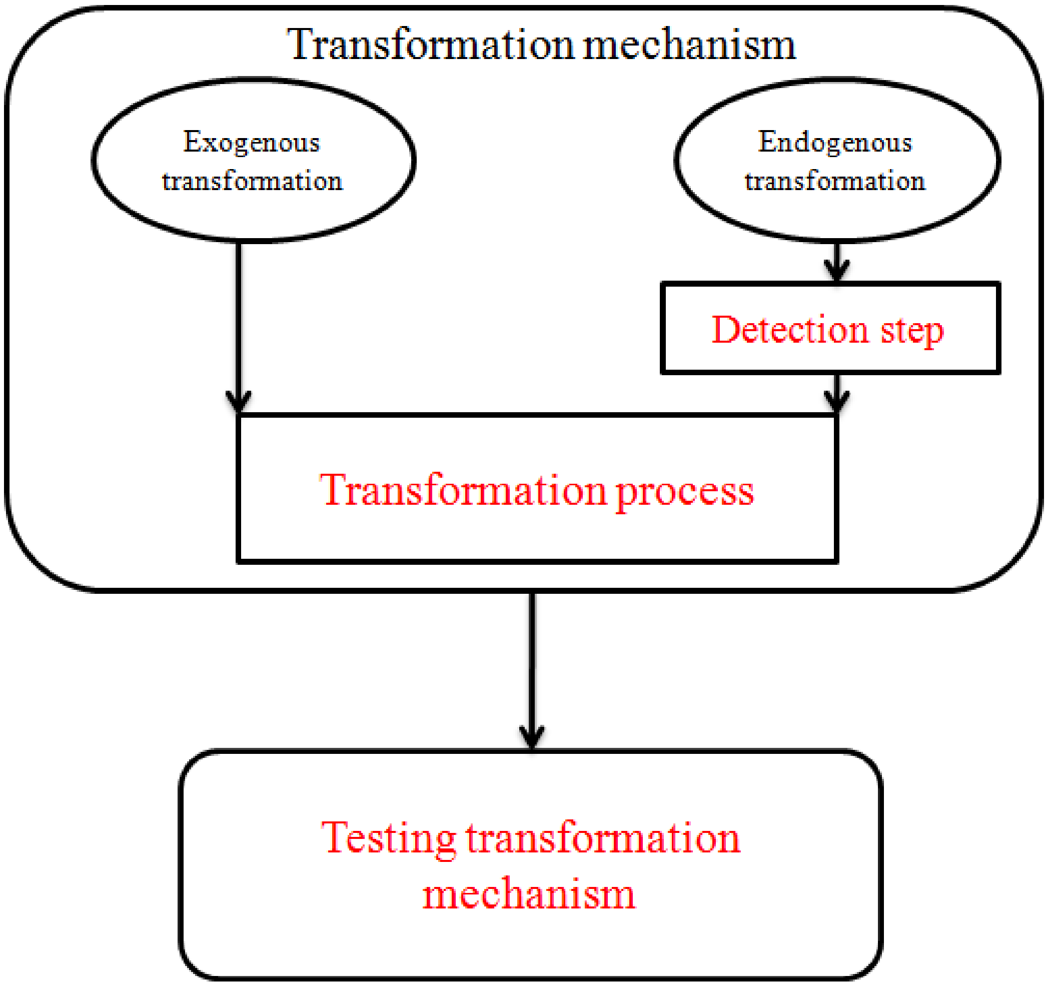


Figure 18 Automated Model-driven Engineering

Figure 2 summarizes the different tasks to automate in this thesis. We have detailed in this section the first part about endogenous and exogenous transformations. The second part about validating a transformation mechanism will be detailed in the next section.

1.1.2 Automated Testing Transformation

As the specification of automated model transformations can also be erroneous, it necessitates finding automated ways to verify the correctness of a model transformation. Indeed, the automated verification of model transformation results represents another important issue in automated model driven engineering. If a transformation is not correct, it may inject errors in the system design. Thus, it is pertinent to have an upstream validation and verification process in order to detect errors as soon as possible, rather than dragging them on all along. The verification increases the reliability and the usability of model transformations [40]. Furthermore, automated verification may significantly reduce the duration, and ultimately the total cost, of performing a model transformation.

To validate the transformation mechanisms, we distinguish between two main categories: formal verification and testing. For proving the correctness of a system model by formal verification, a large number of semi-automated tools exist, based on model checking or theorem proving [22],[3]. They can typically draw more general conclusions on a model by using theorem provers. However, their use requires a significant amount of mathematical expertise and user interaction (not fully automated). Model transformation testing typically consists of synthesizing a large number of different input models as test cases, running the transformation mechanism and verifying the result using an oracle function. In this context, two important issues must be addressed: the efficient generation/selection of test cases and the definition of the oracle function to analyze the validity of transformed models. Testing transformation mechanisms is an approximate method and represents the main difference with formal methods. The definition of an oracle function for model transformation testing is a challenge [64],[89] and requires addressing many problems as detailed in the next section.

1.2 Problem Statement

As shown in the previous section, we distinguish between three main problems.

Part 1: Automating model transformation

Problem 1.1: Most of the available work on model transformation is based on the hypothesis that transformation rules exist and that the important issue is how to express them. However, in real problems, the rules may be difficult to define; this is often the case when the source and/or target formalisms are not widely used or proprietary. Indeed, as for any rule-based system, defining the set of rules is not an obvious task and the following obstacles may hinder the results:

(1) **Incompleteness** or missing rules in a rule base. As a result, useful information cannot be derived from the rule base. In the context of model transformation, the result of incompleteness can be viewed as a partial generation of the target model.

(2) **Inconsistency** or conflicting rules. Defining individual transformation rules is not a fastidious task. However, ensuring coherency between the individual rules is not obvious and can be very difficult given the dependencies between model elements while applying transformation rules.

(3) **Redundancy** or the existence of duplicated (identical) or derivable (subsumed) rules in the rule base. Redundant rules not only increase the size of the rule base but may cause useless additional inferences.

Problem 1.2: In the case of dynamic models (e.g., sequence diagram to colored Petri nets), the definition of transformation rules is more difficult: In addition to the problems mentioned previously, dynamic models must consider order (time sequencing) while transforming model elements (composition). Furthermore, in the case of dynamic models, the systematic use of rules generates target models that may need to be optimized in terms of size and structures.

Part 2: Exogenous transformation (design defects detection)

The next five problems are related to design defect detection related to exogenous transformation.

Problem 2.1: There is no exhaustive list of all possible types of design defects. Although there has been a significant work to classify defect types, programming practices, paradigms and languages evolve making it unrealistic for them to permanently support the detection of all possible defect types. Furthermore, there might be company or application-specific design practices.

Problem 2.2: For those design defects that are documented, there is no consensual definition of the symptoms and their severity of impact on the code. Defects are generally described using natural language and their detection relies on the interpretation of the developers. This is a major setback for automation.

Problem 2.3: The majority of detection methods do not provide an efficient manner to guide the manual inspection of the candidate list. Potential defects are generally not listed in an order that helps developers address the most important ones first. There exist few works, such as the one of Khomh et al [89], where probabilities are used to order the results.

Problem 2.4: How to define thresholds when dealing with quantitative information? For example, the Blob [8] detection involves information such as class size. Although, we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given program/community of users could be considered average in another.

Problem 2.5: How to deal with the context? In some contexts, an apparent violation of a design principle is considered as a consensual practice. For example, a class Log responsible for maintaining a log of events in a program, used by a large number of classes, is a common and acceptable practice. However, from a strict defect definition, it can be considered as a class with abnormally large coupling.

Part 3: Testing model transformation

After defining a transformation mechanism, it is necessary to validate it. However, some limitations in exiting work:

Problem 3.1: Current model-driven engineering (MDE) technologies and model repositories store and manipulate models as graphs of objects. Thus, when the expected output model is available, the oracle compares two graphs. In this case, the oracle definition problem has the same complexity as the graph isomorphism problem, which is NP-hard [121]. In particular, we can find a test case output and an expected model that look different (contain different model elements), but have the same meaning. So, the complexity of these data structures makes it difficult to provide an efficient and reliable tool for comparison.

Problem 3.2: the majority of existing works are based on constraints verification. The constraints are defined at the metamodel level and conditions are generally expressed in OCL. However, the number of constraints to define can be very large to cover all rules and patterns. This is especially the case of contracts related to one-to-many mappings. Moreover, being formal specifications, these constraints are difficult to write in practice [21].

Problem 3.3: transformation errors can have different causes: transformation logic (rules) or source/ target meta-models [23]. To be effective, a testing process should allow the identification of the cause of errors.

1.3 Contributions

To overcome the previously identified problems, we propose the following contributions, organized in three major parts:

Part 1: Model transformation by example

Contribution 1.1 We propose an approach for model transformation that does not use or produce transformation rules. We start from the premise that experts give transformation examples more easily than complete and consistent transformations rules. In the absence of rules or an exhaustive set of examples, an alternative solution is to derive a partial target model from the available examples. The generation of such a model consists of finding situations in the examples that best match the model to transform. Thus, we propose the alternative view of MT as an optimization problem where a (partial) target model can be automatically derived from available examples. For this, we introduce a search-based approach to automate MT called MOTOE (model transformation as optimization by examples) [79],[74].

Contribution 1.2 We extend MOTOE to the case of dynamic model transformation, e.g., sequence diagram to colored petri net (CPN). The primary goal is to add to contribution 1.1 the constraint of temporal coherence during the transformation process. Another goal is to generate optimal target models (in terms of size) by using good example bases.

Part 2: Design Defects Detection by example

Contribution 2.1 In this effort, we view the detection of design defects as one that can be addressed by the mechanisms of detection-identification-response of an artificial immune system (AIS), which use the metaphor of a biological immune system. In both cases, known and unknown problems should be discovered. Instead of trying to find all possible infections, an immune system starts by detecting what is not normal. The more an element is abnormal, the more it is considered risky. This first phase is called discovery. After the risk has been assessed, the next phases consist of identifying if the risk corresponds to a known category of problems and subsequently producing the proper response. Similarly, our contribution is built on the idea that the higher the dissimilarity between a code fragment and a reference (good) code, the higher is the risk that this code could constitute a design defect. The efficiency of our approach is evaluated by studying the relationship between dissimilarity and risk for different open source projects.

Contribution 2.2 we propose another solution by using examples of manually found design defects to derive detection rules. Such examples are in general available as documents as par of the maintenance activity (version control logs, incident reports, inspection reports, etc.). The use of examples allows the derivation of rules that are specific to a particular company rather than rules that are supposed to be applicable to any context. This includes the definition of thresholds that correspond to the company best practices. Learning from examples aims also at reducing the list of detected defect candidates. Our approach allows to automatically find detection rules, thus relieving the designer from doing so manually. Rules are defined as combinations of metrics/thresholds that better conform to known instances of design defects (defect examples). In our setting, we use a music-inspired algorithm [56] for rule extraction. We evaluate our approach by finding potential defects in three different open-source systems.

Part 3: Testing Transformation by example

Contribution 3. We also adapt the by-example approach based on the immune system metaphor to automate the test of transformation mechanisms. We propose an oracle function that compares target test cases to the elements of a base of examples containing good quality transformation traces, and then assigns a risk level to the former, based on dissimilarity between the two as determined by an AIS algorithm. As a result, one no longer needs to define an expected model for each test case and the traceability links help the tester understand the origin of an error. Furthermore, the detected faults are ordered by degree of risk to help the tester perform further analysis. For this, a custom tool was developed to visualize the risky fragments found in the test cases with different colors, each related to an obtained risk score. We illustrate and evaluate our approach with different transformation mechanisms.

1.4 Roadmap

The remainder of this dissertation is organized as follows:

Chapter 2 reviews related work on model transformation, design defect detection, transformation testing, by-example software engineering and search-based software engineering; Chapter 3 reports our contribution for automating model transformation using examples and search-based techniques. We present our Software and System Modeling journal paper [79] that shows an illustration of our approach for the case of static transformation. For dynamic transformation, our European Conference on Modelling Foundations and Applications [74] illustrates the application of our approach to sequence diagram to colored Petri nets transformation. Chapter 4 presents our approach to design defects detection based on an immune system metaphor. This contribution is illustrated via our Automated Software Engineering conference paper [82]. Chapter 5 details our contribution for design defects rules generation. We present in this chapter our European Conference on Software Maintenance and Reengineering paper [80]. Chapter 6 presents a description for our contribution about testing transformation mechanism by example [78]. It is subject to a paper accepted in the Journal of Automated Software Engineering. Chapter 7 presents the conclusions of this dissertation and outlines some directions for future research.

Chapter 2: Related Work

This chapter gives an overview of basic works related to this thesis. The work proposed in this thesis crosscuts four research areas: (1) endogenous and exogenous transformations; (2) correctness of model transformation; (3) by-example software engineering; and (4) search-based software engineering. The chapter provides a survey of existing works in these four areas and identifies the limitations that are addressed by our contributions.

The structure of the chapter is as follows: Section 2.1 summarises exiting works in model transformation, including endogenous and exogenous transformations. We identify different criteria to identify them and we focus on by-example approaches. Section 2.2 discusses the state of the art in validating transformation mechanisms; Section 2.3 is devoted toward describing work based on the use of examples; Section 2.4 provides a description of leading work in search-based software engineering.

2.1 Model Transformation

Model transformation programs take as input a model conforming to a given source meta-model and produce as output another model conforming to a target meta-model. The transformation program, composed of a set of rules, should itself be considered a model. Consequently, it has a corresponding meta-model that is an abstract definition of the used transformation language.

As previously stated, we distinguish between two model transformation categories: (1) *exogenous* transformations in which the source and target meta-models are not the same, e.g., transforming a UML class diagram to Java code, and (2) *endogenous* transformations in which the source and target meta-models are the same, e.g., refactoring a

UML class diagram or code. Exogenous transformations are used to exploit the constructive nature of models in terms of vertical transformations, thereby changing the level of abstraction and building the bases for code generation, and also to allow horizontal transformation of models that are at the same level of abstraction [13]. Horizontal transformations are of specific interest to realize different integration scenarios such as model translation, e.g., translating a relational schema (RS) model into a UML class model. In contradistinction to exogenous transformations where the entire source model elements must be transformed to their equivalents in the target model, we distinguish two steps in endogenous transformations. The first step is the identification of source model elements (only some model fragments) to transform, and the second step is the transformation itself. In most cases, the endogenous transformations correspond to model refactoring where the input and output meta-model are the same. In this case, the first step is the detection of refactoring opportunities (e.g., design defects), and the second one is the application of refactoring operations (transformation).

We now describe existing work according to these two categories: endogenous and exogenous transformation. For endogenous transformation, we focus on refactoring activities.

2.1.1 Exogenous Transformation

2.1.1.1 Classification and Languages

In the following, a classification of endogenous transformation approaches is briefly reported. Then, some of the available endogenous transformation languages are separately described. The classification is mainly based upon [110] and [13].

Several endogenous transformation approaches have been proposed in the literature. In the following, classifications of model-to-model endogenous transformation approaches discussed by Czarnecki and Helsen [110] are described:

Direct manipulation approach. It offers an internal model representation and some APIs to manipulate it. It is usually implemented as an object-oriented framework, which may also provide some minimal infrastructure. Users have to implement transformation rules, scheduling, tracing and other facilities in a programming language.

An example of used tools in direct manipulation approaches is Builder Object Network (BON), a framework which is relatively easy to use and is still powerful enough for most applications. BON provides a network of C++ objects. It provides navigation and update capabilities for models using C++ for direct manipulation.

Operational approach. It is similar to direct manipulation, but offers more dedicated support for model transformation. A typical solution in this category is to extend the utilized meta-modeling formalism with facilities for expressing computations. An example would be to extend a query language such as OCL with imperative constructs. Examples of systems in this category are Embedded Constraint Language (ECL) [50], QVT Operational mappings[91], XMF [122], MTL [26] and Kermeta [49].

Relational approach. It groups declarative approaches in which the main concept is mathematical relations. In general, relational approaches can be seen as a form of constraint solving. The basic idea is to specify the relations among source and target element types using constraints that, in general, are non-executable. However, the declarative constraints can be given executable semantics, such as in logic programming where predicates can describe the relations. All of the relational approaches are side-effect free and, in contrast to the imperative direct manipulation approaches, create target elements implicitly. Relational approaches can naturally support multidirectional rules. They sometimes also provide backtracking. Most relational approaches require strict separation between source and target models, that is, they do not allow in-place update. Examples of relational approaches are QVT Relations and ATL [36]. Moreover, in 14] the application of logic programming has been explored for the purpose.

Graph-transformation based approaches. They exploit theoretical work on graph transformations and require that the source and target models be given as graphs. Performing model transformation by graph transformation means to take the abstract syntax graph of a model, and to transform it according to certain transformation rules. The result is the syntax graph of the target model. More precisely, graph transformation rules have an LHS and an RHS graph pattern. The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place. In particular, LHS represents the pre-conditions of the given rule, while RHS describes the post-conditions. LHS∩RHS defines a part which has to exist to apply the rule, but which is not changed. LHS − LHS ∩ RHS defines the part which shall be deleted, and RHS − LHS ∩ RHS defines the part to be created. The LHS often contains conditions in addition to the LHS pattern, for example, negative conditions. Some additional logic is needed to compute target attribute values such as element names. GReAT [38] and AToM3 [54] are systems directly implementing the theoretical approach to attributed graphs and transformations on such graphs. They have built-in fixed point scheduling with non-deterministic rule selection and concurrent application to all matching locations.

Mens et al [13] provide a taxonomy of model transformations. One of the main differences with the previous taxonomy is that Czarnecki and Helsen propose a hierarchical classification based on feature diagrams, while the Mens et al. taxonomy is essentially multi-dimensional. Another important difference is that Czarnecki et al. classify the specification of model transformations, whereas Mens et al. taxonomy is more targeted towards tools, techniques and formalisms supporting the activity of model transformation.

For these different categories, many languages and tools have been proposed to specify and execute exogenous transformation programs. In 2002, OMG issued the Query/View/Transformation request for proposal [91] to define a standard transformation language. Even though a final specification was adopted at the end of 2008, the area of model transformation continues to be a subject of intense research. Over the last years, in parallel to the OMG effort, a number of model transformation approaches have been proposed both from academia and industry. They can be distinguished by the used paradigms, constructs, modeling approaches, tool support, and suitability for given problems. We briefly describe next some well-known languages and tools.

ATL (ATLAS Transformation Language) [35] is a hybrid model transformation language that contains a mixture of declarative and imperative constructs. The former allows dealing with simple model transformations, while the imperative part helps in coping with transformations of higher complexity. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. During the execution of a transformation, source models may be navigated through, but changes are not allowed. Transformation definitions in ATL form modules. A module contains a mandatory header section, import section, and a number of helpers and transformation rules. There is an associated ATL Development Toolkit available as open source from the GMT Eclipse Modeling Project [28]. A large library of transformations is available at [15], [43].

GReAT [1] (Graph Rewriting and Transformation Language) is a meta-model-based graph transformation language that supports the high-level specification of complex model transformation programs. In this language, one describes the transformations as sequenced graph rewriting rules that operate on the input models and construct an output model. The rules specify complex rewriting operations in the form of a matching pattern and a subgraph to be created as the result of the application of a rule. The rules (1) always operate in a context that is a specific subgraph of the input, and (2) are explicitly sequenced for efficient execution. The rules are specified visually using a graphical model builder tool called GME [2].

AGG is a development environment for attributed graph transformation systems that support an algebraic approach to graph transformation. It aims at specifying and rapid prototyping applications with complex, graph structured data. AGG supports typed graph transformations including type inheritance and multiplicities. It may be used (implicitly in "code") as a general-purpose graph transformation engine in high-level Java applications employing graph transformation methods.

The source, target, and common meta-models are represented by type graphs. Graphs may additionally be attributed using Java code. Model transformations are specified by graph rewriting rules that are applied non-deterministically until none of them can be applied anymore. If an explicit application order is required, rules can be grouped in ordered layers. AGG features rules with negative application conditions to specify patterns that prevent rule executions. Finally, AGG offers validation support that is consistency checking of graphs and graph transformation systems according to graph constraints, critical pair analysis to find conflicts between rules (that could lead to a non-deterministic result) and checking of termination criteria for graph transformation systems. An available tool support provides graphical editors for graphs and rules and an integrated textual editor for Java expressions. Moreover, visual interpretation and validation is supported.

VIATRA2 [118] is an Eclipse-based general-purpose model transformation engineering framework intended to support the entire life-cycle for the specification, design, execution, validation and maintenance of transformations within and between various modelling languages and domains. Its rule specification language is a unidirectional transformation language based mainly on graph transformation techniques. More precisely, the basic concept in defining model transformations within VIATRA2 is the (graph) pattern. A pattern is a collection of model elements arranged into a certain structure fulfilling additional constraints (as defined by attribute conditions or other patterns). Patterns can be matched on certain model instances, and upon successful pattern matching, elementary model manipulation is specified by graph transformation rules. There is no predefined order of execution of the transformation rules. Graph transformation rules are assembled into complex model transformations by abstract state machine rules, which provide a set of commonly used imperative control structures with precise semantics.

VIATRA2 is a hybrid language since the transformation rule language is declarative, but the rules cannot be executed without an execution strategy specified in an imperative manner. Important specification features of VIATRA2 include recursive (graph) patterns, negative patterns with arbitrary depth of negation, and generic and meta-transformations (type parameters, rules manipulating other rules) for providing reuse of transformations [118].

A conclusion to be drawn from studying the existing endogenous transformation approaches, tools and techniques is that they are often based on empirically obtained rules

[5]. In fact, the traditional and common approach toward implementing model transformations is to specify the transformation rules and automate the transformation process by using an executable model transformation language. Although most of these languages are already powerful enough to implement large-scale and complex model transformation tasks, they may present challenges to users, particularly to those who are unfamiliar with a specific transformation language. Firstly, even though declarative expressions are supported in most model transformation languages, they may not be at the proper level of abstraction for an end-user, and may result in a steep learning curve and high training cost. Moreover, the transformation rules are usually defined at the meta-model level, which requires a clear and deep understanding about the abstract syntax and semantic interrelationships between the source and target models. In some cases, domain concepts may be hidden in the meta-model and difficult to unveil (e.g., some concepts are hidden in attributes or association ends, rather than being represented as first-class entities). These implicit concepts make writing transformation rules challenging. Thus, the difficulty of specifying transformation rules at the meta-model level and the associated learning curve may prevent some domain experts from building model transformations for which they have extensive domain experience.

To address these challenges inherent from using model transformation languages, an innovative approach called Model Transformation By Example (MTBE) is proposed that will be described in the next section.

2.1.1.2 Model Transformation by Example

The commonalities of the by-example approaches for transformation can be summarized as follows: All approaches define an example as a triple consisting of an input model and its equivalent output model, and traces between the input and output model elements. These examples have to be established by the user, preferably in concrete syntax. Then, generalization techniques such as hard-coded reasoning rules, inductive logic, or relational concept analysis are used to derive model transformation rules from the examples, in a deterministic way that is applicable for all possible input models which have a high similarity with the predefined examples.

Varrò and Balogh [23] propose a semi-automated process for MTBE using Inductive Logic Programming (ILP). The principle of their approach is to derive transformation rules semi-automatically from an initial prototypical set of interrelated source and target models. Another similar work is that of Wimmer et al [31] who derive ATL transformation rules from examples of business process models. Both contributions use semantic correspondences between models to derive rules. Their differences include the fact that [31] presents an object-based approach that finally derives ATL rules for model transformation, while [41] derives graph transformation rules. Another difference is that they respectively use abstract versus concrete syntax: Varro uses IPL when Wimmer relies on an *ad hoc* technique. Both models are heavily dependent on the source and target formalisms. Another similar approach is that of Dolques et al. [123] which aims to alleviate the writing of transformations, and where engineers only need to handle models in their usual (concrete) syntax and to describe the main cases of a transformation, namely the examples. A transformation example includes the source model, the target model and trace links that make explicit how elements from the source model are transformed into elements of the target model. The transformation rules are generated from the transformation traces, using formal concept analysis extended by relations, and they are classified through a lattice that helps navigation and choice. This approach requires the examples to cover all the transformation possibilities and it is only applicable for one-to-one transformations.

Recently, a similar approach to MTBE, called Model Transformation by Demonstration (MTBD), was proposed [124]. Instead of the MTBE idea of inferring the rules from a prototypical set of mappings, users are asked to demonstrate how the MT should be done, through direct editing (e.g., add, delete, connect, update) of the source model, so as to simulate the transformation process. A recording and inference engine was developed, as part of a prototype called MT-Scribe, to capture user operations and infer a user's intention during a MT task. A transformation pattern is then generated from the inference, specifying the preconditions of the transformation and the sequence of operations needed to realize the transformation. This pattern can be reused by automatically matching the preconditions in a new model instance and replaying the necessary operations to simulate the MT process. However, this approach needs a large number of simulated patterns to be efficient and it requires a high level of user intervention. In fact, the user must choose the suitable transformation pattern. Finally, the authors do not show how MTBD can be useful to transform an entire source model and only provide examples of transforming model fragments. On the other hand, the MTBD approach, in contradiction with others by-example approaches is applied to endogenous transformations. Another very similar by demonstration approach was proposed by Langer et al. [97]. The difference with Sun et al. work, that uses the recorded fragments directly, Langer et al. use them to generate ATL rules. Another difference is that Langler approach is related to exogenous transformation.

Brosch et al.[96] introduced a tool for defining composite operations, such as refactorings, for software models in a user-friendly way. This by-example approach prevents modelers from acquiring deep knowledge about the metamodel and dedicated model transformation languages. However, this tool able only to apply refactoring operations and do not detect automatically refactoring operations.

The commonalities of the by-example approaches for exogenous transformation can be summarized as follows: All approaches define an example as a triple consisting of an input model and its equivalent output model, and traces between the input and output model elements. The examples have to be established by the user, preferably in concrete syntax. Then, generalization techniques such as hard-coded reasoning rules, inductive logic or relational concept analysis are used to derive model transformation rules from the examples, in a deterministic way that is applicable to all possible input models which have a high similarity with the predefined examples.

None of the mentioned approaches claims that the generation of the model transformation rules is correct or complete. In particular, all approaches explicitly state that some complex parts of the transformation involving complex queries, attribute calculations such as aggregation of values, non-deterministic transformations, and counting of elements have to be developed by the user, by changing the generated model transformations. Furthermore, the approaches recommend developing the model transformations using an iterative methodology. This means that, after generating the transformations from initial examples, the examples must be adjusted or the transformation rules changed if the user is not satisfied with the outcome. However, in most cases, deciding that the examples or the transformation rules need changing is not an obvious process to the user.

2.1.1.3 Traceability-based Model Transformation

Some other meta-model matching works can also be considered as variants of by-example approaches. Garcia-Magarino et al. [46] propose an approach to generate transformation rules between two meta-models that satisfy some constraints introduced manually by the developer. In [47], the authors propose to automatically capture some transformation patterns in order to generate matching rules at the meta-model level. This approach is similar to MTBD, but it is used at the meta-model level.

Most current transformation languages [66],[37],[58] build an internal traceability model that can be interrogated at execution time, for example, to check if a target element was already created for a given source element. This approach is specific to each transformation language and sometimes to the individual transformation specification. The language determines the traceability meta-model and the transformation specification determines the label of the traces (in case of QVT/Relational the traceability meta-model is deduced from the transformation specification). The approach taken only provides access to the traces produced within the scope of the current transformation. Marvie describes a transformation composition framework [100] that allows manual creation of linkings (traces). These linkings can then be accessed by subsequent transformation, although this is limited to searching specific traces by name, introducing tight coupling between sub-transformations.

2.1.2 Endogenous Transformation

In contradistinction to exogenous transformations where the entire source model elements must be transformed to their equivalents in the target model, we distinguish two steps in endogenous transformations. The first step is the identification of source model elements (only some model fragments) to transform, and the second step is the transformation itself. In most cases, the endogenous transformations correspond to model refactoring where the input and output meta-model are the same. In this case, the first step is the detection of refactoring opportunities (e.g., design defects) and the second one is the application of refactoring operations (transformation).

In this thesis, we focus on program-code transformation that represents the major parts of existing work in exogenous transformation.

Code transformation can be performed as model transformations. In fact, a programming language have a defined meta-model (for example: JAVA) and a program can be considered as an instance of this metamodel. Given that all code transformations can be performed as model transformations, one can classify the source and target models of a transformation in terms of their structure. Code transformation has applications in many areas of software engineering such as compilation, optimization, refactoring, program synthesis, software renovation, and reverse engineering. The aim of code transformation is to increase programmer productivity by automating programming tasks, thus enabling programming at a higher-level of abstraction, and increasing maintainability and re-usability. In our work, we are interested in code transformation as the identification and correction of design defects in code using refactoring. The term refactoring, introduced by Opdyke in his PhD thesis [90], refers to “the process of changing an [object-oriented] software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure". Refactoring can be regarded as the object-oriented equivalent of restructuring, which is defined by Chikofsky and Cross [31] as "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics). [...] it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes to improve aspects of the system." In other words, the refactoring process consists of a number of activities: (1) identify where the software should be refactored; (2) determine which refactorings should be applied to the identified places; (3) guarantee that the applied refactoring preserves behaviour; (4) apply the refactoring; (5) assess the effect of refactoring on software quality characteristics; (6) maintain consistency between refactored program code and other software artifacts (or vice versa). Each of these activities could be automated to a certain extent.

Several studies have recently focused on detecting and correction (by applying refactorings) of design defects in software using different techniques. These techniques range from fully automatic detection techniques [99],[89], to manual inspection techniques. This section can be separated in three broad categories: metric-based approaches, correction opportunity based approaches, graph transformation and visualization.

2.1.2.1 Metric-based Approaches

Marinescu [99] defined a list of rules based on metrics to detect design flaws of OO design at method, class and subsystem levels. However, the choice of the metrics to use and the proper threshold values for those metrics are not addressed explicitly in his research. Erni et al. [31] introduce the concept of multi-metrics, as an n-tuple of metrics expressing a quality criterion (e.g., modularity). Unfortunately, multi-metrics neither encapsulate metrics in a more abstract construct, nor do they allow a flexible combination of metrics. Alikacem et al. [64] express metrics in a generic manner based on fuzzy logic rules. However, they use their technique only for rule activation to detect a defect and not to estimate the probability of design defect occurrence.

In general, many limitations are related to the use of metrics. Also, the use of specific metrics does not consider context, and we need to adapt the related rules by hand to the context of use. Even for a single system, this task can be costly because of constant evolution. Another issue is the different interpretations of defect definitions by analysts. A final problem is the use of threshold values. Different systems can follow different development practices. Consequently, different thresholds might apply. These issues were partially addressed by Moha et al. [88] in their framework DECOR. They automatically convert high-level defect specifications into detection algorithms. Theoretically, the exact metrics used for the detection could vary, but this issue was almost not studied in practice. This explains the high number of false positives they detected. An additional problem is that the detected defects are not ordered. This implies that a maintainer does not have a clear idea of which possible defects should be inspected first. Khomh et al. [89] extended DECOR to support uncertainty in smell detection: they used Bayesian belief networks (BBNs) to implement rules from DECOR. The output of their model is probabilities that classes are occurrences of design defects. Although the technique allows ranking of the results (by probability), it still suffers from the problem of selecting specific metrics to conduct a detection.

2.1.2.2 Visualization-based techniques

The need for visualization-based defect detection has been proposed to take advantage of the expertise of analysts. Visualization is considered to be a semi-automatic technique since the information is automatically extracted and then presented to an analyst for interpretation. Kothari et al. [60] present a pattern-based framework for developing tool support to detect software anomalies by representing potential defects with different colors using a specific metaphor. Dhambri et al. [57] propose a visualization-based approach to detect design anomalies for cases where the detection effort already includes the validation of candidates. However, these approaches need a lot of human intervention and expertise. Their results show that by using visualization, instead of directly using metrics, the anomaly detection process suffers from fewer variations between maintainers, but the detection results are the same.

2.1.2.3 Correction Opportunity-based Approaches

The authors in [73] introduce the concept of considering defect detection as an optimization problem; they use a combination of 12 metrics to measure the improvements achieved when methods are moved between classes. A fitness function (score) is computed by applying the sequence of transformations to the program at hand and by measuring the improvement in the metrics of interest [69]. Indeed, this search-based approach combines the detection and correction steps because an opportunity of refactoring is detected if a randomly selected correction improves the design quality. This is because the order of detected defects is related to the quality of improvements (difference in fitness). Furthermore, the problems mentioned before for metrics still apply for search-based techniques since they use a fitness function that consists of a combination of metrics.

Graph transformations can lead to an underlying theory of refactoring [107] where each refactoring corresponds to a graph production rule, and each refactoring application corresponds to a graph transformation. The theory of graph transformation can be used to reason about applying refactorings in parallel, using theoretical concepts such as confluence and critical pair analysis. These categories of approaches combine the identification of code to refactor and which refactorings to apply. In them, programs can be expressed as graphs, and refactorings correspond to graph production rules or graph transformations. Mens et al [64] use graph rewriting formalism to prove that refactorings preserve certain kinds of relationships (updates, accesses and invocations) that can be inferred statically from the source code. Bottoni et al [110] describe refactorings as coordinated graph transformation schemes in order to maintain consistency between a program and its design when any of them evolves by a refactoring. Heckel [102] uses graph transformations to formally prove the claim (and corresponding algorithm) of Roberts [27] that any set of refactoring post-conditions can be translated into an equivalent set of preconditions. Van Eetvelde and Janssens [116] propose a hierarchical graph transformation approach to be able to view and manipulate the software and its refactorings at different levels of detail.

2.2 Correctness of Model Transformation

Correctness of model transformations can be analyzed from different perspectives. Existing works can be classified into categories: formal verification-based approaches and testing approaches.

We start by describing existing work in the first category. Baleani et al. argue in [86] that correctness of model transformations for industrial tools should be based on formal models in order to ensure correctness by construction. For this purpose, they suggest to use a block diagram formalism called synchronous reactive model of computation. However, correct interpretation of the model transformation rules does not imply a correct result, one that is a model of the target language. Semantic correctness is discussed by Karsai et al. in [59], where specific behavior properties of the source model shall be reflected in the target model. In [42], semantic correctness is ensured by using the same rules for the model transformation, also for the transformation of the operational semantics, which is given by graph rules. By doing this, the behaviour of the source model can be compared with the one of the target model by checking mixed confluence. However, this paper concentrates on syntactical correctness based on the integrated language generated by the triple rules. [9],[118] are some works on using graph transformation rules to specify the dynamic behavior of systems. For example, [118] presents a meta-level analysis technique where the semantics of a modeling language are defined using graph transformation rules. A transition system is generated for each instance model, which can be verified using a model checker. Furthermore, [9] verifies if a transformation preserves certain dynamic consistency properties by model checking the source and target models for properties p and q, where property p in the source language is transformed into property q in the target language. This transformation requires validation by a human expert. Especially in the area of graph transformations some work has been conducted that uses Petri Nets to check formal properties of graph production rules. Thereby, the approach proposed in [105] translates individual graph rules into a Place/-Transition Net and checks for its termination. Another approach is described in [117], where the operational semantics of a visual language in the domain of production systems are described with graph transformations. Varrò presents in [10] a translation of graph transformation rules to transition systems, serving as the mathematical specification formalism of various model checkers to achieve the formal verification of model transformation. Thereby, only the dynamic parts of the graph transformation systems are transformed to TS in order to reduce the state space. In [40], a simple error taxonomy for model transformations is presented, which is then used to automatically generate test cases for model transformations. A very similar approach is presented by Darabos et al. in [10], focusing on common errors in graph transformation languages in general, and on errors in the graph pattern matching phase in particular. Both taxonomies are, however, rather general and only describe possible errors in graph transformation specifications.

After studying the existing work in formal verification-based approaches, we can conclude that one its important problem is that the results of a formal analysis can be invalidated by erroneous model transformations. In fact, the systems' engineers cannot distinguish whether an error is in the design or in the transformation. Furthermore, existing work requires a significant amount of mathematical expertise and user interaction (not fully automated). In addition, the existing work based on model checking and graph transformation does not combine the syntactic and semantic correctness of model transformations in one approach. For a syntactic correctness analysis, one has to decide whether the result of the transformation is a well-formed model of the target language. In case of semantic correctness analysis, we need to decide if the model transformation preserves (transformation specific) correctness properties.

Many works exist on model transformation testing [125],[32]. Fleurey et al. [34] and Steel et al. [14] discuss the reasons why testing model transformation is distinct from testing traditional implementations: the input data are models that are complex when compared to simple data types. Both papers describe how to generate test data in MDA by adapting existing techniques, including functional criteria and bacteriologic approaches [14]. Lin et al. [125] propose a testing framework for model transformation, built on their modeling tools and transformation engine, that offers a support tool for test case construction, test execution and test comparison; but the test models are manually developed in their work.

One of the widely-used techniques for test generation is mutation analysis. Mutation analysis is a testing technique that was designed to evaluate the efficiency of a test set. Mutation analysis consists of creating a set of faulty versions or mutants of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. Mottu et al. [113] have adapted this technique to evaluate the quality of test cases. They introduce some modifications in the transformation rules (program-mutant). Then using the same test cases as input an oracle function compares between the results (target models). If all results are the same, we can assume that the input cases were not sufficient to cover all the transformation possibilities (rules). Comparing to our work, our goal is not to evaluate the quality of a data set but to propose a generic oracle function to detect transformation errors. Our oracle function compares between some potential errors (detectors) and transformation traces to evaluate. However, in mutation analysis the oracle function compares between two target models, one generated by the original mechanism (rules) and another after modifying the rules. In addition, our technique does not create program variation (rules modifications) but traces variation that differs from good ones. Furthermore, the mutation analysis technique needs to define an expected model for each test case in order to compare it with another target model obtained from the same test case after modifying the rules (mutant).

Some other approaches are specific to test case generation for graph-transformation mechanism. Küster[58], addresses the problem of model transformation validation in a way that is very specific to graph transformation. He focuses on the validation of the rules that define the model transformation with respect to termination and confluence. His approach aims at ensuring that a graph transformation will always produce a unique result. Küster's work is an important step for model transformation validation but it does not aim at validating the functionality of a transformation (i.e., it does not aim at running a transformation to check if it produces a correct result). Darabos et al. [25] also investigate the testing of graph transformations. They consider graph transformation rules as the specification of the transformation and propose to generate test data from this specification. Their testing technique focuses on testing pattern matching activity that is considered the most critical of a graph transformation process. They propose several fault models that can occur when computing the pattern match as well as a test generation technique that targets those particular faults. However, the Darabos' approach is specific to test only graph transformation mechanisms. Sturmer et al. [22] propose a technique for generating test cases for code generators. The criterion they propose is based on the coverage of graph transformation rules. Their approach allows the generation of test cases for the coverage of both individual rules and rule interactions but it requires the code generator under test to be fully specified with graph transformation rules. Sampath et al. [11] propose a similar method for verification of model processing tools such as simulators and code-generators. They use a meta-model based test-case generation method that generates test-cases for model processors.

Mottu et al. [32] describes six different oracle functions to evaluate the correctness of an output model. These six functions can be classified in three general categories. For the first category, current MDE technologies and model repositories store and manipulate models as graphs of objects. Thus, when the expected output model is available, the oracle compares two graphs. In this case, the oracle definition problem has the same complexity as the graph isomorphism problem, which is NP-hard [6]. In particular, we can find a test case output and an expected model that look different (contain different model elements) but have the same meaning. So, the complexity of these data structures makes it difficult to provide an efficient and reliable tool for comparison [22]. Still, several studies have proposed simplified versions with a lower computation cost [12]. For example, Alanen et al. [4] present a theoretical framework for performing model differencing. However, they rely on the use of unique element identifiers for the model elements. To illustrate the specification conformance category, we present two contributions: design by contract and pattern matching [112]. For design by contract, the transformation is specified by pre- and post-conditions, and transformation invariants that must be satisfied. The constraints are defined at the meta-model level and expressed in OCL. For pattern matching, templates are used to specify the expected features of the input and output models with pre- and post-conditions for the transformation. The difference with design by contract approaches is that specific constraints must be defined for each output model. Both oracles are difficult to define. Indeed, the number of constraints to define can be very large to cover all rules and patterns [112]. This is especially the case of contracts related to one-to-many mappings. Moreover, being formal specifications, these constraints are difficult to write in practice. In pattern matching, the constraints are described at the model level and may lead to a fastidious task to define them for each model instance [112].

More general, when many test models are necessary, at least many test cases are created. To reduce the effort and the risk of making an error, it is necessary that each test case does not have its own oracle, but that an oracle is reused in different test cases. Such an oracle is generic and not dedicated to a test case, its test model, and its corresponding output model. Oracle functions using patterns or expected models are not adapted since they need the writing of at least one oracle data for each test case. Generic oracle data are preferable since they are written only once, and could be used with their corresponding oracle function in any test case. In addition, all these approaches to model transformation validation and testing consider a particular technique for model transformation and leverage the specificities of this technique to validate the transformation. This has the advantage of having validation techniques that are well-suited to the specific faults that can occur in each of these techniques. The results of these approaches are difficult to adapt to other transformation techniques (that are not rule-based).

2.3 By-Example Software Engineering

Examples play a key role in the human learning process. There exist numerous theories on learning styles in which examples are used. For a description of today's popular learning style theories, see [95],[7].

Our work is based on using past transformation examples. Various "by-example" approaches have been proposed in the software engineering literature.

What does by-example really mean? What do all by-example approaches have in common? The main idea, as the name already suggests, is to give the software examples of how things are done or what the user expects, and let it do the work automatically. In fact this idea is closely related to fields such as machine learning or speech recognition. Common to all by-example approaches is the strong emphasis on user friendliness and a "short" learning curve. According to [20] the by-example paradigm dates back to 1970 - see "Learning Structure Descriptions from Examples" in [90].

Programming by example [95] is the best known by-example approach. It is a technique for teaching the computer new behavior by demonstrating actions on concrete examples. The system records user actions and generalizes a program that can be used for new examples. The generalization process is mainly based on user responses to queries about user intentions. Another well-known approach is Query by Example (QBE) [7]. It is a query language for relational databases constructed from sample tables filled with example rows and constraints. QBE is especially suited for queries that are not too complex and can be expressed in terms of a few tables. In web-engineering, Lechners et al [62] present the language TBE (XML transformers by example) that allows defining transformers for WebML schemes by example, i.e., stating what is desired instead of specifying the operations to get it. Advanced XSLT tools are also capable of generating XSLT scripts using examples from schema level (like MapForce from Altova) or document (instance) level mappings (such as the pioneering XSLerator from IBM Alphaworks, or the more recent StylisStudio).

The problems addressed by the above-mentioned approaches are different from ours in both the nature and the objectives.

2.4 Search-based Software Engineering

Our approach is largely inspired by contributions in Search-Based Software Engineering (SBSE). SBSE is defined as the application of search-based approaches to solving optimization problems in software engineering [72]. Once a software engineering task is framed as a search problem, there are numerous approaches that can be applied to solving that problem, from local searches such as exhaustive search and hill-climbing to meta-heuristic searches such as genetic algorithms (GAs) and ant colony optimisation [70]. Many contributions have been proposed for various problems, mainly in cost estimation, testing, and maintenance [101] ,[72]. Module clustering, for example, has been addressed using exhaustive search [70], genetic algorithms [72] and simulated annealing (SA)[103]. In those studies that compared search techniques, hill-climbing was perhaps surprisingly found to produce better results than meta-heuristic GA searches. Model verification has also been addressed using search-based techniques. Shousha et al. [70] propose an approach to detect deadlocks in UML models, but the generation of a new quality predictive model starting from a set of existing ones by using simulated annealing (SA) that is reported in [103] is probably the problem that is the most similar to MT by examples. In that work, the model is also decomposed into fine-grained pieces of expertise that can be combined and adapted to generate a better prediction model. To the best of our knowledge, inspired among others by the road map paper of Harman [72], the idea of treating model transformation as a combinatorial optimization problem to be solved by a search-based approach was not studied before our proposal.

2.5 Summary

This chapter has introduced the existing work in different domains related to our work. The closest work to our proposal is model transformation by example (MTBE). The commonalities of the by-example approaches for model transformation can be summarized as follows: All approaches define an example as a triple consisting of an input model and its equivalent output model, and traces between the input and output model elements. These examples have to be established by the user, preferably in concrete syntax. Then, generalization techniques such as hard-coded reasoning rules, inductive logic [23], or relational concept analysis or pattern are used to derive model transformation rules from the examples, in a deterministic way that is applicable for all possible input models which have a high similarity with the predefined examples. One conclusion to be drawn from studying the existing by-example approaches is that they use semi-automated rules generation, with the generated rules further refined by the user. In practice, this may be a lengthy process and require a large number of transformation examples to assure the quality of the inferred rules. In this context, the use of search-based optimization techniques can be a more preferable transformation approach since it directly generates the target model from the existing examples, without using the rules step. This also leads to a higher degree of automation than exiting by-example approaches. Table 1 summarizes existing transformation by-example approaches according to given criteria. The majority of these approaches are specific to exogenous transformation and based on the use of traceability.

By-example approaches	**Exogenous transformation**	**Endogenous transformation**	**Traceability**	**Rules generation**
Varrò et al. [23]	X		X	X
Wimmer et al.[65]	X		X	X
Sun et al. [124]		X	X	
Dolques et al.[123]	X		X	X
Langler et al.[97]	X		X	X
Brosch et al. [96]		X	X	

Table 2 By-example Approaches

As shown in the search-based section, like many other domains of software engineering, MDE is concerned with finding exact solutions to these problems, or those that fall within a specified acceptance margin. Search-based optimization techniques are well-suited for the purpose. For example, when testing model transformations, the use of deterministic techniques can be unfeasible due to the number of possibilities to explore for test case generation, in order to cover all source meta-model elements. However, the complex nature of MDE problems sometimes requires the definition of complex fitness functions [73]. Furthermore, the definition is specific to the problem to solve and necessitate expertise in both search-based and MDE fields. It is thus desirable to define a generic fitness function, evaluating a quality of a solution that can be applied to various MDE problems with low adaptation effort and expertise.

To tackle these challenges, our contribution combines search-based and by-example techniques. The difference with case-based reasoning approaches is that many sub-cases can be combined to derive a solution, not just the most adequate case. In addition, if a large number of combinations have to be investigated, the use of search-based techniques becomes beneficial in terms of search speed to find the best combination. In the next chapters, we detail our contribution based on this combination between by-example and search-based techniques.

Part 1: Exogenous Transformation by Example

The first part of this thesis presents our solution for the problem of automating exogenous transformation based on the use of examples. Most of the available work on model transformation is based on the hypothesis that transformation rules exist and that the important issue is how to express them. However, in real problems, the rules may be difficult to define as is often the case when the source and/or target formalisms are not widely used or proprietary. Indeed, as for any rule-based system, defining the set of rules is not an obvious task and many difficulties accompany the results [24].

As a solution, we described MOTOE (Model Transformation as Optimization by Example), a novel approach to automate model transformation (MT) using heuristic search. MOTOE uses a set of transformation examples to derive a target model from a source model. The transformation is seen as an optimization problem where different transformation possibilities are evaluated and, for each possibility, a quality is associated depending on its conformance with the examples at hand. The search space is explored with two methods. In the first one, we use PSO (Particle Swarm Optimization) with transformation solutions generated from the examples at hand as particles. Particles progressively converge toward a good solution by exchanging and adapting individual construct transformation possibilities. In the second method, a partial run of PSO is performed to derive an initial solution. This solution is then refined using a local search with SA (Simulated Annealing). The refinement explores neighboring solutions obtained by trying individual construct transformation possibilities derived from the example base. In both methods, the quality of a solution considers the adequacy of construct transformations as well as their mutual consistency.

We distinguish two types of models to transform: dynamic and static. The dynamic model is used to express and model the behaviour of a problem domain or system over time, whereas the static model shows those aspects that do not change over time. UML static models are mainly expressed using a class diagram that shows a collection of classes and their interrelationships, for example generalization/specialization and association.

The transformation of dynamic models is more difficult than static ones. It may be not obvious to realize, due to two main reasons [29]. First, defining transformation rules, for dynamic models, can be difficult since the source and target languages have constructs with different semantics; therefore, 1-to-1 mappings are not sufficient to express the semantic equivalence between constructs. Second, in addition to ensuring structural (static) coherence, it should guarantee behavioral coherence in terms of time constraints and weak sequencing.

We evaluate our by example-approach to the two kind of models. For static models, we consider class diagram to relational schema transformation; for dynamic models, we adapt our approaach to sequence diagram to colored Petri nets transformation. We detail these two contributions in the next two chapters.

Chapter 3: Static Model Transformation by Example

3.1 Introduction

In this chapter, we describe our solution for the problem of automating static model transformation using examples. This contribution has been accepted for publication in the Journal of System and Software Modeling (SOSYM) [79]. The paper, entitled "Search-based Model Transformation by Example", is presented next.

3.2 Class Diagram to Relational Schema Transformation by Example

Search-based Model Transformation by Example

MAROUANE KESSENTINI[1], HOUARI SAHRAOUI[1], MOUNIR BOUKADOUM[2] AND OMAR BEN OMAR[1]

Abstract Model transformation (MT) has become an important concern in software engineering. In addition to its role model driven development, it is useful in many other situations such as measurement, refactoring, and test-case generation. Roughly speaking, MT aims to derive a target model from a source model by following some rules or principles. So far, the contributions in MT have mostly relied on defining languages to express transformation rules. However, the task of defining, expressing, and maintaining these rules can be difficult, especially for some formalisms. In other situations, companies have accumulated examples from past experiences. Our work starts from these observations to view the transformation problem as one to solve with fragmentary knowledge, *i.e.* with only examples of source-to-target model transformations. Our proposal has two main advantages: 1) for any source model, it always proposes a transformation, even when rule induction is impossible or difficult to achieve. 2) it is independent from source and target formalisms; aside from the examples, no extra information is needed. In this context, we propose an optimization-based approach that consists of finding in the examples combinations of transformation fragments that best cover the source model. To this end, we use two strategies based on two search-based algorithms: Particle Swarm Optimization (PSO) and Simulated Annealing (SA). The results of validating our approach on industrial projects show that the obtained models are accurate.

Keywords *Search-based software engineering, Automated model transformation, Transformation by example.*

1 Introduction

In the context of model driven development (MDD) [37], the creation of models and model transformations is a central task that requires a mature development environment, based on the best practices of software engineering principles. For a comprehensive approach to MDD, models and model transformations must be designed, analyzed, synthesized, tested, maintained and subjected to configuration management to ensure their quality. This makes model transformation a central concern in the MDD paradigm: not used only in forward engineering, it allows concentrating the maintenance effort on models and using transformation mechanisms to generate code. As a result, many transformation languages are emerging.

Practical model-to-model transformation languages are of prime importance. Despite the many approaches [33,36,20] that addressed the request for proposals of OMG QVT RFP[34,35], the MT problem has no universal solution because the majority of exisiting approaches are dependent to the source and target metamodels. A popular view attributes the situation to the difficulty of defining or expressing transformation rules, especially for proprietary or non-widely used formalisms. Indeed, most contributions in MT are concerned with defining languages to express transformation rules. Transformation rules can be implemented using: (1) general programming languages such as Java or C#; (2) graph transformation languages like AGG [19] and VIATRA [14]; (3) specific languages such as ATL [18] and QVT [35]. Sometimes, transformations are based on invariants: pre-conditions and post-conditions specified in languages such as OCL [43]. These approaches have been successfully applied to transformation problems where there exists knowledge about the mapping between the source and target models. Still, there exist situations where defining the set of rules is a complex task and many difficulties accompany the results [45] (incompleteness, redundancy, inconsistency, etc.). In particular, the experts may find it difficult to master both the source and target meta-models [15].

On the other hand, it is recognized that experts can more easily give transformation examples than complete and consistent transformations rules [2]. This is particularly true for industrial organizations where a memory of past transformation examples can be found, and it is the main motivation for transformation-by-examples approaches such as the one proposed in [13]. The principle of this approach is to semi-automatically derive transformation rules from an initial set of examples (interrelated source and target models), using inductive logic programming (ILP). However, it is not adaptable to new situations where no examples are available.

We can alternatively view MT as an optimization problem where a (partial) target model is to be automatically derived from available examples. In this context, we recently introduced an optimization-based approach to automate MT called MOTOE (model transformation as optimization by examples) [29]. MOTOE views MT as essentially a combinatorial optimization problem where the transformation of a source model is obtained by finding, for each of its constructs, a similar transformation in an example base. Due to the large number of possible combinations, a heuristic-search strategy is used to build the transformation solution as a set of individual construct transformations. Comparing to our pervious paper [29], we extend MOTOE with a more sophisticated transformation building process and use a larger scale validation with industrial data. In particular, we compare two strategies: (1) parallel exploration of different transformation possibilities (call it population-based MT) by means of a global search heuristic implemented with PSO (*Particle Swarm Optimization*) [22], and (2) initial transformation possibility improvements (call it adaptation-based MT) implemented with a hybrid heuristic search that combines PSO with the local search heuristic SA (*Simulated Annealing*) [17].

The approach we propose has the advantage over rule-based algorithms that, for any source model, it always proposes a transformation, even when rule induction is impossible or difficult to achieve. Although, it can be seen as a form of case-based reasoning (CBR) [8], it actually differs from CBR approaches in that all the existing models are used to derive a solution, not only the most similar one. Another interesting advantage is that our approach is independent from source and target formalisms; aside from the examples, no extra information is needed. In conclusion, our approach is not meant to replace rule-based approaches; instead, it applies to situations where rules are not available, difficult to define, or non-consensual.

In this paper, we illustrate and evaluate our approach on the well-known case of transforming UML class diagrams (CLD) to relational schemas (RS). As will be shown in Section 4, the models obtained using our transformation approach are comparable to those derived by transformation rules. Although transformation rules exist in this case, our choice of CLD-to-RS transformation is motivated by the fact that it is well-known and reasonably complex; this allows us to focus on describing the technical aspects of out approach and comparing its results with a well-known alternative. However, our approach can also be applied to more complex transformations such as sequence diagrams-to-colored Petri-nets [47].

The remainder of this paper is structured as follows. Section 2 is dedicated to the MT-problem statement. In Section 3, we describe the principles of our approach. The details are discussed in Section 4; they include the adaptation of two search algorithms for the MT problem. Section 5 contains the validation results of our approach with industrial projects and a comparison between the global- and adaptation-based strategies. In Section 6, the related work in model transformation is discussed. We conclude and suggest research directions in Section 7.

2 Approach Overview

This section shows how, under some circumstances, MT can be seen as an optimization problem. We also show why the size of the corresponding search space makes heuristic search necessary to explore it. Finally, we give the principles of our approach.

2.1 Problem Statement

Defining transformations for domain-specific or complex languages requires proficiency in high programming languages, knowledge of the underlying metamodels, and knowledge of the semantic equivalency between the meta-models concepts [37]. Therefore, creating MT rules may become a complex task [30]. On the other hand, it is often easier for experts to show transformation examples than to express complete and consistent transformation rules [15]. This observation has led to a new research direction: model transformation by example (MTBE), where, like in [13], rules are semi-automatically derived from examples.

In the absence of rules or an exhaustive set of examples that allows rule extraction, an alternative solution is to derive a partial target model from the available examples. The generation of such models consists of finding, in the examples, some model fragments that best match the model to transform. To characterize the problem, we start with some definitions.

***Definition* 3.1 *(Model to Transform)*.** A model to transform, *M,* is a model composed of *n* constructs expressed in a predefined abstract syntax.

Definition 3.2 (Model Construct). A construct is a source or target model element.

For example, a class in a CLD. It may contain properties that describe it, e.g. its name. Complex constructs may contain sub-constructs; for example, a class could have attributes. For graph-based formalisms, constructs are typically nodes and links between nodes. For instance, classes, associations, and generalizations are model constructs in UML class diagrams.

Definition 3.3 (Block). A block defines a previously performed transformation trace between a set of constructs in the source model and a set of constructs in the target model. Constructs that should be transformed together are grouped within the same block.

For example, a generalization link g between two classes A and B cannot be mapped independently from the mapping of A and B. In our case, we assume that blocks are manually defined by domain experts when transforming models. Finally, blocks are not general rules since they involve concept instances (e.g., class Student) instead of just concepts (e.g., class concept). In other words, where transformation rules are expressed in terms of metamodels, blocks are expressed in terms of concrete models.

Definition* 3.4 *(Transformation Example). A transformation example, TE, is a mapping of constructs from a source model to the corresponding target model. Formally, we view a TE as a triple <*SMD, TMD, MB*>, where SMD denotes the source model, TMD denotes the target model, and MB is a set of mapping blocks that relate sets of constructs in SMD to their equivalents in TMD.

For example, the creation of a database schema from a UML class diagram describing student records is a transformation example. ***The Base of examples*** is a set of transformation examples.

Our goal is to combine and adapt transformation blocks - which are fragments coming from one or more model transformations in the base of examples - to generate a new transformed model by similarity. A fragment from an example model is considered as similar to one from the source model if it shares the same construct types with similar properties. For instance, in a class diagram, a fragment with an association pays between two classes Client and Bill is similar to a fragment from another diagram containing an association evaluates relating classes ControlExam and Module. The degree of similarity depends on the properties of the classes and associations (attributes types, cardinalities, etc.). In the absence of transformation rules, any combination of blocks is a transformation possibility. For example, when transforming a class diagram into a database schema, any class can be translated into a table, a foreign key in an existing table, two tables, or any other possible combination of target constructs. However, with transformation examples, possibilities are reduced to transformations of similar constructs in these examples.

The transformation of a model M with n constructs, using a set of examples that globally define m possibilities (blocks), consists of finding the subset from the m possibilities that best transforms each of the n constructs of M. "Best transforms" means that each construct can be transformed by one of the selected possibilities and that construct transformations are mutually consistent. In this context, m^n possible combinations have to be explored. This number can quickly become huge. For example, an average UML class diagram with 40 classes and 60 links (generalization, associations, and aggregations) defines 100 constructs (40 + 60). At the same time, an example base with a reasonable number of examples may contain hundreds of blocks, say 300. In this case, 300100 possible combinations should be explored. If we limit the possibilities for each construct to only blocks that contain similar constructs, the number of possibilities becomes $m_1 \times m_2 \times m_3 \times \ldots \times m_n$ where each $m_i \leq m$ represents the number of transformation possibilities for construct *i*. Although the number of possibilities is reduced, it could still be very large for large CLDs. In the same example, assuming that each of the 100 constructs has 8 or more mapping possibilities leads to exploring at least 8^{100} combinations. Considering these magnitudes, exploring all the possibilities cannot be done within a reasonable time frame. This calls for alternative approaches such as heuristic search.

2.2 Approach Overview

We propose an approach that uses knowledge from previously solved transformation cases (examples) so that a new MT problem is solved using a combination of the past problem solutions, and the (partial) target model is automatically derived by an optimization process that exploits the available examples.

Figure 1 shows the general structure of MOTOE. The approach takes as inputs a base of examples (i.e., a set of transformation examples) and a source model to transform, and generates as output a target model. The generation process can be viewed as the selection of the subset of transformation fragments (blocks) in the example base that best matches the constructs of the source model (using a similarity function). In other words the transformation is done as an assembly of building blocks. The quality of the produced target model is measured by the conformance of the selected fragments to structural constraints, i.e., by answering the following two questions: (1) did we choose the right blocks? and (2) did they fit together?

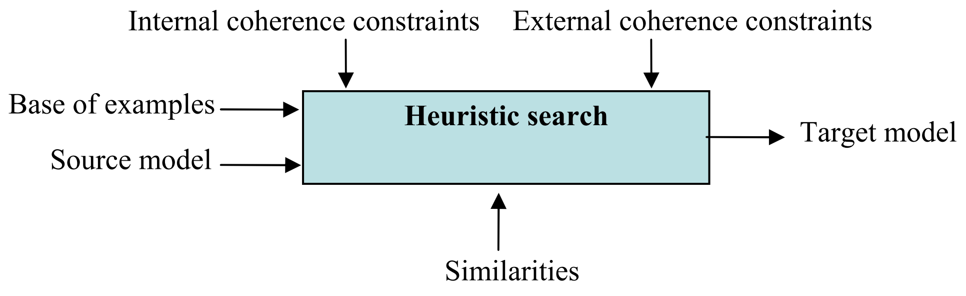


Fig 1. MOTOE overview

Figure 2 illustrates the case of a source model with 6 constructs to transform represented by dots. A transformation solution consists of assigning to each construct ci a mapping block, i.e. a transformation possibility from the example base (blocks are represented by rectangles in Figure 2). A possibility is considered to be adequate if the assigned block contains a construct similar to ci (similarity evaluation is discussed in Section 3.3).

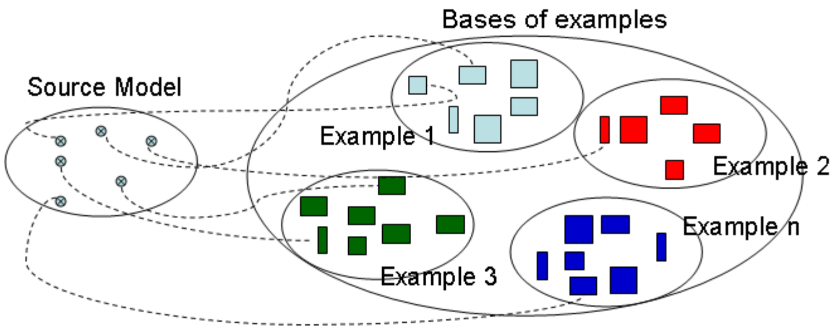


Fig 2. Illustration of the proposed transformation process

As many block assembly schemes are possible, the transformation is a combinatorial optimization problem. In fact, the number of possible solutions becomes very high. Thus, a deterministic search is unfeasible and a heuristic search is needed to find an acceptable solution. The dimensions of the solution space are the constructs of the source model to transform. A solution is determined by the assignment of a transformation fragment (block) to each source model construct. The search is guided by the quality of the solution according to internal coherence (inside a block), and external coherence (between blocks).

To explore the solution space, we study two different search strategies in this work. The first one uses a global heuristic search by means of the PSO algorithm [22]. The second one first uses a global search to reduce the search space and find a first transformation solution; then it uses a local heuristic search, using SA algorithm [17], to refine the first solution.

To illustrate our example-based transformation mechanism, consider the case of model transformation between UML class diagrams (CLD) and relational schemas (RS). Figure 3 shows a simplified metamodel of the UML class diagram, containing concepts like class, attribute, relationship between classes, etc. Figure 4 shows a partial view of the relational schema metamodel, composed of table, column, attribute, etc. The transformation mechanism, based on rules, will then specify how the persistent classes, their attributes and their associations should be transformed into tables, columns and keys.

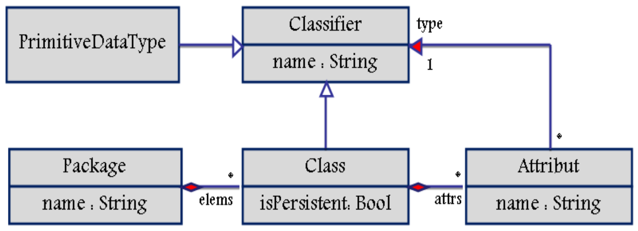


Fig 3. Class diagram metamodel

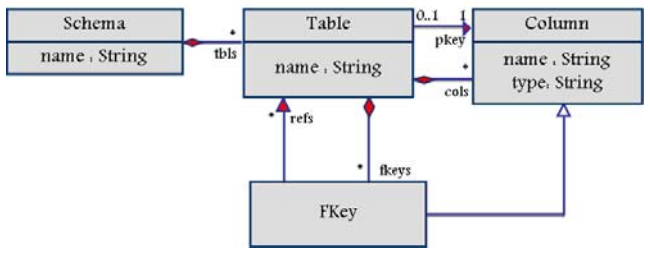


Fig 4. Relational schema metamodel

The choice of this particular example is only motivated by considerations of clarity. As MOTOE is independent from the nature of the transformation problem because it does not depend from the source and target metamodels, it is applicable to any kind of formalisms where prior examples of successful transformation are available.

A transformation example of a CLD to a RS is presented in Figures 5 and 6. The CLD is the source model (a) and the RS is the target one (b). The CLD contains 12 constructs that represent 7 classes (including 2 association classes), 3 associations, and 2 generalization links. The five non-associative classes are mapped to tables with the class attributes mapped to columns of the tables. The associations between *Student* and *Module*, and between *Teacher* and *Module*, are respectively translated into tables *Register* and *Intervene* with, as columns, the attributes of the associative classes. Each of these tables also contains two foreign keys to their related tables. Association *evaluate* becomes a foreign key in table *ControlExam*. Finally, the generalization links are mapped as foreign keys in the tables corresponding to the subclasses.

The decisions made in this transformation example are not unique alternatives. For instance, we can find many rules (point of views) to transform a generalization link. One of them maps abstract class *Person* as a duplication of its attributes in the tables that correspond to classes *Student* and *Teacher*.

Following Definition 3.4 of Section 2.1, *SMD* corresponds to the CLD, *TMD* represents the corresponding RS and *MB* is the set of mapping blocks between the two models. For example, a block describes the mapping of the association *evaluate* and classes *Module* and *ControlExam* in Figure 5. This block respectively assigns tables *Module* and *ControlExam* to the two classes, and foreign key *IDModule* to the association (Figure 5). As mentioned earlier, the transformations of the three constructs are grouped within the same block since they are interdependent.

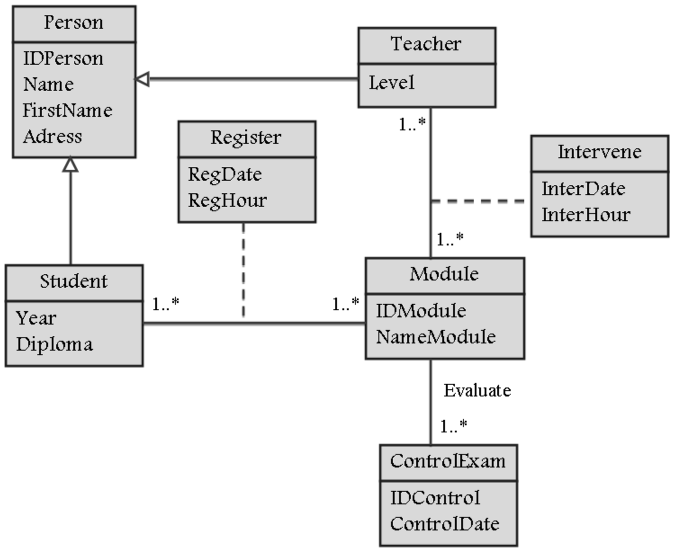


Fig 5. Example of a CLD source model

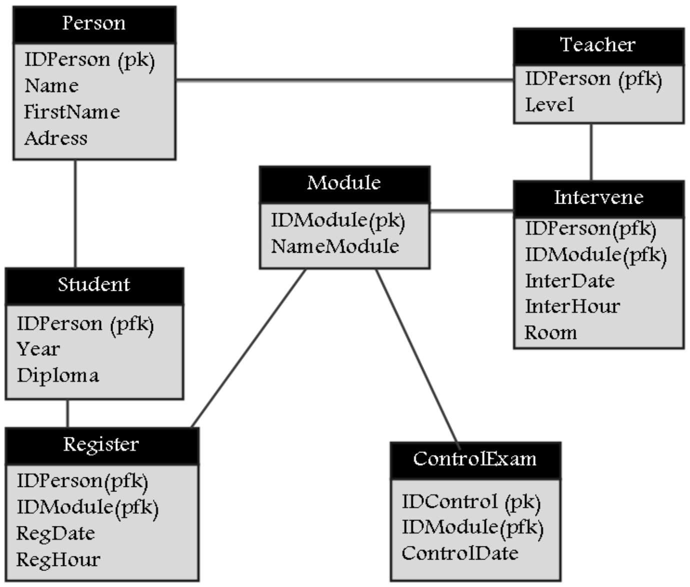


Fig 6. Equivalent RS target model to the CLD source model of Figure 5

To ease the manipulation of the source and target models and their transformation, the models are described using a set of predicates that correspond to the included constructs. Each construct is represented by one or more predicates. For example, Class *Teacher* in Figure 5 is described as follows:

```
Class(Teacher).
Attribute(Level, Teacher,_).
```

The first predicate indicates that *Teacher* is a class. The second states that *Level* is an attribute of that class and that its value is not unique ("_" instead of "unique").

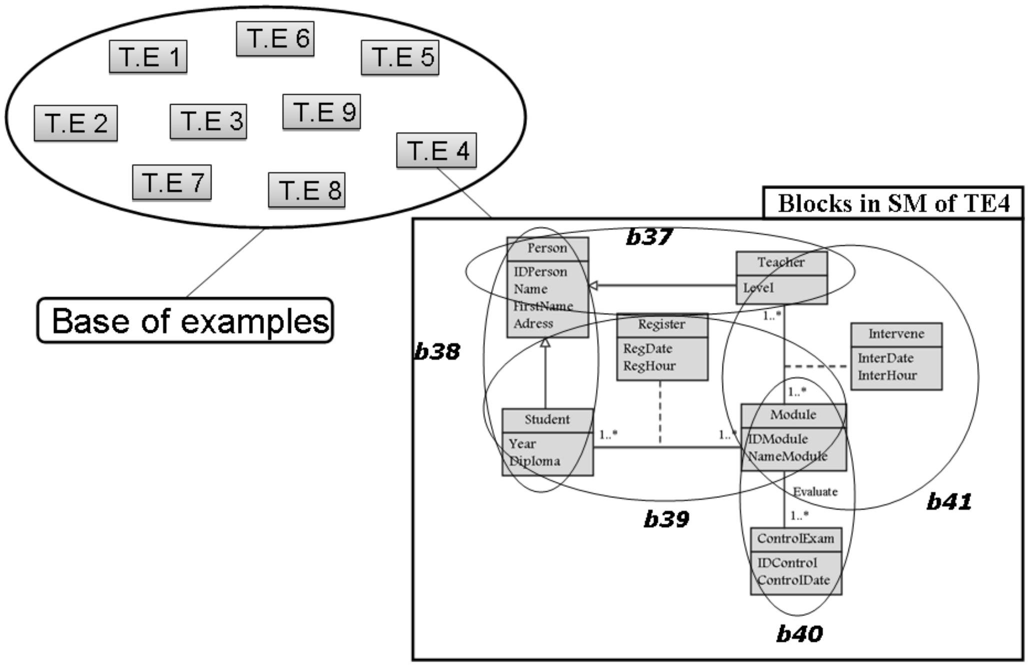


Fig 7. Base of transformation examples and blocks generation in source model of TE4

The mapping blocks relate the predicates in the source model to their equivalent constructs in the target model. In Figure 7, for instance, block B37[1] which contains the generalization link and the two classes *Teacher* and *Person* is described as follows:

```
Begin b37
Class(Person) : Table(Person).
Attribute(IDPerson, Person, unique) : Column(IDPerson,
Person,pk).
Attribute(Name, Person,_) : Column(Name, Person,_).
Attribute(FirstName, Person,_) : Column(FirstName, Person,_).
Attribute(Address, Person,_) : Column(Address, Person,_).
Class(Teacher) : Table(Teacher).
Attribute(Level, Teacher,_) : Column(Level, Teacher,_).
Generalization(Person, Teacher) : Column(IDPerson, Teacher,
pfk).
End b37
```

Mappings are expressed with the ':' character. So, the mapping between predicates `Attribute(IDPerson, Person, unique)` and `Column(IDPerson, Person, pk)` indicates that the "unique" attribute *IDPerson* in class *Person* is transformed into the column *IDPerson* in table *Person* with the status of primary key. Similarly, the mapping between `Generalization(Person, Teacher)` and `Column(IDPerson, Teacher, pfk)` indicates that the generalization link is represented by the primary-foreign key (pfk) *IDPerson* in table *Teacher*.

[1] For ease of traceability, blocks are sequentially numbered, starting from the first transformation example in the example base. For instance, the 9 blocks of example TE1 are labeled B_1 to B_9. The 13 blocks of TE2 B_{10} to B_{22}, and so on. When a solution is produced, it is relatively easy to determine which examples contributed to it.

A model M_i to transform is characterized only by its description SMD_i, *i.e.* a set of constructs expressed by predicates. A construct can be transformed in many ways, each having a degree of relevance. This depends on three factors: (1) the adequacy of the individual construct transformations; (2) the internal consistency of the individual transformations inside the blocks; (3) the transformation (external) coherence between the related blocks, *i.e.*, blocks sharing the same constructs. For example, consider a model to transform that has two classes, *Dog* and *Animal*, related by a generalization link *g*. *g* could become a table, many tables, a column, a foreign key, or any other possibility. A possibility is considered adequate if there exists a block in the example base that maps a generalization link. For instance, the mapping of block B37 (Figure 7(b)) is adequate because it also involves a generalization link. It is also internally consistent since it maps a similar pair of classes. Finally, it is externally coherent if *Dog* and *Animal* are only mapped to tables in the other blocks that contain them.

The transformation quality of a source model is the sum of the transformation qualities of its constructs. Consequently, finding a good transformation is equivalent to finding the combination of construct transformations that maximizes the global quality. But since the number of combinations may be very large because of multiple mapping possibilities, it may become difficult, if not impossible, to evaluate them exhaustively. As stated previously, heuristic search offers a good alternative in this case. The search space dimensions are the constructs and the possible coordinates in these dimensions are the block numbers. A solution then consists of choosing a block number for each construct. The exploration of the search space using heuristic algorithms is presented next.

3 Transformation using Search-Based Methods

We describe in this section the adaptation of PSO and SA to automate MT. To apply them to a specific problem, one must specify the encoding of solutions, the operators that allow movement in the search space so that new solutions are obtained, and the fitness function to evaluate a solution's quality. These three elements are detailed in subsections

3.1, 3.2, and 3.3, respectively. Their use by PSO and SA to solve the MT problem is presented in subsections 3.4 and 3.5.

3.1 Representing Transformation Solutions

One key issue when applying a search-based technique is finding a suitable mapping between the problem to solve and the techniques to use, *i.e.*, in our case, encoding a transformation between a source and a target model. As stated in Section 2, we view the set of potential solutions as points in a *n*-dimensional space where each dimension corresponds to one of the *n* constructs of the model to transform. Each construct could be mapped according to a finite set of blocks, which means that each dimension could take set of discrete values $b = \{i \mid 1 \le i \le m\}$, where m is the number of blocks extracted from transformation examples. For instance, the transformation of the model shown in Figure 8 will generate a 7-dimensional space that accounts for the four classes and the 3 relationships.

To define a particular solution, we associate with each dimension (construct) a block number that contains a transformation possibility. Each block number defines a coordinate in the corresponding dimension, and the resulting *n*-tuple of block numbers then defines a vector position in the *n*-dimensional space. For instance, the solution shown in Table 1 suggests that *construct1* (class *Command*) be transformed according to block28, construct2 (class *Bill*) according to block3, etc. Thus concretely, a solution is implemented as a vector where the constructs of the model to transform are the elements and the block numbers that refer to transformation possibility from the example base are the element values.

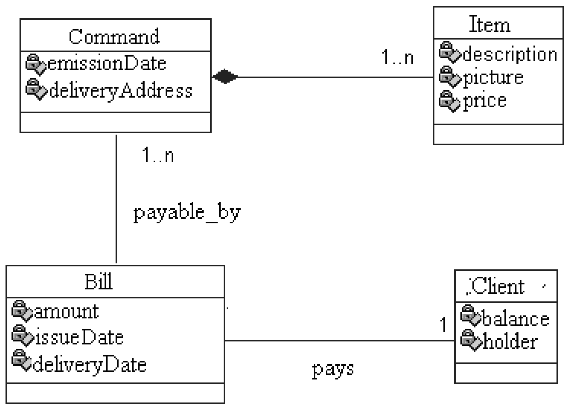


Fig 8. Example of source model (UML-class diagram)

Dimension	Construct	Block number
1	Class(Command)	28
2	Class(Bill)	3
3	Class(Article)	21
4	Class(Seller)	13
5	Aggregation	9
6	Association(payable_by)	42
7	Association(pays)	5

Table 1. Solution Representation

The proposed coding is valid for both heuristics. In the case of PSO, as an initial population, we create *k* solution vectors with a random assignment of blocks. Alternatively, SA starts from a solution vector produced by PSO.

3.2 Deriving A Transformation Solution

A change operator is a modification brought to a solution in order to produce a new one. In our case, it is the modification of a transformation of the source model in order to produce a new one. This is done by changing the blocks for some constructs, which is equivalent to changing the coordinates of the solution in search space. Unlike solution encoding, change operators are implemented differently by the PSO and SA heuristics. PSO changes blocks as a result of movement in the search space driven by a velocity function; SA performs the change randomly.

In the case of PSO, a translation (velocity) vector is regularly updated and added to a position vector to define new solutions (see Section 3.4, Equations 3 and 4 for details). For example, the solution sown in Table 1 may lead to the new solution shown at the bottom of

Figure 9. The velocity vector *V* assigns a real-valued translation for each element of the position vector. After adding the two vectors, the elements of the result are each rounded to the nearest integer to represent block numbers (The allowable values are bound by 1 and the maximum number of available blocks). As shown in Figure 9, the new solution updates the block numbers of all construct. Thus, block 42 replace block 19, block 7 remains here, block 49 replaces block 51, etc.

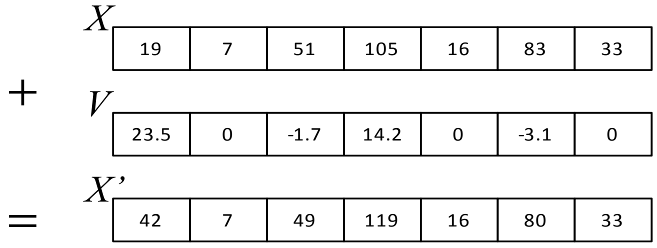


Fig 9. Change Operator in PSO

For SA, the change operator involves randomly choosing *l* dimensions ($l < n$) and replacing their assigned blocks by randomly selected ones from the example base. For instance, Figure 10 shows a new solution derived from the one of Table 1. Constructs 1, 5 and 6 are selected for change. They are assigned respectively blocks 52, 24, and 11 in place of 19, 16, and 83. The other constructs keep their transformation blocks. The number of blocks to change is a parameter of the SA algorithm (three in this example).

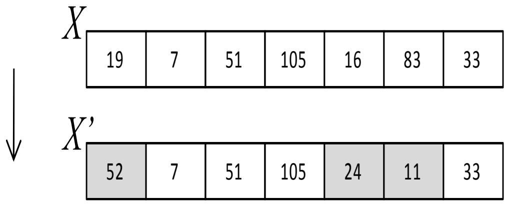


Fig. 10. Change Operator in SA

In summary, regardless of the search heuristic, a change consists of assigning new block numbers to one or more dimensions. Said otherwise, it drives new transformation solution X_{i+1} drived from the previous one X_{i+1}.

3.3 Evaluating Transformation Solutions

The *fitness function* quantifies the quality of a transformation solution, which basically is a 1-to-1 assignment of blocks from the example base to the constructs of the source model. As discussed in Section 2, the fitness function must consider the three following aspects for a construct *j* to transform:

- Adequacy of the assigned block to the construct *j* (a_j).
- Internal coherence of the individual construct transformation (ic_j) .
- External coherence with the other construct transformations (ec_j).

In this context, we define the fitness function of a solution as the sum of qualities of the *n* individual construct transformations. Formally,

$$f = \sum_{j=1}^{n} a_j \times (ic_j + ec_j) \qquad (1)$$

In this equation, a_j represents the adequacy factor with value 1 if the associated block contains a construct containing at least one construct of the same type as the j^{th} construct, and value 0 otherwise. This factor basically penalizes the assignment of blocks that do not contain constructs of the same type as the construct to transform (by giving them a zero value). This is a way to reduce the search space.

The internal-coherence factor ic_j measures the similarity, in terms of properties, between the construct to transform and the construct in the assigned block that has the same type. As shown in Section 3.1, the properties of the constructs are represented by the parameters of the predicates. Formally:

$$ic_j = \frac{\text{number of matched parameters in the predicates of the } \mathrm{j}^{\text{th}} \textit{construct}}{\text{total number of parameters in the predicates of the } \mathrm{j}^{\text{th}} \text{ constrcut}}$$

In general, a block assigned to a construct *j* contains more constructs than the one that is adequate with *j*. The external-coherence factor ec_j evaluates to which extent these constructs match the constructs that are linked to *j* in the source model . ec_j is defined as

$$ec_j = \frac{\text{number of matched constructs related to the } \mathrm{j}^{\text{th}} \text{ construct}}{\text{total number of constructs related to } \mathrm{j}^{\text{th}} \text{ construct}}$$

To illustrate the fitness calculation, consider again the example of Figure 8. The association *payable_by* (6th dimension) is defined by the predicate

```
Association (1,n,1,1,_,Command, Bill)
```

where the first four parameters indicates the multiplicities (1..n and 1..1), the fifth the name of the associative class if it exists, and the two last the source and target classes (*Command* and *Bill*). Consider a solution s_1 that assigns block 42 to this association:

```
Begin b42
Class(Client) : Table(Client).
Attribute(NClient, Client, unique) : Column(NClient, Client,
pk).
Attribute(ClientName, Client, ) : Column(ClientName, Client,
).
Attribute(Address, Client,_) : Column(Address, Client,_).
Attribute(Tel, Client,_) : Column(Tel, Client,_).
Class(Reservation) : Table(Reservation).
Attribute(NReservation, Reservation, unique) :
Column(NReservation, Reservation, pk).
Attribute(StartDate, Reservation,_) : Column(StartDate,
Reservation,_).
Attribute(EndDate, Reservation,_) : Column(EndDate,
Reservation,_).
Attribute(Region, Reservation,_) : Column(Region,
Reservation,_).
Association (1,n,0,n,_, Client, Reservation) :
Column(N_Client, reservation, fk).
End b42
```

In this case, a_6 (adequacy for the 6th construct) is equal to 1 because block 42 contains a predicate *Association* that relates classes *Client* and *Reservation*. This association predicate has five parameters over seven that match the ones of *pays* (1,n,x,x,_, origin and destination class names). As a result, we have ic_6=5/7=0.71. Moreover, according to block 42, to be consistent with the transformation of *payable_by,* classes *Bill* and *Command* have to be mapped to tables. On the other hand, s_1 also assigns blocks 28 and 3 to classes *Bill* (dimension 2) and *Client* (dimension 4), respectively. These two blocks are defined as follows.

```
Begin b28
Class(Position) : Table(Position).
….
Class(Employee) : Table(Employee).
…
Association(0,1, ,n,_, Position, Employee) :
Column(IDPosition, Employee, fk).
End b28

Begin b3
Class(Manager) : Table(Manager).
…
Class(Employee) : Table(Employee).
…
Generalization(Employee, Manager) : Column(IDEmployee,
Manager, fk).
End b3
```

In both blocks, classes are transformed into tables. Since this does not conflict with block 42, we have for the two related constructs c_6=2/2=1.

The fitness function also evaluates the completeness of a transformation indirectly. A solution that does not transform a subset of constructs will be penalized. Those constructs will have null values (a_j being always equal to 0). Finally, to make the values comparable across models with different numbers of constructs, a normalized version of the fitness function is used. For a particular construct, the fitness varies between 0 and 2 (ic_j and ec_j can be both equal to 1). Considering the n constructs, we normalized the fitness function as follows:

$$f_{nor} = \frac{f}{2 * n} \qquad (2)$$

We used this normalized fitness function for both PSO and SA.

3.4 Global Search (Particle Swarm Optimization)

3.4.1 PSO Principles

PSO is a parallel population-based computation technique [22]. It was originally inspired from the flocking behavior of birds, which emerges from very simple individual conducts. Many variations of the algorithm have been proposed over the years, but they all share a common basis. First, an initial population (named swarm) of random solutions (named particles) is created. Then, each particle flies in the n-dimensional problem space with a velocity that is regularly adjusted according to the composite flying experience of the particle and some, or all, the other particles. All particles have fitness values that are evaluated by the objective function to be optimized. Every particle in the swarm is described by its position and velocity. A particle position represents a possible solution to the optimization problem, and velocity represents the search distances and directions that guide particle flying. In this paper, we use basic velocity and position update rules defined by [22]:

$$V_{id+1} = W * V_{id} + C_1 * \mathrm{var}_1 * (P_{id} - X_{id}) + C_2 * \mathrm{var}_2 * (P_{gd} - X_{id}) \quad (3)$$

$$X_{id+1} = X_{id} + V_{id} \quad (4)$$

At each time (iteration), V_{id} represents the particle velocity and X_{id} its position in the search space. P_{id} (also called *pbest* for local best solution), represents the i^{th} particle's best previous position, and P_{gd} (also called *gbest* for global best solution), represents the best position among all particles in the population. w is an inertia term; it sets a balance between the global and local exploration abilities in the swarm. Constants c_1 and c_2 represent cognitive and social weights associated to the individual and global behavior, respectively. There are also two varaibles var_1 and var_2 (normally uniform in the interval [0, 1]) that represent stochastic acceleration during the attempt to pull each particle toward the *pbest* and *gbest* positions. For a n-dimensional search space, the i^{th} particle in the swarm is represented by a n-dimensional vector, $\boldsymbol{x}_i=(x_{i1},x_{i2},\ldots,x_{id})$. The velocity of the particle, *pbest* and *gbest* are also represented by n-dimensional vectors. Algorithm 1 summarizes the generic PSO procedure.

```
PSO algorithm
1: Initial population (particles) creating (initialization)
2: while Termination criterion not met do
3:     for each particle do
4:         Evaluate fitness
5:         Update local/global best (if necessary)
6:         Update velocity and position
```

```
7:      end for
8: end while
9: Return solution corresponding to the global best
```

Algorithm 1. PSO algorithm

3.4.2 PSO for Model transformation

The PSO swarm is represented as a set of *K* particles, each defined by a position vector corresponding to the *n* constructs of the model to transform. For a particle position, the values of the vector elements are the mapping blocks selected for each construct. Our version of PSO starts by randomly generating the particle positions and velocities in the swarm. This is done by randomly affecting a block number to each of the *n* constructs (dimensions). Thus, the initial particle population represents *K* different possibilities (solutions) to transform the source model by combining blocks from the transformation examples. The fitness of each particle is measured by the fitness function defined by Equations 1 and 2.

The particle with the highest fitness is memorized as the global best solution during the search process. At each iteration, the algorithm compares the fitness of each particle with those of the other particles in the population to determine the *gbest* position for use to update the swarm. Then, for each particle, it compares its current positions with *pbest,* and update the latter if an improvement is found. The new positions affect the velocity of each particle according to Equation 3. The algorithm iterates until the particles converge towards a good transformation solution of the source model. In our case, we define a maximum number of iterations after which we select the *gbest* as the transformation solution. The algorithm stops before if all the particles converge to the same solution.

The parameters in Equation 3 have an important effect on the search efficiency of the PSO algorithm. Acceleration constants c_1 and c_2 adjust the amount of "tension" in the system. Low values allow particles to roam far from target regions before being tugged back, while high values result in abrupt movement toward, or past, target regions [40]. Based on past research experience, we set both constants to 1. Equations 3 and 4 may lead to large absolute values for V_{id} and X_{id}, so that a particle may overshoot the problem space.

Therefore, V_{id} and X_{id} should be confined to a maximum velocity V_{max}, and a maximum position X_{max}, such that

$$X_{\max} = N; X_{id} = \min(\max(0, X_{id} + V_{id}), X_{\max}) \qquad (5)$$

V_{max} serves as a constraint to control the global exploration ability of a particle swarm. It should take values in the interval [-*m*, *m*], *m* being the number of blocks in the existing transformation examples. X_{id} represents the block number affected to a construct; it must be a positive integer. Hence, a real value for X_{id} is rounded to the closest block number by dropping the sign and the fractional part.

The inertia weight (*w*) is another important parameter of the PSO search. A proper value for w provides a balance between global and local exploration, and results in less iterations to find a solution on average. In practice, it is often linearly decreased through the course of the PSO, for the PSO to have more global search ability at the beginning of the run and more local search ability near the end. For the validation experience in this paper, the parameter was set as follows [40]:

$$W = W_{\max} - \left(\frac{W_{\max} - W_{\min}}{iter_{\max}} \right) * iter \qquad (6)$$

where W_{max} is the initial value of weighting coefficient, W_{min} is a minimal value of weighting coefficient, $iter_{max}$ is the maximum number of iterations, and *iter* is the current iteration.

3.5 Local Search (Simulated Annealing)

3.5.1 SA Principles

In the case of a quick run of PSO (only a few iterations), the best transformation solution can be improved by using another search heuristic. We propose in this work to use SA in combination with PSO. SA [17] is a search algorithm that gradually transforms a solution following the annealing principle used in metallurgy.

The generic behavior of this heuristic is shown by Algorithm 2. After defining an initial solution, the algorithm iterates on the following three steps:

1 Determine a new neighboring solution,
2 Evaluate the fitness of the new solution
3 Decide on whether to accept the new solution in place of the current one based on the fitness gain/lost.

```
SA algorithm
1: current_solution ← initial_solution
2: current_cost ← evaluate (current_solution)
3: T ← $T_{initial}$
4: while (T > $T_{final}$ ) do
5:     for i=1 to iterations (T) do
6:         new_solution ← move (current_solution)
7:         new_cost ← evaluate(new_solution)
8:         Δcost ← new_cost – current_cost
9:         if (Δcost≤0 OR $e^{-\Delta cost/<T}$ < random() )
10:            current_solution ← new_solution
11:            current_cost ← new_cost
12:        end if
13:    end for
14:  T← next_temp(T)
15: end while
```

Algorithm2. SA algorithm

When $\Delta cost < 0$, the new solution has lower cost than the current solution and it is accepted. For $\Delta cost > 0$ the new solution has higher cost. In this case, the new solution is accepted with probability $e^{-\Delta cost/T}$. The introduction of a stochastic element in the decision process avoids being trapped in a local minimum solution. Parameter T, called temperature, controls the acceptance probability of a lesser good solution. T begins with a high value, for a high probability of accepting a solution during the early iterations. Then, it decreases gradually (cooling phase) to lower the acceptation probability as we advance in the iteration sequence. For each temperature value, the three steps are repeated for a fixed number of iterations.

One attractive feature of the simulated annealing algorithm is that it is problem-independent and can be applied to most combinatorial optimization problems [42, 12]. However, SA is usually slow to converge to a solution.

3.5.2 SA for Model Transformation

To obtain a more robust optimization technique, it is common to combine different search strategies in an attempt to compensate the deficiencies of individual algorithms [12]. In our context, the search for a solution is done in two steps. First, a global search is quickly performed to locate the portion of search space where good solutions are likely to be found. This is performed by PSO and results in a near-optimal solution. In the second step, the obtained solution is refined by the SA algorithm.

As described in Section 3.1, solutions are coded by assigning a block number to each construct to form a vector. The SA algorithm starts with an initial solution generated by a quick run of PSO. As for PSO, the fitness function presented in section 3.3 measures the quality of the solution at the end of each iteration. The generation of a neighboring solution is obtained by randomly changing a number of dimensions with new randomly selected blocks.

The way in which we decrement our temperature is critical to the success of the algorithm. Theory states that we should allow enough iteration at each temperature so that the system stabilises at that temperature. Unfortunately, theory also states that the number of iterations at each temperature to achieve this might be exponential to the problem size. As this is impractical we need to compromise. We can either do this by doing a large number of iterations at a few temperatures, a small number of iterations at many temperatures or a balance between the two. One way to decrement the temperature is use a geometric cooling schedule [17]. The temperature is reduced using:

$$T_{i+1} = \alpha * T_i \qquad (7)$$

where α is a constant less than 1. Experience has shown that α should be between 0.8 and 0.99, with better results being found in the higher end of the range. Of course, the higher the value of α, the longer it will take to decrement the temperature to the stopping criterion.

4 Evaluation and comparison

To evaluate the feasibility of our approach, we conducted an experiment on industrial data. We start by presenting our experimental setting. Then, we describe and discuss the obtained results. We compare in particular the results of PSO with the PSO-SA combination. Finally, we evaluate the impact of the example base size on transformation quality.

4.1 Setting

We used 12 examples of class-diagrams to relational schemas transformations to build an example base $EB = \{<CLD_i, SR_i> \mid 1 \leq i \leq 12\}$. The examples were provided by an industrial partner. As showed in Table 2, the size of the CLDs varied from 28 to 92 constructs, with an average of 58. Altogether, the 12 examples defined 257 mapping blocks. Because our industrial partner uses Rational Rose to derive relational schemas from UML class models, we did not have transformation blocks defined by experts during the transformation. For the need of the experience, we automatically extracted the transformation traces from XMI files produced by Rational Rose. Then, we manually partitioned the traces into blocks.

To evaluate the quality of transformations produced by MOTOE, we used a 12-fold cross validation procedure. For each fold, one class diagram CLD_j is transformed by using the remaining 11 transformation examples ($EB_j = \{<CLD_i, RS_i> \mid i \neq j\}$). Then the transformation result of each fold is checked for correctness. The correctness of a transformation $tCLD_j$ was measured by two methods: automatic correctness (AC) and manual correctness (MC). Automatic correctness consists of comparing the derived transformation $tCLD_j$ to the known RS_j, construct by construct. This method has the advantage of being automatic and objective. However, since a given CLD_j may have different transformation possibilities, *AC* could reject a valid construct transformation because it yields a different RS from the one provided. To account for those situations, we also use *MC* which manually evaluates $tCLD_j$, here again construct by construct. In both cases, the correctness of a transformation is the proportion of constructs that are correctly transformed.

To set the parameters of PSO for the global search strategy, we started with commonly found values in the literature [6, 7] and adapted some of them to the particularities of the transformation problem. The final parameters values were set as follows:

- The swarm is composed of 40 particles. We found this number to provide a good balance between population diversity and the quantity of available examples.
- The inertia weight *W* is initially set to 1.3 and gradually decreased after each iteration according to Equation 6), until it reaches 0.3,
- C_1 and C_2 are both equal to 1 to give equal importance to local and global search.
- The maximum number of iterations is set to twice the size of the population, i.e. 80. This is a generally accepted heuristic [40].
- Since two different executions of a search heuristic may produce different results for the same model, we decided, for each of 12 folds, to take the best result from 5 executions.

As mentioned previously, the initial particle positions are randomly generated. The range of values for each particle coordinate (construct) is defined as [0, *MaxBlocks*] where *MaxBlocks* is the total number of blocks extracted from the 11 examples of the fold. In our case, *MaxBlocks* is 257 minus the number of blocks of the fold example.

For the hybrid search strategy, the SA algorithm was applied using the following parameters:

- The initial temperature of the process is randomly selected in the range [0, 1]
- The geometric cooling coefficient α is 0.98.
- The iteration interval for temperature update is 10000 (to account for SA's slowness).
- The number of dimensions to change for generating a neighboring solution is set to 2. This value offers a good balance with the large number of iterations.
- The stopping criterion (temperature threshold) is 0.1

To quickly generate an initial solution for SA, we limited the number of particles to 10 and the number of iterations to 20 for PSO.

With these parameter values, the transformation of largest diagrams took only a few seconds of run time. We also tried other parameters and obtained similar results each time.

4.2 Results and Discussion

4.2.1 Results

Tables 2 and 3 respectively show the obtained correctness for each of the 12 folds, when using global and hybrid search. Both automatic and manual correctness values were high and, as expected, manual evaluation yielded better correctness since it considered all correct transformations and not only the specific alternatives chosen by our industrial partner. We consider correctness values (74% and 94% for respectively the automatic and the manual validation) as relatively high relatively given the context of no transformation rules and the limited number of used examples.

Table 2 shows the correctness of the generated transformations using the PSO heuristic. The automatic correctness measure had an average value of 73.3%, with most of the models transformed with at least 70% precision. The manual correctness measure was much greater, with an average value of 93.2%; this indicates that the proposed transformations were almost as correct as the ones given by experts. The worst model (SM9) had an acceptable MC of 87% and four models obtained an MC greater than 95%, with a value of 98,1% for SM8.

Source Model	**Number of constructs**	**Fitness**	**AC**	**MC**
SM 1	72	0.696	0.618	0.882
SM 2	83	0.714	0.682	0.928

SM 3	49	0.762	0.721	0.943
SM 4	53	0.796	0.719	0.931
SM 5	38	0.773	0.789	0.952
SM 6	47	0.746	0.652	0.918
SM 7	78	0.715	0.772	0.957
SM 8	34	0.896	0.822	0.981
SM 9	92	0.61	0.634	0.87
SM 10	28	0.892	0.908	0.969
SM 11	59	0.773	0.717	0.915
SM 12	63	0.805	0.762	0.938
Average	*58*	*0.764*	*0.733*	*0.932*

Table 2. 12-fold cross validation with PSO

The hybrid search gave slightly better correctness results as shown in Table 3. Both automatic and manual correctness were slightly better on average (93.4% for AC and 94.8 for MC). With regards to MC, the quality of 8 model transformations was improved while that of 4 was slightly degraded. For instance, MC for the worst transformed model (SM9) improved from 87% to 93.1%, while that for the best transformed model (SM5) decreased from 95.2% to 93%.

Source Model	**Number of constructs**	**Fitness**	**AC**	**MC**
SM 1	72	0.735	0.696	0.947
SM 2	83	0.784	0.723	0.962
SM 3	49	0.632	0.69	0.912

SM 4	53	0.619	0.672	0.956
SM 5	38	0.742	0.733	0.93
SM 6	47	0.737	0.704	0.953
SM 7	78	0.743	0.79	0.942
SM 8	34	0.845	0.813	0.975
SM 9	92	0.648	0.667	0.931
SM 10	28	0.846	0.873	0.983
SM 11	59	0.796	0.73	0.92
SM 12	63	0.772	0.72	0.964
Average	*58*	*0.742*	*0.734*	*0.948*

Table 3. 12-fold cross validation with PSO-SA

4.2.2 Discussion

One observation to be made from the results in Tables 2 and 3 is that, with the exception of model SM7, hybrid search yielded better results than global search for the models with the highest numbers of constructs. This may be due to the fact that, when the number of dimensions is high, the search space is very large and the use of PSO leads to particle movement steps that can only approximate the location of the target solution. A more focused search consisting of global search followed by local exploration produces better results in this case. In contrast, for a smaller search space (less dimensions), area coverage by the particles is easier, and a global search appears to be more efficient to zero in on the solution.

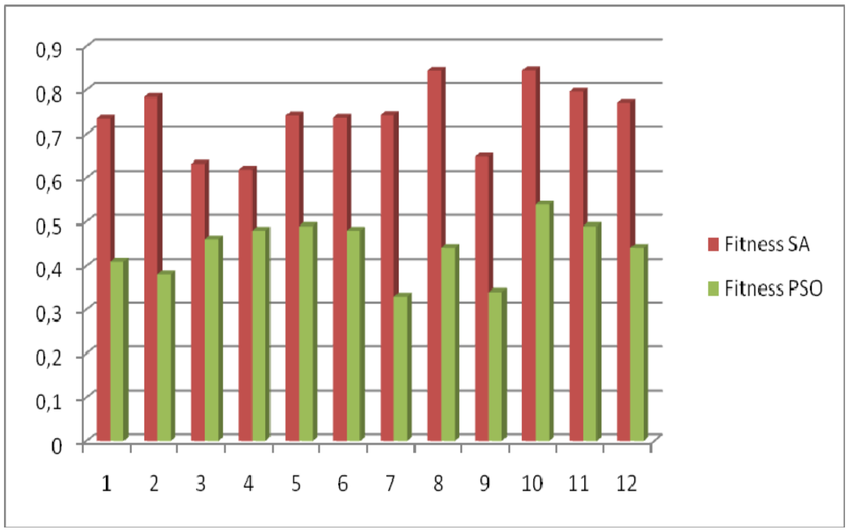


Fig 11. Fitness improvement with SA after PSO initial pass

To better analyze the performance of the hybrid strategy, Figure 11 shows, for all models, the average final values of the fitness function after the quick global search with PSO and their corresponding values after the refinement made by SA. As one can see, substantial fitness improvement occurred (more than 50% in many cases) in each case of the 12-fold cross validation. It appears then that the hybrid strategy brings a good compromise between correctness and execution time. Indeed, it allows improving the transformation correctness with a slight increase in the execution time. The obtained results also show that our fitness function is a good estimator of transformation correctness.

An important consideration is the impact of the example base size on transformation quality. Drawn for SM7, the results of Figures 12 and 13 show that our approach also proposes transformation solutions in situations where only few examples are available. When using the global search strategy, AC seems to grow steadily and linearly with the number of examples. For the hybrid strategy, the correctness seems to follow an exponential curve; it rapidly grows to acceptable values and then slows down. Indeed, AC improved from roughly 30% to 65% as the example base size went from 1 to 4 examples. Then, it grew only by an additional 15% as the size varied from 6 to 11 examples.

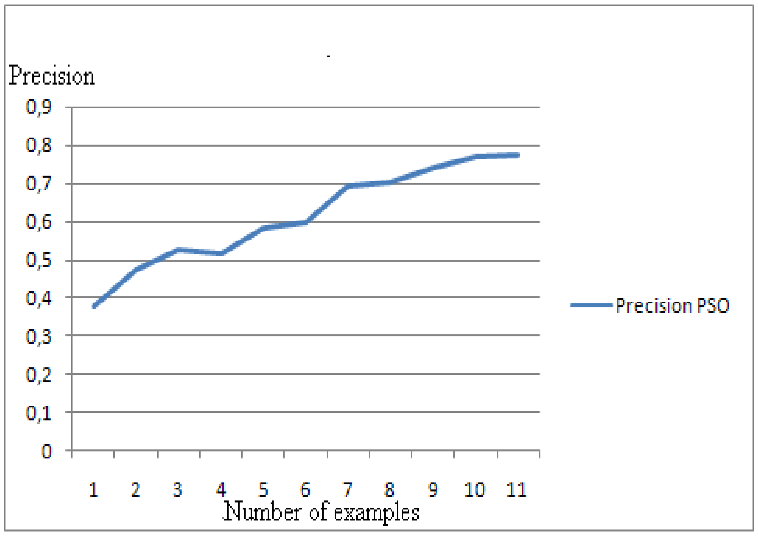


Fig 12. Example-size variation with PSO

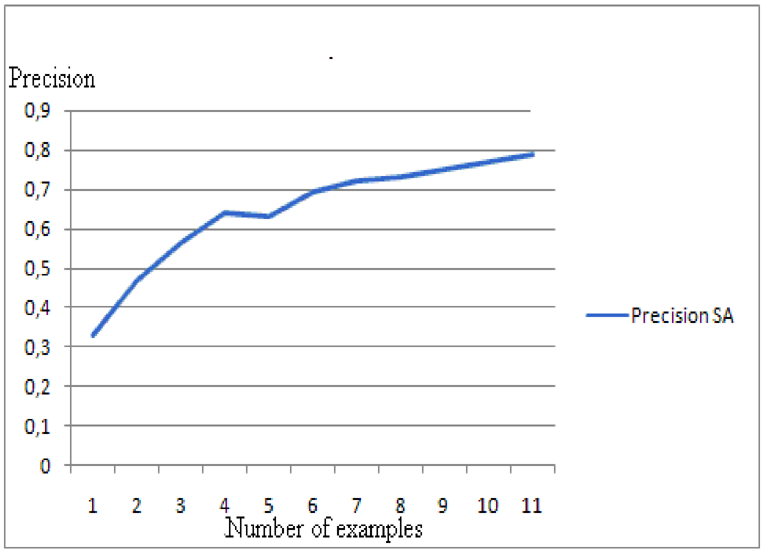


Fig 13. Example-size variation with PSO-SA

When manually analyzing the results, we noticed that some of the 12 models had constructs not present in the other models. Those constructs were generally not transformed as not adequate block could be found for them. However, some others were transformed by adapting the transformation of constructs of the same nature. This was the case, for instance, for an association with multiplicity (1..*N,* 1..*N*). Since the multiplicity elements are considered as parameters of the construct, the transformation of an association (0..*N,* 0..*N*) was applied with a penalty on the fitness function. Although these few cases of adaptation improved the global correctness scores, we did not specifically address the issue at the current stage of our research.

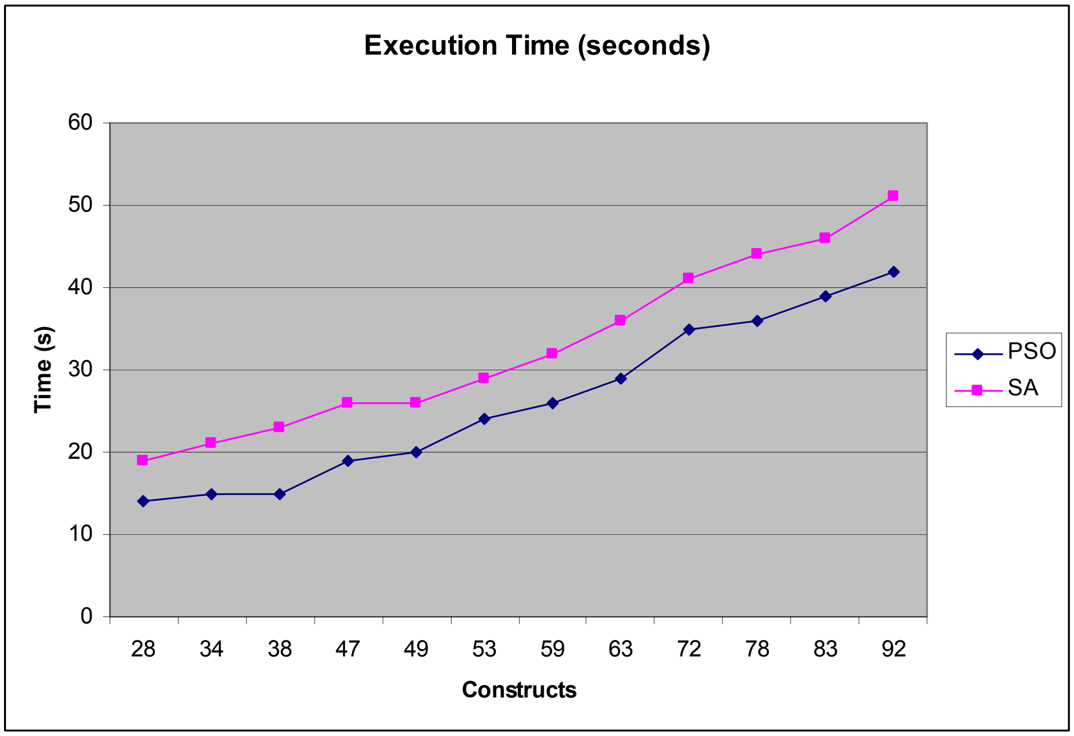


Fig 14. Execution time

Finally, since we viewed the transformation problem as a combinatorial problem addressed with heuristic search, it is important to contrast the correctness results with the execution time. We executed our algorithm on a standard desktop computer (Pentium CPU running at 2 GHz with 1GB of RAM). The execution time is shown in Figure 14. As suggested by the curve shape, there were no performance problems when transforming models up to 100 elements that corresponds to small and medium models. It should be noted, however, that more important execution times may be obtained in comparison with using rule-based tools for small-dimensional problems. In any case, our approach is meant to apply to situations where rule-based solutions are normally not readily available.

5 Related Work

The work proposed in this paper can be related to three research areas in software engineering, of which the most relevant one is MT in the context of MDD. Some links can also be found with by-example and search-based software engineering, but our concerns are different as will be discussed below. As a result, only a comparison to alternatives in the first area is warranted.

5.1 Model Transformation

Several MT approaches can be found in the literature (see, for example, the classifications given in [24, 44]). Czarnecki and Helsen [24] distinguish between two main types: model-to-model and model-to-code. They describe five categories of the former: Graph-transformation-based [25], relational [11], structure-driven [21], direct-manipulation and hybrid. They use various criteria to analyze them, like the consideration of Model Driven Architecture (MDA) as a basis for transformation, the complexity and scalability of the transformation mechanism, the use or not of annotations, the level of automation, and the used languages and implementation techniques. In general, the reported approaches are based on empirically obtained rules [2, 3] in contradistinction to block transformation in MOTOE. In rules-based approaches, the rules are defined in metamodels while our blocks relate to specific problems, with a varying structure, for different problems.

In existing transformation approaches, likes graph-transformation [25, 58, 59], a transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS accesses the source model, whereas the RHS expands in the target model. By comparison, each block in MOTOE contains a transformation of source elements (LHS) to their equivalents target elements (RHS). However, in a graph-transformation approach, potentials conflicts between transformation elements are verified with pre and post condition. In our case, pre and post conditions are replaced by the fitness function that ensures transformations coherency.

In rule based approaches, a rule is applied to a specific location within its source scope. Since there may be more than one match for a rule within a given source scope, we need an application strategy. The strategy could be deterministic, non-deterministic or even interactive [60]. For example, a deterministic strategy could exploit some standard traversal strategy (such as depth-first) over the containment hierarchy in the source. In our work, the transformation possibilities (blocks) are randomly chosen with .no strategy for rules application (rules scheduling, etc).

Transformation rules are usually designed to have a functional character: given some input in the source model, they produce a concrete result in the target model [18]. A declarative rule (i.e., one that only uses declarative logic and/or patterns) can often be applied in the inverse direction. However, since different inputs may lead to the same output, the inverse of a rule may not be a function. We have the same problem in our approach since, blocks only defined in one direction (from CLD to RS for example). To ensure a bidirectional transformation property, we need to apply our methodology to examples from the other direction.

If we define cognitive complexity as the level of difficulty to design a model transformation, we believe that collecting/recording transformation examples may be less difficult than producing and maintaining consistent transformation rule sets. This is consistent with recent trend in industry where we find several tools to record transformations and automatically generate transformation traceability records [61].

The traditional approach for implementing model transformations is to specify transformation rules and automate the transformation process by using a model transformation language [23]. Most of these languages are powerful enough to implement large-scale and complex model transformation tasks. However, the transformation rules are usually defined at the metamodel level, which requires a clear and deep understanding about the abstract syntax and semantic interrelationships between the source and target models. In some cases, domain concepts may be hidden in the metamodel and difficult to unveil [2, 3] (e.g., some concepts are hidden in attributes or association ends, rather than being represented as first-class entities). These implicit concepts may make writing transformation rules difficult.

To help address the previous challenges, an alternative approach called Model Transformation By Example (MTBE) was proposed in [13, 15].In it, users are asked to build a prototypical set of interrelated mappings between the source and target model instances, and then the metamodel-level transformation rules will be semi-automatically generated. Because users configure the mappings at the instance level, without knowing any details about the metamodel definition or the hidden concepts, combined with the generated rules, the simplicity of specifying model transformations can be improved. Varrò and Balogh [13, 15] propose a semi-automated process for MTBE using Inductive Logic Programming (ILP). The principle of their approach is to derive transformation rules semi-automatically from an initial prototypical set of interrelated source and target models. Another similar work is that of Wimmer et al. [31] who derive ATL transformation rules from examples of business process models. Both works use semantic correspondences between models to derive rules. Their differences include the fact that [31] presents an object-based approach that finally derives ATL rules for model transformation, while [13] derives graph transformation rules. Another difference is that they respectively use abstract versus concrete syntax: Varro uses IPL when Wimmer relies on an ad hoc technique. Both approaches provide a semi-automatic generation of model transformation rules that needs further refinement by the user. Also, since both approaches are based on semantic mappings, they are more appropriate in the context of exogenous model transformations between different metamodel. Unfortunately, the generation of rules to transform attributes is not well supported in most MTBE implementations. Our model is different from both previous approaches to MTBE. We do not create transformation rules to transform a source model, directly using examples instead. As a result, our approach is independent from any source or target formalisms.

Recently, a similar approach to MTBE, called Model Transformation By Demonstration (MTBD), was proposed [62]. Instead of the MTBE idea of inferring the rules from a prototypical set of mappings, users are asked to demonstrate how the model transformation should be done, through direct editing (e.g., add, delete, connect, update) of the source model, so as to simulate the transformation process. A recording and inference engine was been developed, as part of a prototype called MT-Scribe, to capture all user operations and infer a user's intention during a model transformation task. A transformation pattern is generated from the inference, specifying the preconditions of the transformation and the sequence of operations needed to realize the transformation. This pattern can be reused by automatically matching the preconditions in a new model instance and replaying the necessary operations to simulate the model transformation process. However, this approach needs a large number of simulated patterns to be efficient and it requires a high level of user intervention. In fact, the user must choose the suitable transformation pattern. Finally, the authors do not show how MTBD can be useful to transform an entire source model and only provide examples of transforming model fragments.

Some others metamodel matching works can be also considered as a variant of By-example approaches. Garcia-Magarino et al. [66] proposes an approach that generates transformation rules between two meta-models that satisfies some constraints introduced manually by the developer. In [65], authors propose to capture automatically some transformation patterns in order to generate some matching rules in the metamodel level. This approach is similar to MTBD but it is used in the meta-model level. The difference of this category of approaches with our proposal that we do not generates transformation rules and MOTOE do not need to specify the source and target metamodels as input.

To conclude, the previous problems limit the applicability of MTBE/MTBD for some transformation problems. In such situations, MOTOE may leads to more relevant solutions.

In our approach, the definition of transformation examples is based on the use of traceability information [61]. Traceability usually allows tracing artifacts within a set of chained operations, where the operations may be performed manually (e.g., crafting a software design for a set of software requirements) or with automated assistance (e.g., generating code from a set of abstract descriptions). For example, Triple Graph Grammars (TGG) [63] explicitly maintains the correspondence of two graphs by means of correspondence links. These correspondence links play the role of traceability links that map elements of one graph to elements of the other graph and vice versa. With TGG, one has to explicitly describe correspondence between the source and target models, which is difficult if the transformation is complex and the intermediate models are required during the transformation. In [52], a traceability framework for Kermeta is discussed. This framework supports the creation of traces throughout a transformation chain. Marvie describes a transformation composition framework [64] that allows manual creation of linkings (traces). However, this framework do not support the automatic generation of traces. In conclusion, A large part of the work on traceability in MDE uses it for detecting model inconsistency and fault localization in transformations. In MOTOE, this is not the goal as the purpose is to use trace information as input to automate the transformation process. The traces information (model correspondence) between a source and target models define a transformation example that is decomposed in some independent blocks as explained before.

Our approach is different from case-based reasoning methods where the level of granularity must be the example as a whole, i.e., a transformation example [8]; in our case, we do not select the most similar example and adapt its transformation; rather, we aggregate the best transformation possibilities coming from different examples.

5.2 By-Example Software Engineering

The approach proposed in this paper is based on using past transformation examples. Various such “by-example” approaches have been proposed in the software engineering literature. However, the problems addressed by them differ from ours in both nature and objectives. The closest work to ours is program transformations by demonstration [1, 5], in which a user manually changes program examples while a monitoring plug-in to the development environment records the changes . Then, the recorded data are analyzed to create general transformations that can be reused in subsequent programs. However, the overall process is not automated and requires frequent interaction with the user, and the generated transformation patterns are found via a different algorithms than the one used by MOTOE.

5.3 Search-Based Software Engineering

Our approach is inspired by contributions in Search-Based Software Engineering (SBSE) [26, 28]. As the name indicates, SBSE uses a search-based approach to solve optimization problems in software engineering. Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, fitness function and solution change operators, there are many search algorithms that can be applied to solve that problem. To the best of our knowledge, inspired among others by the road map paper of

Harman [28], the idea of treating model transformation as a combinatorial optimization problem to be solved by a search-based approach was not studied before our proposal in [29]. For this reason, we can not compare our approach to existing works in SBSE because the application domain is very different.

6 Summary and Conclusion

In summary, we described MOTOE, a novel approach to automate MT using heuristic-based search. MOTOE uses a set of transformation examples to derive a target model from a source model. The transformation is seen as an optimization problem where different transformation possibilities are evaluated and, for each possibility, a quality is associated depending on its conformance with the examples at hand. The search space is explored with two methods. In the first one, we use PSO with transformation solutions generated from the examples at hand as particles. Particles progressively converge toward a good solution by exchanging and adapting individual construct transformation possibilities. In the second method, a partial run of PSO is performed to derive an initial solution. This solution is then refined using a local search with SA. The refinement explores neighboring solutions obtained by trying individual construct transformation possibilities derived from the example base. In both methods, the quality of a solution considers the adequacy of construct transformations as well as their mutual consistency.

We illustrated MOTOE with the transformation of UML class diagrams to relational schemas. In this context, we conducted a validation with real industrial models. The experiment results clearly indicate that the derived models are comparable to those proposed by experts (correctness of more than 90% with manual evaluation). They revealed also that some constructs were correctly transformed although no transformation examples were available for them. This was possible because the approach uses syntactic similarity between construct types to adapt their transformations. We also showed that the two methods used for the space search produced comparable results when properly applied, and that PSO alone is enough with small-to-medium models while the combination PSO-SA is more suitable when the size of the models to transform is larger. For both methods, our transformation process derives a good quality transformation in an acceptable execution time. Finally, the validation study showed that the quality of MT improves with the number of examples. However, it reaches a stable score after as few as nine examples. Also, there were no performance problems when transforming models up to 100 elements that corresponds to small and medium models.

Our proposed method also has limitations. First, MOTOE's performance depends on the availability of transformation examples, which could be difficult to collect. Second, the generation of blocks from the examples is done manually in our present work; we could partially automate this task using decomposition heuristics. Third, due to the nature of our solution, *i.e.,* an optimization technique, the transformation process could be time consuming for large models. Finally, as we use heuristic algorithms, different execution for the same source models could lead to different target models. Nevertheless, we showed in our validation that solutions that have high fitness values also have good correctness. Moreover, this is close to what happens in the real world where different experts could propose different target models.

From the applicability point of view, our approach can theoretically be applied to the transformation of any pair of formalisms. To practically assess this clam, we are currently experimenting with other formalisms such as sequence diagrams to Petri nets. We also plan to work on adapting our approach to other transformation problems such as code generation (model-to- code), refactoring (code-to-code), and reverse engineering (code-to-model). The refactoring problem also has the advantage of exploring endogenous transformations where source and target models conform to the same metamodel. Regarding the quality evaluation of transformations, the fitness function we used could be improved. In this work, we gave equal importance to all constructs. In the real world, some construct types may be more important than others.

References

1. A. Cypher (ed.). Watch What I Do: Programming by Demonstration. The MIT Press, (1993)
2. A. Egyed, Automated abstraction of class diagrams. ACM Trans. Softw. Eng.Methodol. 11(4): 449-491 (2002).

3. A. Egyed, Heterogeneous Views Integration and its Automation, Ph.D. Thesis,Univ. of Southern California, (2000)
4. A. Kleppe , J.Warmer J and W. Bast MDA Explained. The model driven architecture: practice and promise. Addison-Wesley, (2003)
5. A. Repenning and C. Perrone. Programming by example: programming by analogous examples. Comm. of the ACM, vol. 43(3):pp. 90–97, (2000)
6. A. Salman, A. Imtiaz, and Al-Madani, S.: Particle swarm optimization for task assignment problem. In: IASTED Intl. Conf. on Artificial Intelligence and Applications (2001)
7. A. Windisch,, S. Wappler, and J. Wegener,. Applying particle swarm optimization to software testing. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (London, England, July 07 - 11, 2007). GECCO '07. ACM, New York, NY, pp. 1121-1128, (2007)
8. A.Aamodt and E.Plaza, "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches," *Artificial Intelligence Communications* 7 : 1, 39-52(1994)
9. ATLAS Group. The ATLAS Transformation Language. http://www.eclipse.org/gmt.
10. Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, http://www.omg.org/cgi-bin/doc?ad/2003-08-07 (2003)
11. D. H. Akehurst and S.Kent. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.): *UML 2002 – The* Unified Modeling Language 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings, LNCS 2460, 243-258, (2002)
12. D. Mitra, F. Romeo, and A. Sangiovanni-Vincentelli, "Convergence and Finite-Time Behaviour of Simulated Annealing," *Proc 1985* Decision and Control Con5 , (1985).
13. D. Varro and Z. Balogh, Automating Model Transformation by Example Using Inductive Logic Programming, ACM Symposium on Applied Computing | Model Transformation Track, (2007)
14. D. Varro, A. Pataricza : Generic and meta-transformations for model transformation engineering. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273. Springer, Heidelberg (2004)
15. D. Varro. Model transformation by example. In Proc.MODELS 2006, pp. 410–424, Vol. 4199 of LNCS. Springer, (2006)
16. D.S. Coming and O.G. Staadt, "Velocity-Aligned Discrete Oriented Polytopes for Dynamic Collision Detection," *IEEE Trans. Visualization and Computer Graphics*, vol. 14, no. 1, pp. 1-12 , doi:10.1109/TVCG.2007.70405, (2008)
17. D.S. Kirkpatrick, Jr. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671680, (1983)
18. F. Jouault, and I. Kurter,: Transforming models with ATL. In: Proc. Of the Model Transformations in Practice Workshop at MoDELS 2005, Jamaica (2005)
19. G. Taentzer : AGG: a graph transformation environment for system modeling and validation. In: Proc. Tool Exihibition at Formal Methods 2003, Pisa, Italy (2003)
20. Interactive Objects and Project Technology, MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-11, ad/03-08-12, ad/03-08-13 (2003)
21. Interactive Objects Software GmbH, Project Technology, Inc. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-11, http://www.omg.org/cgi-bin/doc?ad/2003-08-11 (2003)
22. J. Kennedy, and R.C Eberhart : Particle swarm optimization. In: Proc. IEEE Intl.Conf. on Neural Networks, pp. 1942–1948 (1995)
23. K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal, special issue on Model-Driven Software Development. 45(3), pp. 621-645 (2006)
24. K. Czarnecki, and S. Helsen, Classification of model transformation approaches.OOSPLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, USA, (2003)

25. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph Transformation for Specification and Programming. Technical Report 7/96, Universität Bremen, see http://citeseer.nj.nec.com/article/andries96graph.html (1996)
26. M. Harman and B. F. Jones, Search-based software engineering, *Information & Software Technology*, Vol. 43, No. 14, pp. 833-839 (2001).
27. M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *GECCO'07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106{1113,New York, NY, USA ACM Press, (2007)
28. M. Harman, The Current State and Future of Search Based Software Engineering, In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, 20-26 May, Minneapolis, USA (2007)
29. M. Kessentini, H.Sahraoui and M.Boukadoum Model Transformation as an Optimization Problem. In Proc.MODELS 2008, pp. 159-173 Vol. 5301 of LNCS. Springer, (2008)
30. M. Siikarla and T. Syst, Decision Reuse in an interactive Model Transformation, 12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece, 123-132, (2008)
31. M. Wimmer, M. Strommer, H. Kargl, and G. Kramler.Towards model transformation generation by-example. In Proc. of HICSS-40 Hawaii International Conference on System Sciences. Hawaii, USA., (2007)
32. O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909{1916, New York, NY, USA, (2006)
33. Object Management Group (OMG), Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Final Adopted Specification, ptc/05-11-01, http://www.omg.org/docs/ptc/05-11-01.pdf (2005)
34. OMG: Meta Object Facility (MOF). Version 1.4. www.omg.org/technology/documents/formal/mof.htm
35. OMG: Request For Proposal: MOF 2.0/QVT. OMG Document ad/2002-04-10, http://www.omg.org/cgi-bin/doc?ad/2002-04-10 (2002)
36. QVT-Merge Group. MOF 2.0 Query/Views/Transformations RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, http://www.omg.org/cgi-bin/doc?ad/2004-04-01 (2004)
37. R. France, and B. Rumpe,: Model-driven Development of Complex Software: A Research Roadmap. In: Briand, L., Wolf, A. (eds.) Intl. Conf. on Software Engineering (ICSE 2007): Future of Software Engineering. IEEE Computer Soceity Press,Los Alamitos (2007)
38. R. Heckel, J.M .Kuster, and G.Taentzer, Confluence of Typed Attributed Graph Transformation *Systems*. Proc. ICGT'02, LNCS 2505, pp.: 161-176, Springer, (2002)
39. R. Krishnamurthy, S.P. Morgan, M.M. Zloof: Query-By-Example: Operations on Piecewise Continuous Data. Proc. 9th International Conference on Very Large Data Bases, October 31 - November 2, Florence, Italy., pp. 305-308 (1983)
40. R.C. Eberhart and Y. Shi : Particle swarm optimization: developments, applications and resources. In: Proc. IEEE Congress on Evolutionary Computation (CEC 2001), pp. 81–86 (2001)
41. S. Bouktif, H. Sahraoui, and G. Antoniol. Simulated annealing for improving software quality prediction. In *GECCO2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1893–1900, (2006)
42. S. Geman and *D.* Geman, "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images,'' IEEE Trans. Pat. Ana. and Mach. Intel. PAMI-6(6) pp. 721-741 (1984).
43. T. Clark and J. Warmer, Object Modeling with the OCL, The Rationale behind the Object Constraint Language. Springer-Verlag, London, UK (2002)
44. T. Mens and P. Van Gorp, A Taxonomy of Model Transformation, Proc. Intl. Workshop on Graph and Model Transformation, (2005)

45. U. Behrens, M. Flasinski, L.Hagge, J. Jurek, and K.Ohrenberg, Recent developments of the ZEUS expert system ZEX, IEEE Trans. Nucl. Sci., NS-43, pp. 65-68, (1996)
46. Y. DuanCheung, X. Fu and Y. Gu, A metamodel based model transformation approach, Proc. ACIS Intl. Conf. on Software Engineering Research, Management and Applications,pp.184-191, (2005)
47. M. Kessentini, A. Bouchoucha, H. Sahraoui and M. Boukadoum Example-based Sequence Diagrams to Colored Petri Nets Transformation using Heuristic Search, proceedings of 6th European Conference on Modeling Foundations and Applications (ECMFA 2010) , Paris.
48. J.R Falleri, M. Huchard, M. Lafourcade, Clémentine Nebut Meta-model Matching for Automatic Model Transformation Generation, ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008), (2008)
49. I. Galvão and A. Goknil, "Survey of Traceability Approaches in Model-Driven Engineering". EDOC'07, pages 313-326, (2007)
50. F. Jouault, Loosely coupled traceability for atl. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability. (2005)
51. R. Marvie, A transformation composition framework for model driven engineering. Technical Report LIFL-2004-10, LIFL (2004)
52. J.R Falleri., M. Huchard, C. Nebut, Towards a traceability framework for model transformations in kermeta, HAL - CCSd - CNRS (2006)
53. S. Lechner and M. Schrefl. Defining web schema transformers by example. In Proceedings of DEXA'03. Springer, (2003)
54. M. Shousha, L. Briand, Y. Labiche, A UML/SPT Model Analysis Methodology for Concurrent Systems Based on Genetic Algorithms. In Proceedings of the 11th international Conference on Model Driven Engineering Languages and Systems MODELS08, 475-489, (2008)
55. J. M. Küster, S. Sendall, M. Wahler *Comparing Two Model Transformation Approaches* Proceedings UML 2004 Workshop OCL and Model Driven Engineering, Lisbon, Portugal, October 12, (2004)
56. H. Bunke, Graph matching: theoretical foundations, algorithms, and applications, in: Proceedings of the Vision Interface 2000, Montreal/Canada, pp. 82–88, (2000)
57. T. Mens, P. Van Gorp, D. Varro, and G. Karsai. Applying a Model Transformation Taxonomy to Graph Tansformation Technology. In G. Karsai, and G. Taentzer, editors, Proceedings of Graph and Model Transformation Workshop, to appear in ENTCS, (2005)
58. Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovsky, Ulrike Prange, Daniel Varro, and Szilvia Varro-Gyapay. Model transformation by graph transformation: A comparative study. In Workshop on Model Transformations in Practice, September 2005.
59. T. Mens, P. Van Gorp, D. Varro, and G. Karsai. Applying a Model Transformation Taxonomy to Graph TRansformation Technology. In G. Karsai, and G. Taentzer, editors, Proceedings of Graph and Model Transformation Workshop, ENTCS, 2005.
60. E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of Lecture Notes in Computer Science, pages 216--238. Spinger-Verlag, June 2004.
61. B Vanhoof, S Van Baelen, W Joosen, Y Berbers. Traceability as Input for Model Transformations. In Proc. Traceability Workshop, European Conference in Model Driven Architecture (EC-MDA), 2007.
62. Yu Sun, Jules White, and Jeff Gray, "Model Transformation by Demonstration," *MoDELS09*, Denver, CO, October 2009, pp. 712-726.
63. H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Models '06: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 543–557. Springer Verlag, October 2006.

64. D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings, volume 4199 of LNCS, pages 321{335. Springer, Oct. 2006.
65. Marcos Didonet Del Fabro, Patrick Valduriez: Towards the efficient development of model transformations using model weaving and matching transformations. Software and System Modeling 8(3): 305-324 (2009)
66. I. Garcia-Magarino, J. J. Gomez-Sanz, and R. Fuentes-Fernandez: Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In Proc. of the 2nd Int. Conf. on Theory and Practice of Model Transformations (ICMT'09), volume 5563 of LNCS, pages 52-66. Springer, 2009.

Chapter 4: Dynamic Model Transformation by Example

4.1 Introduction

After presenting the case of static models chapter3, we describe in this chapter our solution for the problem of automating dynamic model transformation using examples. We adapt our approach MOTOE to the sequence diagram to colored Petri Nets transformation. This contribution was accepted for publication in the Sixth European Conference on Modelling Foundations and Applications (ECMFA 2010) [74]. The paper, entitled "Example-based Sequence Diagrams to Colored Petri Nets Transformation using Heuristic Search", is presented next.

4.2 Sequence Diagrams to Colored Petri Nets Transformation by Example

Example-based Sequence Diagrams to Colored Petri Nets Transformation using Heuristic Search

Marouane Kessentini[1], Arbi Bouchoucha[1], Houari Sahraoui[1] and Mounir Boukadoum[2]

Abstract. Dynamic UML models like sequence diagrams (SD) lack sufficient formal semantics, making it difficult to build automated tools for their analysis, simulation and validation. A common approach to circumvent the problem is to map these models to more formal representations. In this context, many works propose a rule-based approach to automatically translate SD into colored Petri nets (CPN). However, finding the rules for such SD-to-CPN transformations may be difficult, as the transformation rules are sometimes difficult to define and the produced CPN may be subject to state explosion. We propose a solution that starts from the hypothesis that examples of good transformation traces of SD-to-CPN can be useful to generate the target model. To this end, we describe an automated SD-to-CPN transformation method which finds the combination of transformation fragments that best covers the SD model, using heuristic search in a base of examples. To achieve our goal, we combine two algorithms for global and local search, namely Particle Swarm Optimization (PSO) and Simulated Annealing (SA). Our empirical results show that the new approach allows deriving the sought CPNs with at least equal performance, in terms of size and correctness, to that obtained by a transformation rule-based tool.

Keywords: Model transformation, Petri nets, Sequence diagrams, Search-based software engineering

1 Introduction

Model Transformation plays an important role in Model Driven Engineering (MDE) [1]. The research efforts by the MDE community have produced various languages and tools, such as ATL [2], KERMETA [3] and VIATRA [4], for automating transformations between different formalisms. One major challenge is to automate these transformations while preserving the quality of the produced models [1, 6].

Many transformation contributions target UML models [1, 6]. From a transformation perspective, UML models can be divided into two major categories: static models, such as class diagrams, and dynamic models, such as activity and state diagrams [7]. Models of the second category are generally transformed for validation and simulation purposes. This is because UML dynamic models, such as sequence diagrams (SDs) [7], lack sufficient formal semantics [8]. This limitation makes it difficult to build automated tools for the analysis, simulation, and validation of those models [9]. A widely accepted approach to circumvent the problem uses concomitant formal representations to specify the relevant behavior [11]; Petri Nets (PNs) [10] are well suited for the task. PNs can model, among others, the behavior of discrete and concurrent systems. Unlike SDs, PNs can derive new information about the structure and behavior of a system via analysis. They can be validated, verified, and simulated [11]. Moreover, they are suitable for visualization (graphical formalism) [11]. These reasons motivate the work to transform UML SDs to PNs.

SD-to-PN transformation may be not obvious to realize, due to two main reasons [29]. First, defining transformation rules can be difficult since the source and target languages have constructs with different semantics; therefore, 1-to-1 mappings are not sufficient to express the semantic equivalence between constructs. The second problem is the risk of a state explosion [11]. Indeed, when transformation rules are available for mapping dynamic UML models to PNs, systematically applying them generally results in large PNs [11]. This could compromise the subsequent analysis tasks, which are generally limited by the number of the PNs' states. Obtaining large PNs is not usually related to the size of the source models [29]. In fact, small sequence diagrams containing complex structures like references, negative traces or critical regions can produce large PNs. To address this problem, some work has been done to produce reduction rules [35].

In this paper, we explore a solution based on the hypothesis that traces of valid transformations of SD-to-PN (performed manually for instance), called transformation examples, can be used by similarity to derive a PN from a particular SD. In this context, our approach, inspired by the Model-Transformation-by-Examples (MTBE) school [12, 13, 14], helps define transformations without applying rules. Because it reuses existing valid model transformation fragments, it also limits the size of the generated models.

More concretely, to automate SD-to-PN transformations, we propose to adapt, the MOTOE approach [14, 15]. MOTOE views a model transformation as an optimization problem where solutions are combinations of transformation fragments obtained from an example base. However, the application of MOTOE to the SD-to-PN transformation problem is not straightforward. MOTOE was designed for and tested with static-diagram transformations such as class-diagrams-to-relational schemas [14, 15]. The transformation of a dynamic diagram is more difficult [8] because, in addition to ensuring structural (static) coherence, it should guarantee behavioral coherence in terms of time constraints and weak sequencing. For instance, the transformation of a SD message depends on the order (sequence) inside the diagram and the events within different operands (parallel merge between the behaviors of the operands, choice between possible behaviors, etc.).

This paper adapts and extends MOTOE to supports SD-to-CPN transformation. The new version, dMOTOE, preserves behavioral coherence. We empirically show that the new approach derives the correct models, and that the obtained CPNs have a significantly lower size than those obtained with a rule-based tool [16] taken for comparison.

The remainder of this paper is structured as follows. In section 2, we provide an overview of the proposed approach for automating SD-to-CPN transformations and discuss its rationale in terms of problem complexity. Section 3 describes the transformation algorithm based on the combined PSO and SA search heuristics. An evaluation of the algorithm is explained and its results are discussed in Section 4. Section 5 is dedicated to the related work. Finally, concluding remarks and future work are provided in section 6.

2 SD-to-CPN Transformation Overview

A model transformation takes a model to transform as input, the source model, and produces another model as output, the target model. In our case, the source model is a UML sequence diagram and the target model is a colored Petri net. First, we describe the principles of our approach and discuss the rationale behind given the complexity of the transformation problem.

2.1 Overview

dMOTOE takes a SD to transform and a set of transformation examples form an example base as inputs, and generates an equivalent CPN as output. The generation process can be viewed as selecting the subset of the transformation fragments (mapping traces) in the example base that best matches the constructs of the SD according to a similarity function. The outcome is a CPN consisting of an assembly of building blocks (formally defined below). The quality of the produced target model is measured by the level of conformance of the selected fragments to structural and temporal constraints, i.e., by answering the following three questions: 1) Did we choose the right blocks? 2) Did they fit together? 3) Did we perform the assembly in the right order?

As many block assembly schemes are possible, the transformation process is a combinatorial optimization problem where the dimensions of the search space are the constructs of the SD to transform. A solution is determined by the assignment of a transformation fragment (block) to each SD construct. The search is guided by the quality of the solution in terms of its internal coherence (individual construct vs. associated blocks), external coherence (between blocks) and temporal coherence (message sequence). To explore the solution space, the search is performed in two steps. First, we use a global heuristic search by means of the PSO algorithm [18] to reduce the search space size and select a first transformation solution. Then, a local heuristic search is done using the SA algorithm [19] to refine this solution. In order to provide the details of our approach, we define some terms.

A ***construct*** is a source or target model element; for example, messages or objects in a SD. An element may contain ***properties*** that describe it such as its name. Complex constructs may contain ***sub-constructs.*** For example, a message could have a guard that conditions its execution.

A ***Transformation example*** (TE) is a mapping of constructs from a particular SD to a CPN. Formally, we view a TE as a triple <*SMD, TMD, MB*> where SMD denotes the source model (SD in our case), TMD denotes the target model (optimal CPN in our case), and MB is a set of mapping blocks that relate subsets of constructs in SMD to their equivalents in TMD. The ***Base of examples*** is a set of transformation examples. The transformation examples can be collected from different experts or by automated approaches.

Each TE is viewed as a set of blocks. A ***block*** defines a transformation trace between a subset of constructs in the source model and a subset of constructs in the target model. Constructs that should be transformed together are grouped into the same block. For example, a message *m* that is sent from an object $\underline{A}$ to an object $\underline{B}$ cannot be mapped independently from the mapping of $\underline{A}$ to $\underline{B}$. In our examples, blocks correspond to concrete traces left by experts when transforming models. They are not general rules as they involve concept instances (*e.g.,* a message *m*) instead of concepts (*e.g.,* message concept). In other words, where transformation rules are expressed in terms of meta-models, blocks are expressed in terms of concrete models.

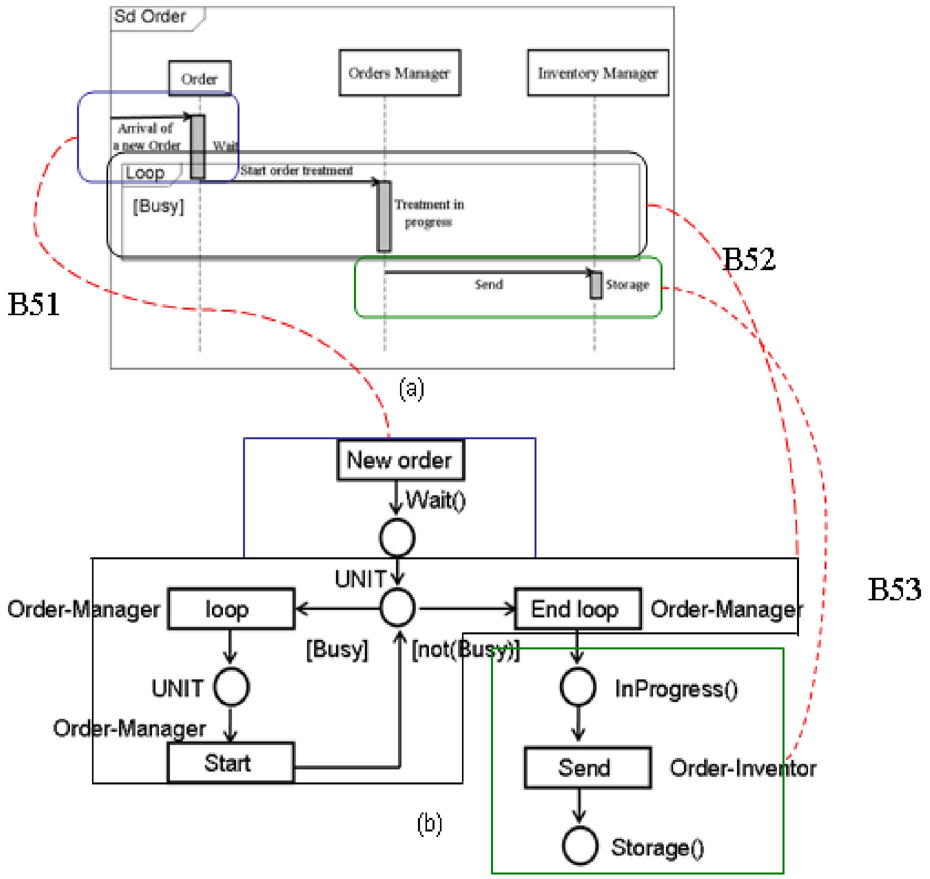


Fig. 1. (a) Example of SD (source model) and (b) his equivalent CPN (target model).

In a SD-to-CPN transformation, blocks correspond to transformation traces of loops (*loop*), alternatives (*alt*), concurrent interactions (*par*), *activation boxes*, and *messages* (see UML2.0 SD specification for more details about these constructs [7]). In the case where the constructs are imbedded, a single block is created for the higher-level construct. Blocks can be derived automatically from the transformation trace of the whole model.

An example of a SD-to-CPN transformation is presented in Figure 1. For legibility reasons, we present an example containing only one complex fragment *loop*. In the validation section, we will use more complex SDs that involve different CPN constructs. The SD in Figure 1.a contains 10 constructs that represent 3 objects, 3 messages, 1 loop and 3 activation boxes. Three blocks are defined[2]: B_{51} for message *Arrival of a new Order* and

[2] For traceability purpose, blocks are sequentially numbered. For instance, the 3 blocks of this example TE_i are B_{51} to B_{53}. Those of TE_{i+1} are B_{54} to B_{xx}, and so on and so forth. When a solution is produced, it is easy to determine which examples contributed to it.

activation box *Wait*, B_{52} for the loop with guard *[Busy]*, message *Start order treatment*, and activation box *Treatment in progress*, and B_{53} for message *Send* and activation box *Storage*. Notice that only one block is defined in B_{52} as the activation box is inside the loop.

In block B_{51}, for example, *Arrival of a new Order* was transformed by an expert into the transition *New order* and *Wait* into the place *Wait()* (Figure 1.b).

To manipulate them more conveniently, the models (source or target) are described by sets of predicates, each corresponding to a construct (or a sub-construct) type. The order of writing predicates is important in the case of a dynamic model. The predicate types for SDs are:

```
Object (ObjetName, ObjetType);
Message (MessageType, Sender, Receiver, MessageName,
 ActivityName);
Activity (ActivityName, ObjectName, Duration, MessageNumber);
Loop (StartMessageName, EndMessageName, ConditionValue);
Par (StartMessageName, EndMessageName, ConditionValue,
 ConditionType);
```

Similarly, those of CPN are:

```
Place (PlaceName);
Transition (TransitionName);
Input(TransitionName, PlaceSourceName)
Output(TransitionName, PlaceDestinationName)
```

For example, the message *Arrival of a new Order* in Figure 1.a can be described by

```
Message (Synchronic,_, Order, ArrivalOfNewOrder, Wait);
```

The predicate indicates that *Arrival of a new Order* is a synchronic message sent to Order (with “_” meaning no sender) and connected to activation box *Wait*. Mapping traces are also expressed using predicate correspondences with the symbol “:”. In Figure 1.b, for instance, block B_{51} is defined as follows:

```
Begin B51
Message (Synchronic, _, Order, ArrivalOfNewOrder, Wait). :
 Transition (NewOrder, Coulor1), Input(NewOrder, _),
 Output(NewOrder, Wait).
Activity (Wait, Order, 10, 2). :  Place (Wait).
End B51
```

In the absence of transformation rules, a construct can be transformed in many ways, each having a degree of relevance. A SD M_i to transform is characterized by its description

SMD$_i$, *i.e.,* a set of predicates. Figure 2 shows a source model with 6 constructs to transform represented by circles. A transformation solution consists of assigning to each construct a mapping block transformation possibility from the example base (blocks are represented by rectangles in Figure 2). A possibility is considered to be adequate if the block maps a similar construct.

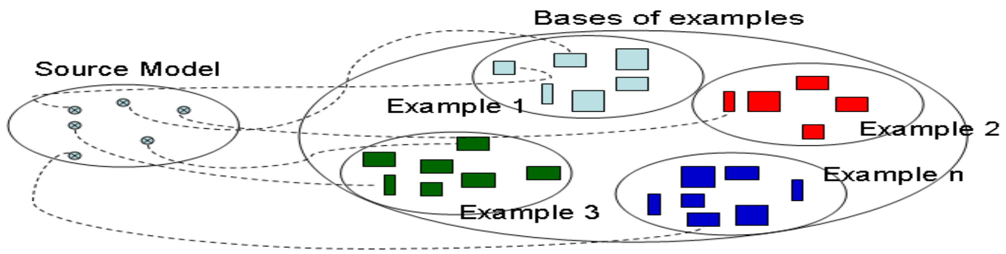


Fig. 2. Transformation solution as blocks-to-constructs mapping

2.2 Transformation Complexity

Our approach is similar to case-based reasoning [21] with the difference that we do not select and adapt the whole transformation of a similar SD. Instead, we combine and adapt fragments of transformations coming from the transformations of several SDs.

The transformation of a SD M_i with n constructs, using a set of examples that globally define m possibilities (blocks), consists of finding the subset from the m possibilities that better transforms each of the n constructs of M_i. In this context, m^n possible combinations have to be explored. This value can quickly become huge.

If we limit the possibilities for each construct to only blocks that contain similar constructs, the number of possibilities becomes $m_1 \times m_2 \times m_3 \times \ldots \times m_n$ where each $m_i \leq m$ represents the number of blocks containing constructs similar to construct i. Although the number of possibilities is reduced, it could still be very large for big SDs. A sequence diagram with 50 constructs, each having 8 or more mapping possibilities, necessitates exploring at least 8^{50} combinations. Considering these magnitudes, an exhaustive search cannot be used within a reasonable time frame. This motivates the use of a heuristic search when a more formal approach is either not available or hard to deploy.

3 Heuristic-based Transformation

We describe in this section the adaptation of two heuristics, PSO [18] and SA [19], to automate SD-to-CPN transformation. These methods each follow a generic strategy to explore the search space. When applied to a given problem, they must be specialized by defining: (1) the coding of solutions, (2) the operators that allow moving in the search space, and (3) the fitness function that measures the quality of a solution. In the remainder of this section we start by giving the principles of PSO and SA. Then, we describe the three above-mentioned heuristic components.

3.1 Principle

To obtain a more robust optimization technique, it is common to combine different search strategies in an attempt to compensate for deficiencies of the individual algorithms [20]. In our context the search for a solution is done in two steps. First, a global search with PSO is quickly performed to locate the portion of the search space where good solutions are likely to be found. In the second step, the obtained solution is refined with a local search performed by SA.

PSO, *Particle Swarm Optimization,* is a parallel population-based computation technique proposed by Kennedy and Eberhart [18]. The PSO swarm (population) is represented by a set of *K* 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 that produces a translation vector. The improvement is assessed by a fitness function.

The particle with the highest fitness is memorized as the global best solution (*gbest*) during the search process. Also, each particle stores its own best position (*pbest*) among all the positions reached during the search process. At each iteration, all particles are moved according to their velocities (Equation 1). The velocity $V_i^{'}$ of a particle *i*, depends on three factors: its inertia corresponding to the previous velocity, its *pbest*, and the *gbest*. Factors are weighted respectively by W, C_1, and C_2. The importance of the local and global position factors varies and is set at each iteration by a random function. The weight of inertia decreases during the search process. The derivation of $V_i^{'}$ is given by Equation 2. After each iteration, the individual *pbest*s and the *gbest* are updated if the new positions bring higher qualities than the ones before.

$$X_i' = X_i + V_i' \tag{1}$$

$$V_i' = W \times V_i + C_1 \times rand\,() \times (pbest_i - X_i) + C_2 \times rand\,() * (gbest - X_i) \tag{2}$$

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

Simulated Annealing (SA) [19] 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 gradually decreased. For each temperature, the following three steps are repeated for a fixed number of iterations: (1) determine a new neighboring solution; (2) evaluate the fitness of the new solution; (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 can still be accepted, but with a certain probability. The 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 more difficulty as the temperature decreases. This mechanism prevents reaching a local optimum.

3.2 Adaptation

To adapt PSO and SA to the SD-to-CPN transformation problem, we must define the following: a solution coding suitable for the transformation problem, a neighborhood function to derive new solutions, and a fitness function to evaluate these solutions.

As stated in Section 2, we model the search space as an *n*-dimensional space where each dimension corresponds to one of the *n* constructs of the SD to transform. A solution is then a point in that space, defined by a coordinate vector whose elements are blocks numbers from the example base assigned to the *n* constructs. For instance, the transformation of the SD model shown in Figure 3 will generate a 7-dimensional space that accounts for the two objects, three messages and two activities. One solution is this space, shown in Table 1, suggests that message *CheckDriver* should be transformed according to *block B_{19}*, activity

Positioning, according to *block* B_7, etc. Thus concretely, a solution is implemented as a vector where constructs are the dimensions (the elements) and block numbers are the element values.

The association between a construct and a block does not necessarily mean that a transformation is possible, *i.e.,* the block perfectly matches the contest of the construct. This is determined by the fitness function described in subsection 3.2.3.

The proposed coding is valid for both heuristics. In the case of PSO, as an initial population, we create *k* solution vectors with a random assignment of blocks. Alternatively, SA starts from the solution vector produced by PSO.

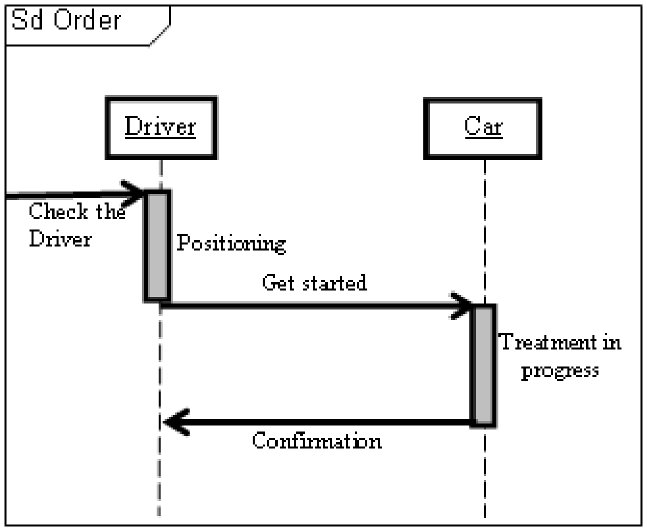


Fig.3. Example of source model

Table 3. Solution representation

Dimensions	**Constructs**	**Block numbers**
1	Message(CheckDriver)	B19
2	Activity(Positioning)	B7
3	Message(GetStarted)	B51
4	Activity(Treatment)	B105
5	Message(Confirmation)	B16
6	Object(Driver)	B83
7	Object(Car)	B33

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

In the case of PSO, a translation (velocity) vector is derived according to equation 2 and added to the position vector. For example, the solution of Table 1 may produce the new solution shown in Figure 4. The velocity vector *V* has a translation value for each element (real values). When summed with the block numbers, the results are rounded to integers. They are also bounded by 1 and the maximum number of available blocks.

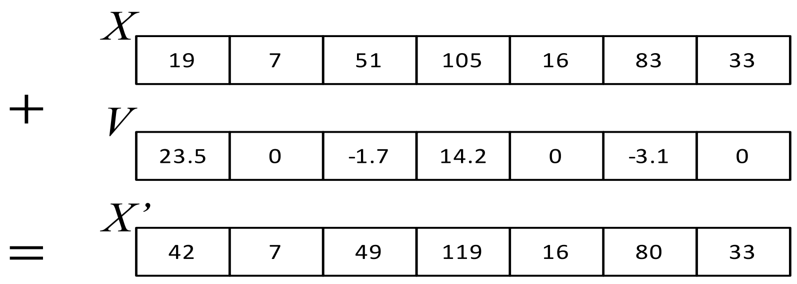


Fig. 4. Change Operator in PSO

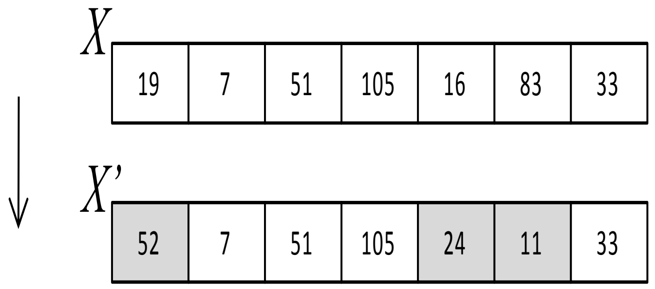


Fig. 5. Change Operator in SA

For SA, the change operator randomly chooses l dimensions ($l < n$) and replaces their assigned blocks by randomly selected ones from the example base. For instance, Figure 5 shows a new solution derived from the one of Table 1. Constructs 1, 5 and 6 are selected to be changed. They are assigned respectively blocks 52, 24, and 11 instead of 19, 16, and 83. The other constructs remain unchanged. The number of blocks to change is a parameter of SA (three in this example). In our validation, we set it to 4 considering that the average number of constructs per SD is 36.

Fitness Function. The fitness function allows quantifying the quality of a transformation solution. As explained in the previous paragraph, solutions are derived by random assignment of new blocks to some constructs. The quality of a transformation solution is then the sum of the individual transformation qualities of the n constructs of the SD. To evaluate if assigned block B_i is a good transformation possibility for construct C_j, the fitness function first evaluates the **adequacy**, *i.e.,* does B_i contains a construct C_k from the same type as C_j? if the answer is “no”, the assigned block is unsuitable. Otherwise, the fitness function checks the three following coherence aspects: (1) **internal coherence** (what is the degree of similarity between C_j and C_k in terms of properties?), (2) **external coherence** (to what extent the transformation proposed by B_i contradicts the transformations of constructs related to C_j?), and (3) **temporal coherence** (to what extent the transformation proposed by B_i preserves the temporal constraints of message sequences in SD?). The fitness function is formally defined as follows:

$$f = \sum_{j=1}^{n} a_j \times (ic_j + ec_j + tc_j) \qquad (3)$$

where a_j is the adequacy of assigning B_i to C_j (1 if B_i is adequate, 0 otherwise), and ic_j, ec_j, and tc_j are respectively the internal, external, and temporal coherences of the assignment. ic_j is defined as the ratio between the number of parameters of the predicate P_j representing C_j that match the parameters of the associated construct in block B_i and the total parameters of P_j.

Consider the SD example shown in Figure 3. Message *GetStarted* is defined by predicate `Message(Synchronic, Driver, Car, GetStrated, Positioning).` This predicate indicates that the message *GetStarted*, which is *synchronic*, is sent by object Driver to Car from the activity *Positioning*. The solution in Table 1 assigns the block B_{51} to this message. Block B_{51} is described in section 2.1 as follows:

```
Begin B51
Message (Synchronic, _, Order, ArrivalOfNewOrder, Wait). :
 Transition (NewOrder, Coulor1), Input(NewOrder, _),
 Output(NewOrder, Wait).
Activity (Wait, Order, 10, 2). :  Place (Wait).
End B51
```

The adequacy a_3 of the transformation of *GetStarted* (3rd construct) is equal to 1 because block B_{51} also contains predicate *message* (*ArrivalOfNewOrder*). The parameters of the two messages are similar except for the sender which is not an object in the case of *ArrivalOfNewOrder*. As a result, internal coherence ic_3=4/5=0.8 (four parameters that match over 5).

For external coherence ec_j, let $RCons_j$ be the set of constructs related to C_j and $RConsM_{ij}$, the subset of constructs in $RCons_j$ whose transformations are consistent with the one of C_j, *i.e.*, we compares the transformation proposed by the block assigned to C_j with the ones suggested by the blocks assigned to the related constructs. ec_j is calculated as the ratio between $RConsM_{ij}$ and $RCons_j$.

In our example, *GetStarted* involves three constructs (sender, receiver, and activity). According to B_{51}, only *Positioning* activity is related (has a predicate) and should be transformed into a place similarly to *Wait* activity. In the solution of , the construct *Positioning* is assigned the block B_7 (dimension 2 of the solution vector). This block is defined as follows:

```
Begin B7
Message (Asynchronic, User, Printer, NewPrint, Progress). :
 Transition (NewPrint, Coulor7), Input(NewPrint, _),
 Output(NewPrint, Progress).
Activities (Progress, Printer, 8, 1). :  Place (Progress).
End B7
```

According to B_7, *Positioning* should also be mapped to a place. Thus there is no conflict between B_{51} and B_7, and ec_3=1 (1/1).

tc_j represents the temporal coherence. It reflects the time constraint specific to dynamic models. To preserve the temporal coherence, we ensure that the transformation of elements that are contiguous to C_j preserve the temporal semantics. To this end, we first consider the block B_{inc} that includes C_j and the blocks B_{pre} and B_{fol} that respectively precedes and follows B_{inc}. Although the model to transform is not in the example base, we identify blocks with only the source part according to the rules given in Section 2.1. Then we consider the block B_i, assigned to C_j by the evaluated solution, and the two blocks B_{pre_i} and B_{fol_i} preceding and following B_i. tc_j is obtained by comparing B_{pre} to B_{pre_i}, B_{inc} to B_i, and B_{fol} to B_{fol_i}. For example, let $P_{pre}(k)$ be the predicate having the k^{th} position in B_{pre} and $P_{pre\_i}(k)$ be the predicate having the k^{th} position in B_{pre_i}, the number of pairs of predicates $PMatch(B_{pre}, B_{pre\_i})$ that match in the two blocks is defined as

$$\left|\left\{(P_{pre}(k), P_{pre\_i}(k)) \middle| P_{pre}(k) = P_{pre\_i}(k)\right\}\right| \qquad (4)$$

tc_i is then defined as follows:

$$tc_j = \frac{|PMatch(B_{pre}, B_{pre\_i})| + |PMatch(B_{inc}, B_i)| + |PMatch(B_{fol}, B_{fol\_i})|}{\max(|B_{pre}|, |B_{pre\_i}|) + \max(|B_{inc}|, |B_i|) + \max(|B_{fol}|, |B_{fol\_i}|)} \qquad (5)$$

Figure 6 shows an example of the calculation of tc_j. Going back to the example of message *GetStarted*, to derive the tc_3, we identify in the SD to transform two blocks: B_s which contains *GetStarted* and B_{s-1} which precedes B_s. Consequently, block B_{51} will be compared to B_s. B_s contains a message followed by an activity and another message. B_{51} contains a message followed by an activity. Then, two pairs of predicates match and the max size between the two blocks is 3. As B_{51} has no preceding block, we consider that no match exists with B_{s-1}, and the corresponding max size is that of B_{s-1}, *i.e.*, 2 for the message and the activity. Finally, as B_s has no following block, no match exists with B_{52}, which follows B_{51}. We take then as max size, the size of B_{52} (3 corresponding to the loop, the message, and the activity). According to equation 5, tc_3=(0+2+0)/(2+3+3)=0.25. This temporal coherence factor is standard and works with any combined fragments of SDs.

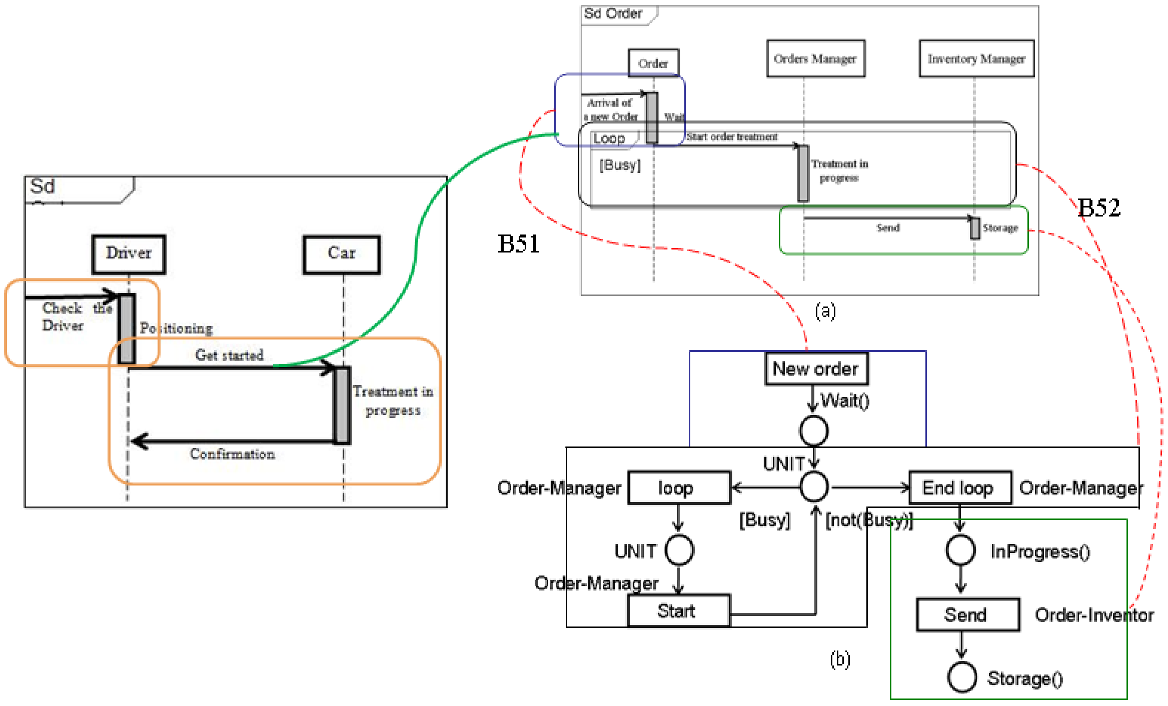


Fig. 6. Temporal coherence

The fitness function does not need a considerable effort to be adapted for other transformations (e.g. state machine to PNs). However, the block definition must be adapted to the semantics of the new transformation.

4 Validation

To evaluate the feasibility of our approach, we conducted an experiment on the transformation of 10 UML sequence diagrams[3]. We collected the transformations of these 10 sequence diagrams from the Internet and textbooks and used them to build an example base EB = {<SD_i, CPN_i> | i=1,2,...,10}. We ensured by manual inspection that all the transformations are valid. The size of the SDs varied from 16 to 57 constructs, with an average of 36. Altogether, the 10 examples defined 224 mapping blocks. The 10 sequence diagrams contained many complex fragments: *loop*, *alt*, *opt*, *par*, *region*, *neg* and *ref*.

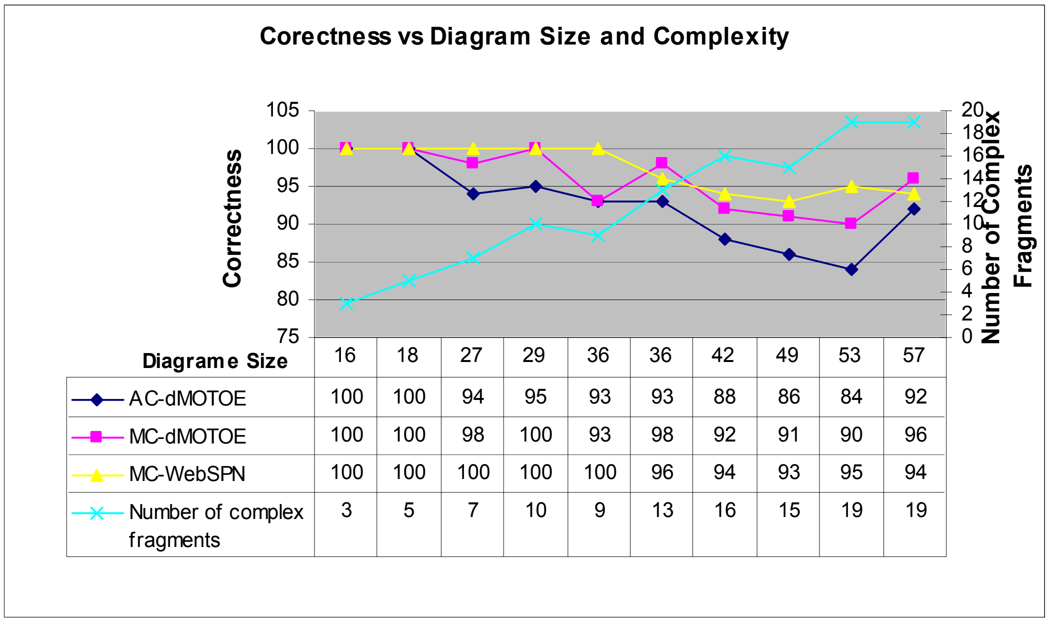
To evaluate the correctness of our transformation method, we used a 10-fold cross validation procedure. For each fold, one sequence diagram SD_j is transformed using the remaining 9 transformation examples. Then, the transformation result for each fold is checked for correctness using two methods: automatic correctness (*AC*) and manual correctness (*MC*). Automatic correctness consists of comparing the derived CPN to the known CPN, construct by construct. This measure has the advantage of being automatic and objective. However, since a given SD_j may have different transformation possibilities, *AC* could reject a valid construct transformation because it yields a different CPN from the one provided. To prevent this situation, we also perform manual evaluation of the obtained CPN. In both cases, the correctness is the proportion of constructs that are correctly transformed.

In addition to correctness, we compare the size of the obtained CPNs with the ones obtained by using the rule-based tool WebSPN for mapping UML diagrams to CPN [16]. The size of a CPN is defined by the number of constructs.

Figure 7 shows the correctness for the 10 folds. Both automatic and manual correctness had values greater than 90% in average (92.5% for *AC* and 95.8% for *MC*). Although few examples were used (9 for each transformation), all the SDs had a transformation correctness greater than 90%, with 3 of them perfectly transformed.

[3] The reader can find in this link www.marouane-kessentini.com/ecmfa2010 all the materials used in our experiments

Figure 7 also shows that, in general, the best transformations are obtained with smaller SDs. After 36 constructs, the quality degrades slightly but steadily. This may indicate that the transformation correctness of complex SDs necessitates more examples in general. However, the largest and most complex SD (57 constructs and 19 complex fragments) has a *MC* value of 96%.



Diagrame Size	16	18	27	29	36	36	42	49	53	57
◆ AC-dMOTOE	100	100	94	95	93	93	88	86	84	92
■ MC-dMOTOE	100	100	98	100	93	98	92	91	90	96
▲ MC-WebSPN	100	100	100	100	100	96	94	93	95	94
✕ Number of complex fragments	3	5	7	10	9	13	16	15	19	19

Fig. 7. Correctness of the transformations

In addition, our results show that the correctness of our transformations is equivalent that of WebSPN. Another interesting finding during the evaluation is that, in some cases, a higher fitness value does not necessarily imply higher transformation correctness. This was the case for the transformations of *SD_3* (fitness of 82% and *MC* = 98%) and *SD_5* (fitness of 92% and *MC* = 93%). This is probably due to the fact that we assign the same weight to simple constructs such as messages and complex constructs such as loops in the fitness function. Indeed, temporal coherence is more difficult to assess for complex constructs.

Manual inspection of the generated CPNs showed that the different transformation errors are easy to fix. They do not require considerable effort and the majority of them is related to the composition of complex fragments. For example, as we did not have an example that mapped two *alts* situated in a *loop*, the optimization technique used one that contained only one *alt* in a *loop*. Almost the same errors were made by WebSPN, including the case of two *alts* in a *loop*.

When developing our approach, we conjectured that the example-based transformation produce CPNs smaller than the one obtained by systematic rule application. Table 2 compares the obtained CPN sizes by using dMOTOE and WebSPN for the 10 transformations. In all cases, a reduction in size occurs when using dMOTOE, with an average reduction of 28.3% in comparison to WebSPN. Although the highest reduction corresponded to the smallest SD, the reductions for larger diagrams were important as well (*e.g.,* 39% for 36 constructs, 38% for 39 constructs, and 29% for 76 constructs). These reductions should be viewed in the context of the correctness equivalence between our approach and WebSPN.

Table 2. CPN size comparison

Size(WebSPN)	Size(dMOTOE)	Variation
22	13	41%
36	22	39%
39	24	38%
43	31	28%
51	36	30%
50	39	22%
56	39	30%
53	44	16%
58	52	10%
76	54	29%
Average Reduction :		28.3%

The obtained results confirm our assumption that systematic application of rules results in CPNs larger than needed and that reusing valid transformed examples attenuates the state explosion problem.

As for execution time, we ran our algorithm on a standard desktop computer (Pentium CPU running at 2 GHz with 2 GB of RAM). The execution time was of the order of a few seconds and increased linearly with the number of constructs, indicating that our approach is scalable from the performance standpoint.

5 Related Work

The work proposed in this paper crosscuts two main research areas: model transformation and traceability in the context of MDD.

In [5], five categories of transformation approaches were identified: graph-based [22], relational [23], structure-driven [24], direct-manipulation, and hybrid. One conclusion to be drawn from studying the existing MT approaches is that they are often based on empirically obtained rules [25].

Recently, traceability gained popularity in model transformation [26]. Usually, trace information are generated manually and stored as models [27]. For example, Marvie et al. [28] propose a transformation composition framework that allows the manual creation of linkings (traces). In the studied approaches and frameworks based on traceability, trace information is used in general for detecting model inconsistency and fault localization in transformations. On the other hand, dMOTO uses traces to automate the transformation process.

More specifically, in the case of SD-to-PN, several approaches were proposed in addition to WebSPN. In [29], the authors describe a meta-model for the SD-to-PN mapping. It defines rules involving concepts of the meta-models representing respectively sequence diagrams and Petri nets. One of the limitations of this approach is that temporal coherence is not addressed explicitly. Additionally, the meta-model representing the rules tends to generate large PNs, as noticed by the authors. In [11], a set of rules to transform UML 2.0 SDs into PNs is proposed. The goal is to animate SDs using the operational semantics of PNs. In our case, we can generate the structure of the targeted CPN in an XMI file that can be used as input for some simulation tools like CPN tools [38]. Other UML dynamic diagrams are also considered for the transformation to PNs. For example, use case constructs are mapped to PN using a multi-layer technique [8].

There are other research contributions that concentrate on supporting validation and analysis of UML statecharts by mapping them to Petri nets of various types [36, 37]. Unlike our approach, this work uses information extracted from different UML diagrams to produce the Petri nets. A general conclusion on the transformation of dynamic models to PNs is that, in addition to the fact that no consensual transformation rules are used, a second step is usually required to reduce the size of the obtained PNs.

dMOTOE uses the “by example” principle to transform models, but what we propose is completely different from other contributions to model transformation by example (MTBE). Varro and Balogh [12, 13] propose a semi-automated process for MTBE using Inductive Logic Programming (ILP). The principle of their approach is to derive transformation rules semi-automatically from an initial prototypical set of interrelated source and target models. Wimmer et al. [30] derive ATL transformation rules from examples of business process models. Both works use semantic correspondences between models to derive rules, and only static models are considered. Moreover, in practice, a large number of transformation learning-examples may be required to ensure that the generated rules are complete and correct. Both approaches provide a semi-automatic generation of model transformation rules that needs further refinement by the user. Also, since both approaches are based on semantic mappings, they are more appropriate in the context of exogenous model transformations between different metamodel. Unfortunately, the generation of rules to transform attributes is not well supported in most MTBE implementations. Our model is different from both previous approaches to MTBE. We do not create transformation rules to transform a source model, directly using examples instead. As a result, our approach is independent from any source or target metamodels.Recently, a similar approach to MTBE, called Model Transformation By Demonstration (MTBD), is proposed [34]. Instead of the MTBE idea of inferring the rules from a prototypical set of mappings, 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. This approach needs a large number of simulated patterns to give good results and, for instance, MTBD cannot be useful to transform an entire source model.

6 Conclusion

In this paper, we propose the approach dMOTOE, to automate SD-to-CPN transformation using heuristic search. dMOTOE uses a set of existing transformation examples to derive a colored Petri net from a sequence diagram. The transformation is seen as an optimization problem where different transformation possibilities are evaluated and, for each possibility, a quality is associated depending on its conformance with the examples at hand.

The approach we propose has the advantage that, for any source model, it can be used when rules generation is difficult. Another interesting advantage is that our approach is independent from source and target formalisms; aside from the examples, no extra information is needed. Moreover, as we reuse existing transformations, the obtained CPN are smaller than those obtained by transformation rules.

We have evaluated our approach on ten sequence diagrams. The experimental results indicate that the derived CPNs are comparable to those defined by experts in terms of correctness (average value of 96%). Our results also reveal that the generated CPNs are smaller than the ones generated by the tool WebSPN [16].

Although, the obtained results are very encouraging, many aspects of our approach could be improved. Our approach currently suffers from the following limitations: 1) in the case of SD-to-PNs transformation, it provides less clean semantics than a rules-based approach; 2) coverage of complex fragments examples is needed for completeness and to ensure consistently good results; 3) the base of examples is difficult to collect especially for complex and not widely used formalisms; 4) the fitness function could weight complex constructs more heavily when evaluating a solution. In addition, a validation on a larger example base is in project to better assess the adaptation capability of the approach, and we can compare the sizes of the reachability graph of the produced CPNs by dMOTOE and WebSPN in order to treat the richer behaviors (in fact, a bigger net is not necessarily worse in some cases). In a broader perspective, we plan to experiment and extend dMOTOE to other transformations involving dynamic models: code generation (model-to-code), refactoring (code-to-code), or reverse-engineering (code-to-model).

REFERENCES

1. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. ICSE 2007 : Future of Software Engineering. (2007)
2. Jouault, F., Kurter, I.: Transforming models with ATL. In: Proc. Of the Model Transformations in Practice Workshop at MoDELS 2005, Jamaica (2005)
3. Muller, P., F.Fleurey, et J. M. Jezequel (2005). Weaving Executability into Object-Oriented Meta-languages. Proc. of MoDELS’05, Montego Bay, Jamaica, pp 264-278.
4. Varro, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. UML 2004. LNCS, vol. 3273. Springer, Heidelberg (2004)
5. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOSPLA 2003, Anaheim, USA (2003)
6. Ehrig, Hartmut; Ehrig, Karsten; de Lara, Juan; Taentzer, Gabriele; Varró, Dániel; Varró-Gyapay, Szilvia: Termination Criteria for Model Transformation. *Vol. 3442FASE 2005*
7. Booch, Grady, Jacobson, Ivar and Rumbaugh,James: "The Unified Modeling Language Users Guide", Addison Wesley 1998.
8. J. Saldhana and S. M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. *SEKE*, pages 103–110, July 2000.
9. J. Lilius and I. P. Paltor. vUML: A Tool for Verifying UML Models. ASE99.
10. T. Murata. Petri Nets: Properties, Analysis, and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
11. Ribeiro OR, Fernandes JM; Some Rules to Transform Sequence Diagrams into Coloured Petri Nets, CPN2006, Jensen K (ed.), Aarhus, Denmark, pp. 237-56, Oct/2006.
12. D. Varro. Model transformation by example. In Proc.MODELS 2006, pp. 410–424.
13. D. Varro and Z. Balogh, Automating Model Transformation by Example Using Inductive Logic Programming, ACM Symposium, 2007 (SAC 2007).
14. M. Kessentini, H.Sahraoui and M.Boukadoum Model Transformation as an Optimization Problem. In Proc.MODELS 2008, pp. 159-173 Vol. 5301 of LNCS. Springer, 2008.
15. M. Kessentini, H.Sahraoui and M.Boukadoum, Search-based Model Transformation by Example. Submitted to SoSym (under review)
16. Salvatore Distefano, Marco Scarpa, Antonio Puliafito: Software Performance Analysis in UML Models. FIRB-Perf 2005: 115-125 (https://mdslab.unime.it/webspn/mapping.htm)
17. K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Volume 1, Basic Concepts. Monographs in Theoretical Computer Science , 1997.
18. J. Kennedy, and R.C Eberhart : Particle swarm optimization. In: Proc. IEEE Intl.Conf. on Neural Networks, pp. 1942–1948 (1995)
19. D.S. Kirkpatrick, Jr. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671-680, 1983.
20. Kelner, V., Capitanescu, F., Léonard, O., and Wehenkel, L. 2008 A hybrid optimization technique coupling an evolutionary and a local search algorithm. *J. Comput. Appl. Math.*
21. A.Aamodt and E.Plaza, "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches," *AIC* (1994), 39-52.
22. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph Transformation for Specification and Programming. Technical Report 7/96, Universität Bremen, 1996

23. D. H. Akehurst and S.Kent. A Relational Approach to Defining Transformations in a Metamodel. *UML 2002* Proceedings, LNCS 2460, 243-258, 2002
24. Interactive Objects Software GmbH, Project Technology, Inc. MOF 2.0 Query/Views/Transformations RFP, Revised Submission.
25. Egyed, A.: Heterogeneous Views Integration and its Automation, Ph.D. Thesis (2000)
26. I. Galvão and A. Goknil, "Survey of Traceability Approaches in Model-Driven Engineering". *EDOC'07,* pages 313-326,
27. Jouault, F.: Loosely coupled traceability for atl. In: (ECMDA). (2005)
28. Marvie, R.: A transformation composition framework for model driven engineering. Technical Report LIFL-2004-10, LIFL (2004)
29. Adel Ouardani, Philippe Esteban, Mario Paludetto, Jean-Claude Pascal, "A Meta-modeling Approach for Sequence Diagrams to Petri Nets Transformation", *ESMC* 2006.
30. M. Wimmer, M. Strommer, H. Kargl, and G. Kramler.Towards model transformation generation by-example. HICSS-40 Hawaii International Conference on System Sciences.
31. M. Harman and B. F. Jones, Search-based software engineering, *Information & Software Technology*, Vol. 43, No. 14, pp. 833-839 (2001).
32. O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. *GECCO '06*, pages 1909-1916..
33. M. Harman, The Current State and Future of Search Based Software Engineering, In *Proceedings of ICSE 2007*, 20-26 May, Minneapolis, USA (2007)
34. Yu Sun, Jules White, and Jeff Gray, "Model Transformation by Demonstration," *MoDELS09*, Denver, CO, October 2009, pp. 712-726.
35. Uzam, M. The use of Petri net reduction approach for an optimal deadlock prevention policy for flexible manufacturing systems. *Int. J. Adv. Manuf. Technol.*, 23, 204–219.
36. Z. Hu and S. M. Shatz, "Mapping UML Diagrams to a Petri Net Notation for System Simulation", (SEKE), Banff, Canada, June 2004, pp. 213-219
37. S. Bernardi, S. Donatelli, J. Merseguer, From UML Sequence Diagrams and StateCharts to analysable Petri Net models, WOSP02, pages 35-45, Rome (Italy), July 2002.
38. http://wiki.daimi.au.dk/cpntools/cpntools.wiki

Part 2: Endogenous Transformation by Example

In part 1 of this thesis, we described our contributions to exogenous transformation. In this part, we focus on endogenous transformations. In an endogenous transformation, the source and target meta-models are the same. Furthermore, they require two steps: 1) identify the elements to transform in the source model; 2) transform the identified elements. Endogenous transformations are principally related to the following maintenance activities: 1) Optimization (a transformation aimed to improve certain operational qualities (e.g., performance), while preserving the semantics of the software); 2) Refactoring (a change to the internal structure of software to improve certain software quality characteristics such as understandability, modifiability, reusability, modularity, adaptability) without changing its observable behaviour.

In this thesis, we focus on code transformation in order to improve quality. We distinguish two steps for this task: 1) detecting refactoring opportunities that correspond to design defects; 2) applying some refactoring methods (move method, add super class, etc) to modify the defected classes. The second step is out of the scope of this work and we will only address the first one.

The first step related to detecting design defects is important. In fact, detecting and fixing defects is a difficult, time-consuming, and to some extent, manual process. The number of outstanding software defects typically exceeds the resources available to address them. In many cases, mature software projects are forced to ship with both known and unknown defects for lack of the development resources to deal with everyone. For example, one Mozilla developer claimed that "everyday, almost 300 bugs and defects appear . . . far too much for only the Mozilla programmers to handle" [53]. To help cope with this magnitude of activity, several automated detection techniques have been proposed [27].

Although there is a consensus that it is necessary to detect design anomalies, our experience with industrial partners has shown that there are many open issues that need to be addressed when developing a detection tool. Design anomalies have definitions at different levels of abstraction. Some of them are defined in terms of code structure, others in terms of developer/designer intentions, or in terms of code evolution. These definitions are in many cases ambiguous and incomplete. However, they have to be mapped into rigorous and deterministic rules to make the detection effective.

In the following, we discuss some of the open issues related to the detection.

How to decide if a defect candidate is an actual defect? Unlike software bugs, there is no general consensus on how to decide if a particular design violates a quality heuristic. There is a difference between detecting symptoms and asserting that the detected situation is an actual defect.

Are long lists of defect candidates really useful? Detecting dozens of defect occurrences in a system is not always helpful. In addition to the presence of false positives that may create a rejection reaction from development teams, the process of using the detected lists, understanding the defect candidates, selecting the true positives, and correcting them is long, expensive, and not always profitable.

What are the boundaries? There is a general agreement on extreme manifestations of design defects. For example, consider an OO program with a hundred classes from which one implements all the behavior and all the others are only classes with attributes and accessors. There is no doubt that we are in presence of a Blob. Unfortunately, in real industrial systems, we can find many large classes, each one using some data classes and some regular classes. Deciding which ones are Blob candidates depends heavily on the interpretation of each analyst.

How to define thresholds when dealing with quantitative information? For example, Blob detection involves information such as class size. Although, we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given program/community of users could be considered average in another.

How to deal with the context? In some contexts, an apparent violation of a design principle is considered as a consensual practice. For example, a class Log responsible for maintaining a log of events in a program, used by a large number of classes, is a common and acceptable practice. However, from a strict defect definition, it can be considered as a class with abnormally large coupling.

In addition to these issues, the process of defining rules manually is complex, time-consuming and error-prone. Indeed, the list of all possible defect types can be very large and each type requires specific rules.

To address or circumvent the above mentioned issues, we propose two different automated detection approaches that are completely different from the state of art.

For the first one, instead of characterizing each symptom of each possible defect type, we apply the principle of negative selection, the process used by biological immune systems to identify antigens. An immune system does not try to detect specific bacteria and viruses. Rather, it starts by detecting what is abnormal, i.e., what is different from the healthy cells of the body. The more something is different, the more it is considered risky.

For the second approach, we propose a solution that uses knowledge from previously manually inspected projects, called defects examples, in order to detect design defects that will serve to generate new detection rules based on combinations of software quality metrics. In short, the detection rules are automatically derived by an optimization process, based on the Harmony search algorithm [56] that exploits the available examples.

In the next chapter, we provide the details of our proposal for automating design defects detection based on the immune system metaphor.

Chapter 5: An Immune-Inspired Approach for Design Defects Detection

5.1 Introduction

We describe our solution to the problem of design defects detection. This problem is considered as an endogenous transformation problem where, as mentioned earlier, two steps are needed. We focus on the first step that consists of finding the elements to transform. Our solution is based on the use of well-designed code examples and on considering each deviation from these examples as risky. This mechanism corresponds to the immune system process where foreign substances are detected via deviations from normal cell behaviour. This contribution was accepted for publication in the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010) [82]. The paper, entitled "Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code", is presented in the next section.

5.2 Design Defects Detection by Example: An Immune System Metaphor

Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code

Marouane Kessentini, Stéphane Vaucher, Houari Sahraoui
DIRO. Université de Montréal. CANADA

ABSTRACT

We propose an approach for the automatic detection of potential design defects in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. Taking inspiration from artificial immune systems, we generated a set of detectors that characterize different ways that a code can diverge from good practices. We then used these detectors to measure how far code in assessed systems deviates from normality. We evaluated our approach by finding potential defects in two open-source systems (Xerces-J and Gantt). We used the library JHotDraw as the code base representing good design/programming practices. In both systems, we found that 90% of the riskiest classes were defects, a precision far superiour to state of the art rule-based approaches.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance—*Restructuring, reverse engineering, and reengineering*

General Terms

Design

Keywords

Maintenance, design defects, artificial immune systems

1. INTRODUCTION

In order to limit maintenance costs and improve the quality of their software systems, companies try to both enforce good design/development practices and prevent bad practices. As a result, these practices have been studied by professionals and researchers alike with a special attention given to design-level problems.

There has been much research focusing on the study of bad design practices sometimes called defects, antipatterns [9], smells [1], or anomalies [2] in the literature [1]. Although these bad practices are sometimes unavoidable, in most cases, development teams should try to prevent them and remove them from their code base as early as possible. Hence, many fully-automated detection techniques have been proposed [3, 4, 5].

Several problems limit the effectiveness of existing techniques. Indeed, the vast majority of existing work relies on rule-based detection [6, 4]. Different rules identify key symptoms that characterize a defect using combinations of mainly quantitative (metrics), structural, and/or lexical information. Therefore, to identify and remove defects in a system, all possible defects should be known and their symptoms characterized with rules. Moreover, the rules must to be applied equally to any system in any context. This is not reasonable considering the variety of software systems and the difficulty of expressing some types of symptoms. These difficulties explain a large portion of the high false-positive rates mentioned in existing research [5].

In this article, we propose an automated detection approach that is completely different from the state of art. Instead of characterizing each symptom of each possible defect type, we apply the principle of negative selection, the process used by biological immune systems [7] to identify antigens. An immune system does not try to detect specific bacteria and viruses. Rather, it starts by detecting what is abnormal, *i.e.*, what is different from the healthy cells of the body. The more something is different, the more it is considered risky.

We apply the same principle to the detection of design defects by, first, defining what is normal. Normality is defined using a code base containing examples of well designed and implemented software elements. Then, we create a set of detectors that represent different ways that a code can diverge from the good code. Finally, elements of assessed systems that are similar to detectors are considered as risky.

To evaluate our approach, we used classes from the JHotDraw library as our examples of well-designed and implemented code. Two systems, Xerces-J and Gantt, were then analyzed using our approach. Almost all the identified riskiest classes (with levels > 90%) were found in a list of classes tagged as defects (blobs, spaghetti code and functional decomposition) in another project [4].

Our contributions to automation are as follows. First, our technique is fully automatable from the creation of detectors to the evaluation of classes. Second, our technique

[1]In the remainder of this article, we use the generic term *defect* to refer to an occurrence of a bad practice in the code

does not require an expert to write rules for every defect type, and adapt them to different systems. Finally, using only standard algorithms to measure similarity, our technique not only outperforms rule-based techniques in terms of precision, but we are also able to find a good mix of defects types. The major limitation of the approach is that we require a code base representing of good design practices. Our results indicate however that JHotdraw seems to be usable and could serve as a starting point for a company wishing to use our approach..

The remainder of this paper is structured as follows. Section 2 is dedicated to the problem statement. In Section 3, we describe the principles of the Artificial Immune System that inspires our approach and the adaptations of these principles to the detection of design defects. Section 4 presents and discusses the validation results. A summary of the related work in defect detection is given in Section 5. We conclude and suggest future research directions in Section 6.

2. PROBLEM STATEMENT

In this section, we describe the problem of defect detection. We start by defining important concepts. Then, we detail the specific problems that are addressed by our approach.

2.1 Basic Concepts

Design defects, also called **design anomalies** refer to design situations that adversely affect the development of a software. In general, they make a system difficult to change which may in turn introduce bugs.

Different types of defects presenting a variety of symptoms have been studied with the intent of improving their detection [8] and suggesting improvements paths. The two following types of defects are commonly mentioned. In [1], Beck defines 22 sets of symptoms of common defects, named **code smells**. These include large classes, feature envy, long parameter lists, and lazy classes. Each defect type is accompanied by some refactoring suggestions to remove them. Brown *et al.* [9] define another category of design defects named **anti-patterns**, which includes blob classes, spaghetti code, and cut & paste programming. In both books, the authors focus on describing the symptoms to look for in order to identify specific defects.

Regarding our detection approach, we use the following concepts:

- A **code fragment** represents a software element that is evaluated. This could be a class, method, or package in an object-oriented code. Although our approach could be applied to evaluate any of these entities, in this paper, we use code fragment to refer essentially to a class.

- A **design risk** is a code fragment that is dissimilar (unusual) from known good code. It could be a design defect or simply an unusual design/developpement practice.

- The process of **discovering design defects** consists of finding high-risk code fragments in the system without relying on specific knowledge on the known defect types.

2.2 Problem Statement

Any technique to detect design defect should address/ circumvent many difficulties inherent to the nature of defects. Here is the description of the most important difficulties and how they affect an automation process.

- There is **no exhaustive list** of all possible types of design defects. Although there has been significant work to classify defect types [8, 10, 11], programming practices, paradigms and languages evolve making unrealistic to support the detection of all possible defect types. Furthermore, there might be company or application-specific (bad) design practices.

- For those design defects that are documented, there is *no consensual definition of symptom detections.* Defects are generally described using natural language and their detection relies on the interpretation of the developers. This limits the automation of the detection.

- The majority of detection methods do not provide **an efficient manner to guide the manual inspection of the candidate list**. Potential defects are generally not listed in an order that helps developers addressing in priority the most severe ones. There is little work, such as the one of Khomh *et al* [3], where probabilities are used to order the results.

3. AIS-BASED DETECTION ALGORITHM

Our approach is based on the metaphor of biological immune systems. In this section, we present the principles of this metaphor, and our adaptation to the problem of detecting design defects.

3.1 Principles of Artificial Immune Systems

The role of a biological immune system (IS) is to protect its host organism against foreign elements such as pathogens (*e.g.,* bacteria and viruses) and/or malfunctioning cells (*e.g.,* cancerous cells). This is performed following three phases: (1) *discovery*, (2) *identification*, and (3) *elimination* of foreign elements. *Discovery* is the phase that interests us in particular for our work. Therefore, we explain its principle in the following paragraphs.

There is no central organ that fully controls the IS. Instead, *detectors* wander in the body searching for harmful elements. Any element that can be recognised by the immune system is called an *antigen*. The cells that originally belong to our body and are harmless to its functioning are termed *self* (for self antigens) while the disease causing elements are named *nonself* (for nonself antigens). The IS classifies cells that are present in the body as self and nonself cells.

The immune system produces a large number of randomly created detectors. A negative selection mechanism eliminates detectors that match cells present in a protected environment (bone marrow and the thymus) where only self cells are assumed to be present. Non-eliminated ones become naive detectors; they die after some time unless they match an element assumed to be a pathogen. Detectors that do match a pathogen are quickly cloned; this is used to accelerate the response to future attacks. Since the clones are not exact replicates (they are mutated), this provides a more focused response to pathogens. This process, called *affinity maturation*, provides an efficient adaptation to a changing non-self environment. A detailed presentation of the biological immune system can be found in books such as [12]).

The success of immune systems at keeping a living organism healthy inspired the emergence of artificial immune systems (AIS) as a generic solution to problems in several domains, such as scheduling, computer security, optimization, or robotics [13]. AIS can be adapted to the problem of defect detection. The following mappings shows the similarity between our problem and the AIS concepts.

- **Body**: the evaluated system, more precisely, its code;
- **Detector**: an artificial code fragment that is very different from a well-designed code base;
- **Self Cells**: well-designed code fragments in the system to evaluate (without design defects);
- **Non-Self Cells**: code fragments in the system to evaluate that present a risk of being design defects;
- **Affinity**: the similarity between detectors and code fragments to evaluate.

3.2 Approach Overview

Figure 1 gives an overview of our approach. The detection process has two main steps: detector generation and risk estimation. Detectors are generated from a collection of code fragments coming from one or more well-designed systems. These code fragments define the reference of what is considered normal code. The generation process of detectors is performed using a heuristic search that maximizes on one hand, the distance between detectors and normal code and, on the other hand, the distance between the detectors themselves. The same set of detectors could be used to evaluate many systems, and it could be updated as the *normal* code base or development practices evolves.

The second step of the detection process consists of comparing the code to evaluate to the detectors. A code fragment that exhibits a similarity with a detector is considered as a risky element. The higher the similarity, the more a code fragment is considered risky. Both the detector generation and risk estimation steps use similarity scores. Before detailing the two steps, we first describe the similarity functions used in this work.

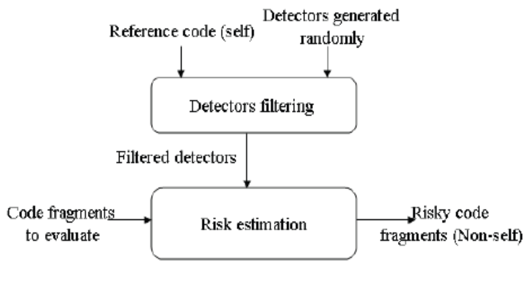
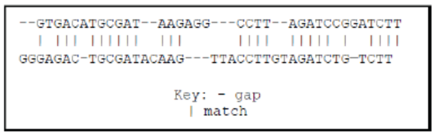

Reference code (self)
Detectors generated randomly
Detectors filtering
Filtered detectors
Code fragments to evaluate
Risk estimation
Risky code fragments (Non-self)

Figure 1: Approach Overview

3.3 Similarity between Code Fragments

To calculate the similarity between two code fragments, we adapted the Needleman-Wunsch alignment [14] algorithm to our context. It is a dynamic programing algorithm used in bioinformatics to efficiently find similar regions between two sequences of DNA, RNA or protein [15]. An example of the algorithm is presented in Figure 2.



```
--GTGACATGCGAT--AAGAGG---CCTT--AGATCCGGATCTT
  | ||| ||||||  |||       ||||  ||||| |  ||||
GGGAGAC-TGCGATACAAG---TTACCTTGTAGATCTG-TCTT

                Key: - gap
                  | match
```

Figure 2: Global alignment of two strings

As we are manipulating code elements and not sequences (strings), we represent these elements by sets of predicates. Each predicate type corresponds to a construct type of an object-oriented system: Class (C), attribute (A) , method (M), parameter (P), generalization (G), and method invocation relationship between classes (R). For example, in Figure 3, the sequence of predicates CGAAMPPM corresponds to a class with a generalization link, containing two attributes and two methods. The first method has two parameters.

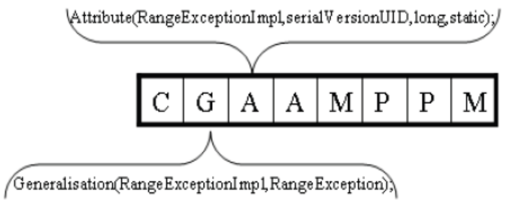

Attribute(RangeExceptionImpl,serialVersionUID,long,static);
C G A A M P P M
Generalisation(RangeExceptionImpl,RangeException);

Figure 3: Encoding

Predicates include details about the associated constructs (visibility, types, *etc.*). These details (thereafter called parameters) determine ways a code fragment can deviate from a notion of normality. The example of Figure 3 is a representation of class *RangeExceptionImpl* from Xerces-J. The corresponding predicate set, extracted using our inhouse eclipse plugin is as follows:

```
Class(RangeExceptionImpl,public);
Generalisation(RangeExceptionImpl,RangeException);
Attribute(RangeExceptionImpl,serialVersionUID,long,static);
Attribute(RangeExceptionImpl,serialImplUID,short,static);
Method(RangeExceptionImpl,RangeExceptionImpl,void,Y,public);
Parameter(RangeExceptionImpl,RangeExceptionImpl,code,short);
Parameter(RangeExceptionImpl,RangeExceptionImpl,message,
String);
Method(RangeExceptionImpl,implSerial,void,Y,private);
```

As described below, the Needleman-Wunsch global alignment algorithm [14] is described recursively. . When aligning two sequences (a_1,...,a_n) and (b_1,...,b_m). Each position $s_{i,j}$ in the matrix corresponds to the best score of alignment considering the previously aligned elements of the sequences. The algorithm can introduce gaps (represented by ”-”) to improve the matching of subsequences.

$$s_{i,j} = Max \begin{cases} s_{i-1,j} - g & \text{// insert gap for } b_j \\ s_{i,j-1} - g & \text{// insert gap for } a_i \\ s_{i-1,j-1} + sim_{i,j} & \text{// match} \end{cases}$$

where $s_{i,0} = g * i$ and $s_{0,j} = g * j$

At any given point, algorithm considers two possibilities. First, it considers the case when a gap should be inserted. When a gap is inserted for either a or b, the algorithm applies a penalty of g. Second, it tries to match predicates. The similarity function $sim_{i,j}$ returns the reward or cost of matching a_i to b_j. The final similarity is contained in $s_{n,m}$.

Our adaptation of the algorithm is straightforward. We define the gap penalty g and the similary function to match individual predicates (sim). We do not seek perfect matches in terms of number of predicates. A class with 4 is not necessarily different from one with 6 methods if the methods are similar. To eliminate the sensitivity of the algorithm to size, we thus set the gap penalty to 0.

We define a predicate-specific function to measure the similarity. First, if the types differ, the similarity is 0. As we manipulate sequences of complex predicates and not strings, $sim_{i,j}$ is defined as a predicate-matching function, PM_{ij}. PM_{ij}, measures the similarity in terms of the elements of the predicates associated to a_i and b_j. This similarity is the ratio of common parameters in both predicates.

$$PM_{ij} = \frac{\forall_{p \in a_i, q \in b_i} \cap (p, q)}{max(|a_i|, |b_j|)|}$$

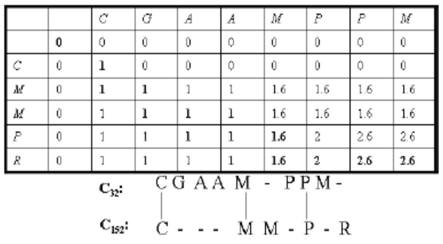
where a_i and b_j are treated as sets of predicates. The equivalence between predicate parameters depends on each type of parameter. For visibility and element types, it means equality. Specific names are not considered. Instead, they are used to indicate a common reference by other predicates. For example, if a class defines an attribute and its related getter method. They will both share the same class name.

To illustrate an example for the local alignment algorithm, let us consider the class *RangeExceptionImpl*, described previously, as a code fragment C_{32} and *Options*,C_{152} as a second code fragment to compare with C_{32}. The code fragments are sequentially numbered. C_{152} is defined as follows :

```
Class(Options,public);
Method(Options,isFractionalMetrics,boolean,N,public);
Method(Options,isTextAntialiased,boolean,N,private);
Parameter(Options,isTextAntialiased,id,String);
Relation(AbstractFigure;getFontRenderContext;isFractionalMetrics,
Options,N);
```

According to the coding mentioned previously, the predicate sequence for C_{32} is CGAAMPPM and one of C_{152} is CMMPR. The alignment algorithm finds the best alignment sequence as shown in Figure 4.

There are three matched predicates between C_{32} and C_{152}: one class, one method, and one method parameter. If we consider the second matched predicates $p1_5$ = `Method(RangeExceptionImpl,RangeExceptionImpl,void,Y,public)` from C_{32} and $p2_2$ = `Method(Options,isFractionalMetrics,boolean, N,public);` from C_{152}. The predicates have two common parameters out of a possible five. The resulting similarity is consequently 40%. We normalize this absolute similarity measure, $s_{n,m}$, by the maximum number of predicates to produce our similarity measure:



		C	*G*	*A*	*A*	*M*	*P*	*P*	*M*
	0	0	0	0	0	0	0	0	0
C	0	**1**	0	0	0	0	0	0	0
M	0	**1**	**1**	1	1	1.6	1.6	1.6	1.6
M	0	1	**1**	**1**	**1**	1.6	1.6	1.6	1.6
P	0	1	1	**1**	1	**1.6**	2	2.6	2.6
R	0	1	1	1	1	**1.6**	**2**	**2.6**	**2.6**

```
C32:   C G A A M  -  P P M -
       |       |     |
C152:  C -  - -  M M - P - R
```

Figure 4: Best alignment sequence between C_{32} and C_{152}

$$Sim(A, B) = \frac{s_{n,m}}{max(n, m)} \tag{1}$$

3.4 Detectors Generation

This section describes how a set of detectors is produced starting from the reference code. The generation, inspired by the work of Gonzalez and Dasgupta [16], follows a genetic algorithm [17]. The idea is to produce a set of detectors that best covers the possible deviations from the reference code. As the set of possible deviations is very large, its coverage may require a huge number of detectors, which is infeasible in practice. For example, pure random generation was shown to be infeasible in [18] for performance reasons. We therefore consider the detector generation as a search problem. A generation algorithm should seek to optimize the following two objectives:

- Maximize the generality of the detector to cover the non-self by minimizing the similarity with the self;
- Minimize the overlap (similarity) between detectors.

These two objectives define the cost function that evaluates the quality of a solution and, then guides the search. The cost of a solution D (set of detectors) is evaluated as the average costs of the included detectors. We derive the cost of a detector d_i as a weighted average between the scores of respectively, the lack of generality and the overlap. Formally,

$$cost(d_i) = \frac{LG(d_i) + O(d_i)}{2} \tag{2}$$

Here, we give equal weight to both scores. The lack of generality is measured by a matching score $LG(d_i)$ between the predicate sequence of a detector d_i and those of all the classes s_j in the reference code (call it S). It is defined as the average value of the alignment scores $Sim(d_i, s_j)$ between d_i and classes s_j in S. Formally,

$$LG_{d_i} = \frac{\sum_{s_j \in S} Sim(d_i, s_j)}{|S|}$$

Similarly, the overlap O_i, is measured by the average value of the individual $Sim(d_i, d_j)$ between the detector d_i and all the other detectors d_j in the solution D. Formally,

$$O_{d_i} = 1 - \frac{\sum_{d_j, j \neq i} Sim(d_i, d_j)}{|D|}$$

The cost function defined above is used in our genetic-based search algorithm. Genetic algorithms (GA) implement the principle of natural selection [17]. Roughly speaking, a GA is an iterative procedure that generates a population of individuals from the previous generation using two operators: crossover and mutation. Individuals having a high fitness have higher chances to reproduce themselves (by crossover), which improves the global quality of the population. To avoid falling in local optima, mutation is used to randomly change individuals. Individuals are represented by chromosomes containing a set of genes.

For the particular case of detector generation, we reuse the predicate sequences as chromosomes. Each predicate represents a gene. We start by randomly generating an initial population of detectors. The size of this population is a parameter that will be discussed later in Section 4. This size is maintained constant during the evolution. The fitness of each detector is evaluated by the inverse function of cost. The fitness determines the probability of being selected for the crossover. This process is called a wheel-selection strategy [17].

In fact, for each crossover, two detectors are selected by applying twice the wheel selection. Even though detector are selected, the crossover happens only with a certain probability. The crossover operator allows to create two offspring o_1 and o_2 from the two selected parents p_1 and p_2. It is defined as follows:

- A random position k, is selected in the predicate sequences.
- The first k elements of p_1 become the first k elements of o_1. Similarly, he first k elements of p_2 become the first k elements of o_2.
- The remaining elements of, respectively, p_1 and p_2 are added as second parts of, respectively, o_2 and o_1.

For instance, if k = 3 and p1 = CAMMPPP and p2 = CMPRMPP, then o1 = CAMRMPP and o2 = CMPMPPP.

The mutation operator operator consists of randomly changing a predicate.

3.5 Risk Estimation

The second step of our defect discovery is the assessment of risk for the different code fragments evaluated. These are also represented by predicate sequences. Each sequence is compared using the alignment algorithm to the detectors obtained in the previous step. The risk of being a defect, associated to a code fragment e_i is defined as the average value of the alignment scores $Sim(e_i, d_j)$ obtained by comparing e_i to respectively all the detectors of a set D. Formally,

$$risk_{e_i} = \frac{\sum_{d_j \in D} Sim_l(e_i, d_j)}{|D|}$$

The code fragments can then be ranked according to their risks to be inspected by the maintainers.

4. EVALUATION

To test our approach, we studied its usefulness to guide quality assurance efforts on two open-source programs. In this section, we describe our experimental setup and present the results of an exploratory study.

4.1 Goals and Objectives

The goal of the study is to evaluate the efficiency of our AIS approach for the discovery of design defects from the perspective of a software maintainer conducting a quality audit.

We present the results of the experiment aimed at answering the following research questions:

RQ1: To what extent can the proposed approach discover design defects?

RQ2: What types of defects does it locate?

To answer RQ1, we used an existing corpus of known design defects to evaluate the precision of our approach. We ranked classes in order of decreasing risk and compared results to produced by a rule-based strategy [4]. To answer RQ2, we investigated the type of defects that were found.

4.2 System Studied

We used three open-source Java projects to perform our experiments: GanttProject v1.10.2, Xerces v2.7.0, and JHotdraw v7.1. Table 1 summarizes facts on these programs. GanttProject[2] is a tool for creating project schedules by means of Gantt charts and resource-load charts. GanttProject enables breaking down projects into tasks and establishing dependencies between these tasks. Xerces[3] is a family of software packages for parsing and manipulating XML. It implements a number of standard APIs for XML parsing. JHotdraw v7.1[4] is a framework used to build graphic editors. It was first built as an example of the use of design patterns. JHotdraw was chosen because it contains very few known design defects. In fact, previous work [19] could not find any *Blob* defects. In our experiments, we used all of the classes in JHotdraw as our example set of good code. We chose the Xerces and Gantt libraries because they are medium sized open-source projects and were analysed in related work. The version of Gantt studied was known to be of poor quality, which lead to a new major version. Xerces-J on the other hand has been actively developed over the past 10 years and its design has not been responsible for a slowdown of its development.

In [4], Moha *et al.* asked three groups of students to analyse the libraries to tag instances of specific antipatterns to validate their detection technique, DECOR. For replication purposes, they provided[5] a corpus of describing instances of different antipatters including: *Blob* classes, *Spaghetti code*, and *Functional Decompositions*. Blobs are classes that do or know too much. Spaghetti Code (SC) is code that does not use appropriate structuring mechanisms. Functional Decomposition (FD) is code that is structured as a series of function calls. These represent different types of design risks. In our study, we verified the capacity of our

[2]http://ganttproject.biz/index.php
[3]http://xerces.apache.org/
[4]http://jhotdraw.org
[5]http://www.ptidej.net/research/decor/index_html

approach to locate classes that corresponded to instances of these antipatterns.

Systems	♯ classes	♯ Predicates	KLOC
GanttProjectv1.10.2	245	16640	31
Xerces v2.7.0	991	67810	240
JHotdraw v7.1	471	40354	91

Table 1: Program statistics

4.3 Experimental Setting

For our experiment, we randomly generated 100 detectors for JHotDraw (about a quarter of the number of examples) with a maximum size of 256 characters. The same set of detectors was used on both Xerces and Gantt. The obtained results[6] were compared to those of DECOR [4], a state of the art rule-based detection technique. For every antipattern in Xerces and Gantt, they published the number of antipatterns detected, the number of true positives, and the precision (ratio of true positives over the number detected). Our comparison is consequently done using precision. We would have liked to consider recall, but they did not publish clean, complete data describing all existing antipatterns. We therefore could not perform systematic comparisons of the recall of our approach. Instead, we discuss the proportion of "known antipatterns" detected.

4.4 Results

Tables 2 and 3 summarize our findings. Each class is presented with its risk, its size, and the associated defect types. We only presented classes with a risk level of >= 70%; this corresponds to about 5% of the classes in the system. For Gantt, our precision over the top 20 classes is 95% with the eight riskiest classes being true positives. DECOR on the other hand has a combined precision of 59% for its detection on the same set of antipatterns. For Xerces, our precision is of 90% with the top 30 classes correctly identified as defects. For the same dataset, DECOR had a precision of 67%. In the context of this experiment, we can conclude that our technique is able to accurately identify design anomalies more accurately than DECOR (RQ1).

We noticed that our technique does not have a bias towards the detection of specific anomaly types. In Xerces, we had an almost equal distibution of each antipattern (14 SCs, 13 Blobs, and 13 FDs). On Gantt, the distribution is not as balanced. This is principally due to the number of actual antipatterns in the system. We found all four known Blobs and all 11 SCs in the system. We found 6/17 FDs, two more than DECOR.

Having a relatively good distribution of antipatterns is useful for a quality engineer as he can focus on the notion of riskiest classes regardless of the type. Furthermore, since the results are ranked he can efficiently use his time unlike DECOR.

This ability to identify different types of antipatterns underlines a key strength to our approach: the similarity function is able to abstract out the importance of size. Most other tools and techniques rely heavily on the notion of size to detect defects. This is reasonable considering that some antipatterns like the Blob are associated to a notion of size.

[6] `http://www.iro.umontreal.ca/~sahraouh/papers/ASE2010/`

Class	Risk	S.C.	Blob	F.D.
GanttOptions	0.96	✓		
GanttTree	0.96		✓	
GregorianTimeUnitStack	0.93			✓
GanttDialogPerson	0.92	✓		
CSVSettingsPanel	0.9	✓		
GanttProject	0.9	✓	✓	
GanttTaskPropertiesBean	0.9	✓		
NewProjectWizard	0.87			✓
TimeUnitGraph	0.87			
ResourceLoadGraphicArea	0.85	✓	✓	
GanttCSVExport	0.82	✓		
GanttGraphicArea	0.82	✓	✓	
FindPossibleDependeesAlgo...	0.82			✓
GanttXFIGSaver	0.81	✓		
GanttApplet	0.79			✓
GraphicPrimitiveContainer	0.75	✓		
Shape	0.75			
GanttXMLSaver	0.75	✓		
RecalculateTaskCompletion...	0.71			✓
TaskHierarchyManagerImpl	0.71			✓
Precision	95%			

Table 2: Results for Gantt

For antipatterns like FDs however, the notion of size is irrelevant and this makes this type of anomaly hard to detect using structural information. This difficulty is why DECOR includes an analysis of naming conventions to perform its detection. Using naming convention means that their results depend on the coding practices of a development team. Our results are however comparable to theirs while we do not leverage lexical information.

4.5 Discussion

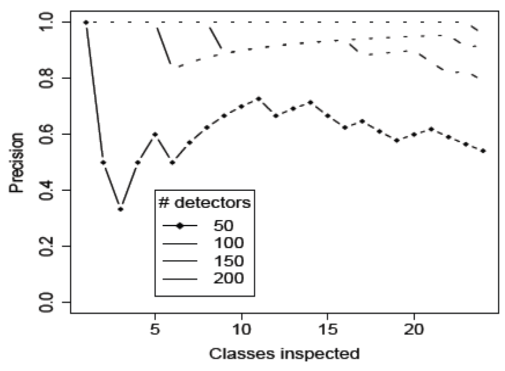
In this section, we discuss different issues concerning the detection of design risks.

Number of Detectors.

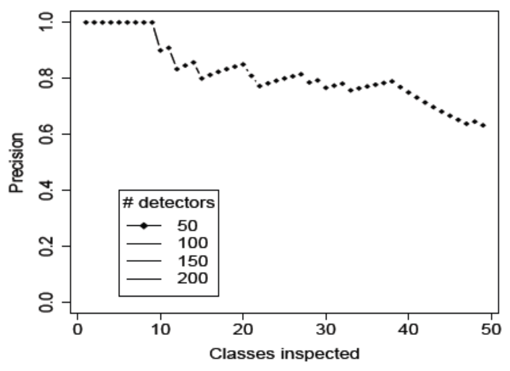
An important factor to our detection technique is the number of detectors generated. In Figure 5, we present the precision of our approach when varying the number of detectors (N_d) with $N_d = \{50,100,150,200\}$. The figure shows that the performance of our approach improves as we consider more detectors. When we use 200 detectors (50% of the total number of cases in JHotDraw), our performance is over 95% for both systems. Our technique requires the comparison of every class to every detector, this improved performance is at a negligible cost in terms of execution time. Indeed, the execution time for applying the detection on each system varies between 2 minutes for 50 detectors and 15 minutes for 200.

Variability in Detector Generation.

Another issue is our selection of interesting detectors. The detection results might vary depending on the detectors which are generated randomly (though guided by a meta-heuristic). To ensure that our results are relatively stable, we compared the results of multiple executions for detector generation. When we consider results up to 70% of risk, we observed an average precision of 92% for Gantt and 91% for Xerces. Furthermore, we found that the majority of defects detected are found in every execution (54% and 60% respectively for Gantt and Xerces). These unanimously detected defects were systematically the riskiest classes in every execution: in Gantt, the top 10 classes were common to all executions.



(a) Gantt



(b) Xerces

Figure 5: Effect of the number of detectors on detection precision vs. #classes inspected

In Xerces, there was only one non-unanimous class in the top ten classes returned which was a false-positive. The average rank for a class detected by a single execution was 18 and 34 for Gantt and Xerces respectively. We consequently believe that since the variability comes from the least risky classes, and that our technique is stable.

Metric-based Detection vs. Similarity-based Detection.

Our approach is significantly different from existing work that are rule-based. A key problem with these approaches is that these rules simplify the different notions that are useful for the detection of certain antipatterns. In particular, to detect blobs the notion of size is important. Most size metrics are highly correlated with one another, and the best measure of size can depend on the system itself. Our use of predicates allows for complex structures to be detected.

For example, we correctly detected TaskHierarchyManagerImpl in Gantt. It holds a reference to the root of the hierarchy, and controls creations of new children to the root.

```
public class TaskHierarchyManagerImpl {
    private TaskHierarchyItem myRootItem =
        new TaskHierarchyItem(null ,null);
    public TaskHierarchyItem getRootItem() {
        return myRootItem;
    }
    public TaskHierarchyItem createItem(Task task) {
        TaskHierarchyItem result =
           new TaskHierarchyItem(task, myRootItem);
        return result;
    }}
public class TaskManagerImpl implements TaskManager { ...
    private final TaskHierarchyManagerImpl myHierarchyManager
        = new TaskHierarchyManagerImpl();
    public TaskHierarchyManagerImpl getHierarchyManager() {
        return myHierarchyManager;
    }...}
```

It is detected for three reasons. First, it declares one attribute type (TaskHierarchyItem) on which it never invokes any methods. Second, it is used in a similar manner by TaskManagerImpl. Finally, apart from creating objects, it never uses any methods. It is consequently a datastructure.

These types of relationships are hard to detect using metrics. On the other hand, our technique produced a detector that was almost a complete match (except the final parameter):

```
Attribute(X,aaaa,AA,N,private); # myRootItem
Attribute(X,aa,X,N,private);    # myHierarchyManager
Class(X,N,N,public);            # TaskHierarchyManagerImpl
Method(X,z,X,Y,N,N,public);     # createItem
Method(X,zzzz,AA,N,N,N,public); # getRootItem
Method(X,zxzzz,X,N,N,N,public); # getHierarchyManager
Parameter(X,z,zuwe,gsfg,declaration); # task
Parameter(X,z,xuqye,fzfgg,local); # result
Parameter(X,zzzz,jdajg,gffgs,declaration); # Match error
```

DECOR also successfully identified this class. However, it did so, not because of metrics, but because the name of the class contains the term *Manager*.

Building an Example Data Set.

The reliability of the proposed approach requires an example set of good code. It can be argued that constituting such a set might require more work than identifying and adapting rules. In our study, we showed that by using JHotdraw directly, without any adaptation, the technique can be used out of the box and this will produce good detection results for the detection of antipatterns for the two systems studied. The performance of this detection (in terms of precision) was superiour to that of DECOR. In an industrial setting, we could expect a company to start with JHotDraw, and gradually migrate its set of good code examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

5. RELATED WORK

Several studies have recently focused on detecting design defects in software using different techniques. These techniques range from fully automatic detection to guided manual inspection. The related work can be classified into three broad categories: metric-based detection, detection of refactoring opportunities, visual-based detection.

Class	Risk	S.C.	Blob	F.D.
DFAContentModel	0.97	✓		
XSFacets	0.96			✓
XMLSerializer	0.96	✓		
XMLVersionDetector	0.96			✓
XML11EntityScanner	0.93	✓		
XSDHandler	0.92		✓	✓
Token	0.91	✓		
XMLEntityManager	0.91		✓	
XSDAbstractTraverser	0.91			✓
XML11DTDValidator	0.91			✓
DOMNormalizer	0.91		✓	
XMLNSDTDValidator	0.88			✓
ParserConfigurationSettings	0.88			✓
SAXParser	0.85			✓
DTDGrammar	0.84		✓	
XML11NonValidatingConfiguration	0.84		✓	
XMLDTDValidator	0.84		✓	
XMLEntityScanner	0.84	✓		
XSAttributeGroupDecl	0.82	✓		
AbstractDOMParser	0.81	✓		
SchemaDOM	0.81			✓
XML11DTDConfiguration	0.81		✓	
XSDAttributeTraverser	0.81	✓		
ObjectFactory	0.8	✓		
XIncludeHandler	0.8		✓	
XSDFACM	0.78	✓		
NonValidatingConfiguration	0.78		✓	
XMLSchemaValidator	0.78		✓	
DTDConfiguration	0.77		✓	
CoreDocumentImpl	0.77	✓	✓	
XSAttributeChecker	0.77		✓	
CMNodeFactory	0.77			
RegexParser	0.75			✓
TimeDV	0.75			
XML11Configuration	0.74	✓		
XMLFilterImpl	0.73			
DOMSerializerImpl	0.71	✓		
XSFacets	0.71			✓
BaseMarkupSerializer	0.71			
ElementSchemePointer	0.71			✓
XMLParser	0.7			✓
XPathMatcher	0.7	✓		
Precision	90%			

Table 3: Results for Xerces

In first category, Marinescu [6] defined a list of rules relying on metrics to detect what he calls design flaws of OO design at method, class and subsystem levels. Erni *et al.* [20] use metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, as an n-tuple of metrics expressing a quality criterion (*e.g.*, modularity). The main limitation of the two previous contribution is the difficulty to define threshold values for metrics in the rules. To circumvent this problem, Alikacem *et al.* [21] express defect detection as fuzzy rules with fuzzy label for metrics, *e.g., small, medium, large.* When evaluating the rules, actual metric value are mapped to truth value for the labels by means of membership functions. Although no thresholds have to be defined, still, it is not obvious to decide for membership functions.

The previous approaches start from the hypothesis that all defect symptoms could be expressed in terms of metrics. Actually, many defects involve notions that could not quantified. This observation was the foudation of the work of Moha *et al.* [4]. In their approach, named DECOR, they start by describing defect symptoms using an abstract rule language. These descriptions involve different notions such as class roles and structures. The descriptions are later mapped to detection algorithms. In addition to the threshold problem, this approach uses heuristics to approximate some notions with results in an important rate of false positives. Another limitation of DECOR is that all the detected defect candidate are listed without any rank that help the maintainers checking/addressing in priority the most severe ones.

Khomh *et al.* [3] extended DECOR to support uncertainty and to sort the defect candidates accordingly. Uncertainty is managed by Bayesian belief networks that implement the detection rules of DECOR. The detection outputs are probabilities that a class is an occurrence of a defect type.

In our approach, all the above mentioned problems related to the use of rules and metrics do not arise. Indeed, the symptoms are not explicitly used, which reduces the adaptation/calibration effort.

In the second category of work, defects are not detected explicitly. They are implicitly because, the approaches refactor a system by detecting elements to change to improve he global quality. For example, in [22], defect detection is considered as an optimization problem. The authors use a combination of 12 metrics to measure the improvements achieved when sequences of simple refactorings are applied, such as moving methods between classes. The goal of the optimization is to determine the sequence that maximize a function, which captures the variations of a set of metrics [23]. The fact that the quality in terms of metrics is improved does not necessary means that the changes make sense. The link between defect and correction is not obvious, which make the inspection difficult for the maintainers. In our case, we separate the detection and correction phase.

The high rate of false positives generated by the automatic approaches encouraged other teams to explore semi-automatic solutions. These solutions took the form of visualization based environments. The primary goal is to take advantage of the human ability to integrate complex contextual information in the detection. Kothari *et al.* [24] present a pattern-based framework for developing tool support to detect software anomalies by representing potentials defects with different colors. Later, Dhambri *et al.* [25] propose a visualization-based approach to detect design anomalies by automatically detecting some symptoms and letting others to the human analyst. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Still, the visualization approach is not obvious when evaluating large-scale systems. Moreover, the information visualized is for the most part metric-based meaning that complex relationships can still be difficult to detect.

In our case, the human intervention is needed for the inspection of the candidate only. This inspection is made easier because, the candidates are ranked by risk, and also because, by analysing the most similar detectors, it is possible to identify what part of the element was problematic.

The work that is closest to ours is by Catal and Diri [26]. The authors use a machine-learning version of AIS called an Artificial Immune Recognition System (AIRS) to learn a prediction model for defect-prone modules. The AIRS used was a generic package implemented in the machine-learning package Weka. This package cannot handle complex structures like predicates and does not implement the negative selection algorithm.

6. CONCLUSION

In this article, we presented a new approach to problem of detecting design defects. Typically, researchers and practitioners try to characterize different types of common design defects and present symptoms to use in order to locate them in a system. In our work, we show that we do not need this knowledge to perform a detection. Instead, all we need is a clear notion of what is good. What significantly diverges is often a defect. Interestingly enough, our study shows that our technique outperforms an DECOR [4], a state of the art, rule-based approach on its test corpus.

By ignoring the detection of specific defect types, we avoid problems with existing detection techniques. First, the detection of most defect is difficult to automate because their definitions are expressed informally. Second, even with a precise definition, some symptoms are context-specific and might or not be useful for a given system. There is consequently a non-negligeable effort to test and adapt a detection process to another system. Finally, by presenting all defects, regardless of types in order of risk, a development team can focus on the most urgent problems first.

This technique was tested on two open-source systems and the results were promising. The discovery process uncovered different types of design defects was more efficiently than DECOR. In fact, for Gantt, our precision is 95% with the eight riskiest classes being true positives. DECOR on the other hand has a combined precision of 59% for its detection of the same set of antipatterns. For Xerces, our precision is of 90% with the top 30 classes correctly identified as defects. For the same dataset, DECOR had a precision of 67%. Furthermore, as DECOR needed an expert to define rules, our results were achieved without any expert knowledge, relying only on the good structure of JHotdraw to guide the detection process.

In this work, we only looked at the first step of an immune systems: the discovery of risk. As part of our future work, we plan to explore the other two steps: identification and correction of detected design defects (refactoring). Furthermore, we need to extend our reference code base with other well-designed code in order to take into consideration different programming contexts. Specifically, we plan on:

- Adapting the AIS metaphor to *identify* discovered defects using immune memory and danger theory [27].
- Adapting the colonal selection algorithm [28] to find the best immune response that correspond the optimal refactorings sequence to apply.
- Using our approach for defect prediction using the estimation risk score.

Acknowledgment

This work has been partly funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Tunisian Ministry of Higher Education and Scientific Research.

7. REFERENCES

[1] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1^{st} ed. Addison-Wesley, June 1999.

[2] N. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. London, UK: International Thomson Computer Press, 1997.

[3] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design Smells," in *Proceedings of the 9^{th} International Conference on Quality Software*, D.-H. Bae and B. Choi, Eds. IEEE Computer Society Press, August 2009.

[4] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *Transactions on Software Engineering (TSE)*, 2009, 16 pages. [Online]. Available: http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/TSE09.doc.pdf

[5] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, "Facilitating software refactoring with appropriate resolution order of bad smells," in *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 265–268.

[6] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the International Conference on Software Maintenance*, 2004, pp. 350–359.

[7] F. Azuaje, "Review of "artificial immune systems: a new computational intelligence approach" by l.n. de castro and j. timmis (eds) springer, london, 2002," *Neural Netw.*, vol. 16, no. 8, pp. 1229–1229, 2003.

[8] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[9] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998. [Online]. Available: www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theantipatterngr/103-4749445-6141457

[10] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, p. 381.

[11] W. C. Wake, *Refactoring Workbook*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[12] K. J., *Immunology*, 5th ed. by Richard A. Goldsby, Thomas J. Kindt, Barbara A. Osborne, W.H, 2002.

[13] D. Dasgupta, Z. Ji, and F. Gonzalez, "Artificial immune system (ais) research in the last five years." in *IEEE Congress on Evolutionary Computation (1)*. IEEE, 2003, pp. 123–130. [Online]. Available: http://dblp.uni-trier.de/db/conf/cec/cec2003-1.html#DasguptaJG03

[14] L. Nanni and A. Lumini, "Generalized needleman-wunsch algorithm for the recognition of t-cell epitopes," *Expert Syst. Appl.*, vol. 35, no. 3, pp. 1463–1467, 2008.

[15] M. Brudno, "Algorithms for comparison of dna sequences," Ph.D. dissertation, Stanford, CA, USA, 2004, adviser-Batzoglou, Serafim.

[16] F. A. González and D. Dasgupta, "Anomaly detection using real-valued negative selection," *Genetic*

Programming and Evolvable Machines, vol. 4, no. 4, pp. 383–403, 2003.

[17] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[18] H. Hou and G. Dozier, "An evaluation of negative selection algorithm with constraint-based detectors," in *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*. New York, NY, USA: ACM, 2006, pp. 134–139.

[19] I. G. Czibula and G. Czibula, "Clustering based automatic refactorings identification," in *SYNASC '08: Proceedings of the 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 253–256.

[20] K. Erni and C. Lewerentz, "Applying design metrics to object-oriented frameworks," in *Proc. IEEE Symp. Software Metrics*. IEEE Computer Society Press, 1996.

[21] H. Alikacem and H. Sahraoui, "Détection d'anomalies utilisant un langage de description de règle de qualité." in *actes du 12e colloque LMO*, LMO, Ed., 2006.

[22] M. O'Keeffe and M. . Cinnéide, "Search-based refactoring: an empirical study." *Journal of Software Maintenance*, vol. 20, no. 5, pp. 345–364, 2008. [Online]. Available: http://dblp.uni-trier.de/db/journals/smr/smr20.html#OKeeffeC08

[23] M. Harman and J. A. Clark, "Metrics are fitness functions too." in *IEEE METRICS*. IEEE Computer Society, 2004, pp. 58–69. [Online]. Available: http://dblp.uni-trier.de/db/conf/metrics/metrics2004.html#HarmanC04

[24] S. C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty, "A pattern-based framework for software anomaly detection," *Software Quality Journal*, vol. 12, no. 2, pp. 99–120, June 2004. [Online]. Available: http://springerlink.com/content/v115717r15420214/?p=bf86b148d5d74754baec247cd0661c7c{\&}pi=53

[25] K. Dhambri, H. A. Sahraoui, and P. Poulin, "Visual detection of design anomalies." in *CSMR*. IEEE, 2008, pp. 279–283. [Online]. Available: http://dblp.uni-trier.de/db/conf/csmr/csmr2008.html#DhambriSP08

[26] C. Catal and B. Diri, "Software defect prediction using artificial immune recognition system," in *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*. Anaheim, CA, USA: ACTA Press, 2007, pp. 285–290.

[27] S. Rawat and A. Saxena, "Danger theory based syn flood attack detection in autonomic network," in *SIN '09: Proceedings of the 2nd international conference on Security of information and networks*. New York, NY, USA: ACM, 2009, pp. 213–218.

[28] W. Pang and G. M. Coghill, "Modified clonal selection algorithm for learning qualitative compartmental models of metabolic systems," in *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2007, pp. 2887–2894.

Chapter 6: Design Defects Detection Rules Generation by Example

6.1 Introduction

In this chapter, we propose another automated approach to derive rules for design defect detection. Instead of specifying rules manually for detecting each defect type, or semi-automatically using defect definitions, we extract them from valid instances of design defects. In our setting, we view the generation of design defect rules as an optimization problem, where the quality of a detection rule is determined by its ability to conform to a base of examples that contains instances of manually validated defects (classes). The generation process starts from an initial set of rules that consists of random combinations of metrics. Then, the rules evolve progressively according to the set's ability to detect the documented defects in the example base. Due to the potentially huge number of possible metric combinations that can serve to define rules, a heuristic approach is used instead of exhaustive search to explore the space of possible solutions. To that end, we use a rule induction heuristic based on Harmony Search (HS) [56] to find a near-optimal set of detection rules.

This contribution was accepted for publication in the 13th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2011) [80]. The paper, entitled "Design Defects Rules Generation: A Music Metaphor", is presented in the next section.

6.2 Design Defects Detection Rules Generation by Example

Design Defects Detection Rules Generation: A Music Metaphor

Marouane Kessentini[1], Mounir Boukadoum[2],
Houari Sahraoui[1] and Manuel Wimmer[3],

Abstract — We propose an automated approach for design defect detection. It exploits an algorithm that automatically finds rules for the detection of possible design defects, thus relieving the designer from doing so manually. Our algorithm derives rules in the form of metric/threshold combinations, from known instances of design defects (defect examples). Due to the large number of possible combinations, we use a music-inspired heuristic that finds the best harmony when combining metrics. We evaluated our approach on finding potential defects in three open-source systems (Xerces-J, Quick UML and Gantt). For all of them, we found more than 80% of known defects, a better result when compared to a state-of-the-art approach, where the detection rules are manually specified.

Keywords — design defects; software maintenance; search-based software engineering;

1. Introduction

There has been much research focusing on the study of bad design practices, also called defects, antipatterns [1], smells [2], or anomalies [3] in the literature. Although these bad practices are sometimes unavoidable, they should otherwise be prevented by the development teams and removed from their code base as early as possible. Hence, several fully-automated detection techniques have been proposed [4, 5, 6].

Problems exist that may limit the effectiveness of the existing techniques. Indeed, the vast majority of existing work relies on rule-based detection [5, 7], where different rules identify the key symptoms that characterize a defect using combinations of mainly quantitative (metrics), structural, and/or lexical information. However, it may be difficult to express these symptoms as rules [8], and the number of possible defects to manually characterize with rules can be so large as to make it difficult to decide what type of defect is detected. In addition, after manually finding the best metrics combination for detecting each design defect, substantial calibration efforts are needed to define a threshold for each metric. These difficulties explain a large portion of the high false-positive rates mentioned in existing research [6]. On the other hand, many defect repositories exist and in many companies, the defects that are identified and corrected are documented. This represents a good source of examples that can be exploited to derive rules automatically.

In this paper, we propose a new automated approach for design defects detection. Instead of specifying rules manually for detecting each defect type, we explore a solution for automating the rule generation process, starting from the hypothesis that valid instances of design defects, called defects examples, can be used to generate detection rules. Indeed, we propose to view design defect rule generation as an optimization problem where rules are automatically derived from available examples. In our case, each example corresponds to an instance of defects (classes) that was validated manually. Then, our contribution starts by randomly generating a set of rules that corresponds to metrics combination and executing them to detect some potential design defects (classes). Then, it evaluates the quality of the proposed solution (rules) by comparing the detected classes and the expected ones from the base of defects examples. Due to the large number of possible rules (metrics combination), a computational method is used to build the solution. To achieve this goal, we used a music-inspired heuristic, called Harmony Search (HS) [9], for finding the best "harmony" when combining metrics. Thus, we draw an analogy with music composition by considering each rule (set of metrics combination) to be a musician, and the various sets of metrics threshold values to be a collection of notes in a musicians' memories. Then, finding the optimal set of rules for detecting design defects rules is akin to the orchestra trying to find the best harmony when playing music. In the musical metaphor, this is accomplished
by the musicians polishing their pitches in order to obtain better harmony [9]. In doing so, HS transforms the qualitative improvisation process into quantitative rules by idealization, and thus turning the beauty and harmony of music into an optimization procedure through the search for perfect harmony. HS has solved an impressive range of problems (e.g., see [9]); however, to our best knowledge, its use in software engineering is a relatively unexplored area.

We believe that this is a promising approach for automating defects detection, due to HS effectiveness in searching very large spaces, such as in the case of quality metrics combination [10].
To evaluate our approach, we used classes from three open source projects, Gantt [11], Quick UML [26] and Xerces-J [12], as examples of badly-designed and implemented code. We used a 3-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining two systems as bases of examples. For example, Xerces-J is analyzed using some defects examples from Gantt and Quick UML. Almost all the identified classes were found, with a precision superior to 80% in a list of classes tagged as defects (blobs, spaghetti code and functional decomposition) in previous projects [5]. The recall also was more than 80%.
The benefits of our approach are as follows: 1) it is fully automatable; 2) it does not require an expert to write rules manually, for every defect type, and adapt them to different systems; 3) the rules generation process is executed once, and the obtained rules can then be used to evaluate any system; 4) our technique outperforms an existing technique [5] in terms of precision as shown in the validation section.
The major limitations of our approach are: 1) we require a code base for representing bad design practices. 2) our rules are solely based on metrics and some defects may require additional or different knowledge to be detected 3) we must ensure that all possible design defects are detected manually in the code base. Nevertheless, our results indicate that some defects examples from Gantt and Xerces-J appear to be usable and could serve as a starting point for a company wishing to use our approach.
The remainder of this paper is structured as follows. Section 2 is dedicated to the problem statement. In Section 3, we describe the overview of our proposal. Then, Section 4 describes the principles of the HS algorithm used in our approach and the adaptations needed to our problem. Section 5 presents and discusses the validation results. A summary of the related work in defect detection is given in Section 6. We conclude and suggest future research directions in Section 7.

2. Problem Statement

In this section, we describe the problem of defect detection. We start by defining important concepts. Then, we detail the specific problems that are addressed by our approach.

2.1 Defintions

Design defects, also called design anomalies, refer to design situations that adversely affect the development of software. In general, they make a system difficult to change, which may in turn introduce bugs.
Different types of defects, presenting a variety of symptoms, have been studied in the intent of facilitating their detection [13] and suggesting improvement paths. The two following types of defects are commonly mentioned. In [2], Beck defines 22 sets of symptoms of common defects, named code smells. These include large classes, feature envy, long parameter lists, and lazy classes. Each defect type is accompanied by some refactoring suggestions to remove it. Brown et al. [1] define another category of design defects, named anti-patterns, which includes blob classes, spaghetti code, and cut & paste programming. In both books, the authors focus on describing the symptoms to look for, in order to identify specific defects.
The design defects detection process itself consists of finding bad code fragments in the system. In general, this process is based on the use of software metrics, each one measuring some property of a piece of software or its specifications [14]. Different kinds of metrics are available to use for identifying design defects: coupling, cohesion, program size, etc [14].

2.2 Problem Statement

Any technique to detect design defect should address/circumvent difficulties that are inherent in the defects. Next is a description of the most important difficulties and how they affect an automation process.

- Different design defects have the same symptoms and it is difficult to manually define rules for similar ones. Although there has been significant work to classify defect types [13, 15, 16], programming practices, paradigms and languages evolve, making unrealistic to support the detection of all possible similar defect types. Furthermore, there might be company- or application-specific (bad) design practices.
- For those design defects that are documented, there is no consensual definition of symptom detections. Defects are generally described using natural language and their detection relies on the interpretations of the developers. As a result, it is difficult for an expert to define the detection rules manually. This limits the automation of the detection process.
- The list of possible defects can be very large [8]. This makes it a a fastidious task to specify rules manually for each defect.
- It is recognized that experts can more easily provide examples than complete and consistent rules [8]. This is particularly true for industrial organizations where a memory of past detected defects examples can be found.

3. Approach Overview

This section shows how, under some circumstances, design defects detection can be seen as an optimization problem. We also show why the size of the corresponding search space makes heuristic search necessary to explore it.

3.1 Overview

We propose an approach that uses knowledge from previously manually inspected projects, called defects examples, in order to detect design defects to generate new detection rules based on a combinations of software quality metrics. More specifically, the detection rules are automatically derived by an optimization process that exploits the available examples.

Figure 1 shows the general structure of our approach. The approach takes as inputs a base of examples (*i.e.,* a set of defects examples) and a set of quality metrics, and generates as output a set of rules. The generation process can be viewed as the combination of the metrics that best detect the defects examples. In other words, the best set of rules is the one that detect the maximum number of defects.

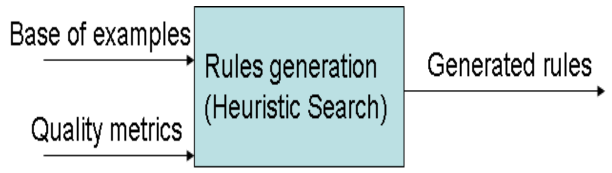


Fig 1. Approach overview

As showed in Figure 2, the base of examples contains projects (systems) that were inspected manually to detect possible defects. In the training process, these systems are iteratively evaluated using rules generated by the algorithm. A fitness functions calculates the quality of each solution (rules) by comparing the list of detected defects with the expected ones from the base.

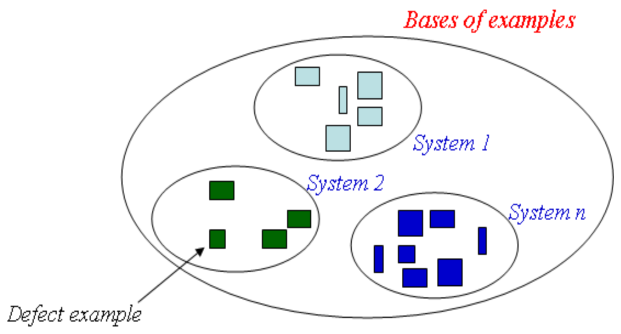


Fig 2. Base of examples

As many metrics combinations are possible, the rules generation process is a combinatorial optimization problem. The number of possible solutions quickly becomes huge as the number of metrics increases. A deterministic search is not practical in such cases, and the use of heuristic search is warranted (see Problem Complexity below). The dimensions of the solution space are set by the metrics and logical operations between them: union (metric1 OR metric2) and intersection (metric1 AND metric2). A solution is determined by assigning a threshold value to each metric. The search is guided by the quality of the solution according to the number of detected defects in comparison to expected ones in the base of examples.

To explore the solution space, we use a global heuristic search by means of the Harmony Search algorithm [9] that was introduced previously and that will be detailed in Section 4.

3.2 Problem Complexity

Our approach assigns to each metric a corresponding threshold value. The number *m* of possible threshold values is usually very large. Furthermore, the rules generation process consists of finding the best combination between *n* metrics. In this context, $(n!)^{m}$ possible solutions have to be explored. This value can quickly become huge. A list of 5 metrics with 6 possible thresholds necessitates exploring at least 120^6 combinations. Considering these magnitudes, an exhaustive search cannot be used within a reasonable time frame. In such situations, or when a formal approach is not possible, heuristic search can be used.

4. Search-based Rules Generation

We describe in this section the adaptation of HS to the automatic generation of design defects detection rules. As a first step, one must specify the encoding of solutions and the fitness function to evaluate a solution's quality. These two elements are detailed in subsections B and C, respectively.

4.1 Harmony Search Algorithm

The HS algorithm is based on musical performance processes that occur when a musician searches for a better state of harmony, such as during jazz improvisation [9]. Jazz improvisation seeks to find a musically-pleasing harmony as determined by an aesthetic standard, just as the optimization process seeks to find a global solution as determined by a fitness function. The pitch of each musical instrument determines the aesthetic quality, just as the fitness function value that determines the quality of a solution.

In music improvisation, each player tries pitches within the possible range, collectively creating a harmony vector. If all the pitches make for a good harmony, the experience is stored in each player's memory, and the possibility to create a good harmony is increased next time. Similarly, in an engineering optimization, each dimension initially chooses values within the possible range to collectively create a solution vector. If this set of the values from the different dimensions represents a good solution, that experience is stored, and the possibility to find a good solution is also increased next time.

The general HS algorithm works as follows:

Step1: Problem and algorithm parameter initialization.

The HS algorithm parameters are specified in this step. They are the harmony memory size (HMS) or number of solution vectors in the harmony memory; the harmony memory consideration rate (HMCR); the bandwidth (bw); the pitch adjustment rate (PAR); and the number of improvisations (K) or stopping criterion.

Step 2: Harmony memory initialization.
The initial harmony memory is generated from a uniform distribution in the range [x_{imin},x_{imax}] (i = 1,2, . . .,N) , as shown in Equation 1 :

$$HM = \begin{bmatrix} x_1^1 & x_2^1 & . & . & x_N^1 \\ . & & & & \\ . & & & & \\ x_1^{HMS} & x_2^{HMS} & & . & x_N^{HMS-1} \end{bmatrix} \quad (1)$$

Step 3: New harmony improvisation.
Generating a new harmony is called improvisation. The new harmony vector x' = (x'$_1$, x'$_2$,…, x'$_N$) is determined by the memory consideration, the pitch adjustment and random selection. Algorithm 1 summarizes the generic HS procedure [9].

```
HS algorithm
1: for each $i \in [1,N]$ do
2:         if rand( ) $\leq$ HMCR then
3:             $x'_i = x_j^i$ (j= 1,2,...,HMS)%memory consideration
4:         if rand( ) $\leq$ PAR then
5:             $x'_i = x_j^i \pm$ rand()* bw%pitch adjustment
6:             if $x'_i \leq x_{imin}$ then
7:                 $x'_i = x_{imin}$
8:             elseif $x'_i \geq x_{imax}$ then
7:                 $x'_i = x_{imax}$
9:     else
 10:           $x'_i = x_{imin} \pm$ rand()* $(x_{imax} - x_{min})$%random selection
```
Algorithm 1. HS algorithm

Step 4: Harmony memory update
If the fitness of the improvised harmony vector x' = (x'$_1$, x'$_2$,…, x'$_N$) is better than the worst harmony, replace the worst harmony in the IHM with x'.

Step 5:.Stopping criterion check
If the stopping criterion (e.g., the maximum number of iterations K) is satisfied, the algorithm stops; otherwise, step 3 is repeated.
The most important step of the HS algorithm is Step 3, and it includes memory consideration, pitch adjustment and random selection. PAR and bw have a profound impact on the performance of the HS algorithm. [10] proposes to dynamically update PAR and bw as follows:

$$PAR(k) = PAR_{\min} + \frac{PAR_{\max} - PAR}{K} \quad (2)$$

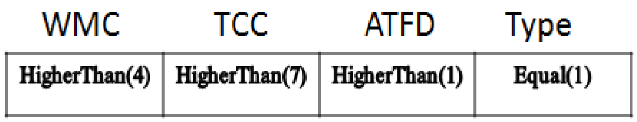
$$bw(k) = bw_{\max} \exp\left(\frac{\ln(\frac{bw_{\min}}{bw_{\max}})}{K} \right) \quad (3)$$

4.2 Solution Representation

One key issue when applying a search-based technique is to find a suitable mapping between the problem to solve and the techniques to use, *i.e.*, in our case, generating design defects detection rules. As stated in Section 3, we view the set of potential solutions as points in a *n*-dimensional space, where each dimension corresponds to one metric or operator (union or intersection) applied to them. Figure 3 shows an example where the *i*th harmony vector, HM^i, stands for the rule: if (WMC≥4) AND (TCC≥7) AND (ATFD≥1) Then Defect_Type(1)_detected. The WMC, TCC and ATFD are metrics defined as [14]:

- *Weighted Method Count* (WMC) is the sum of the statical complexity of all methods in a class. We considered the McCabe's cyclomatic complexity as a complexity measure.
- *Tight Class Cohesion* (TCC) is the relative number of directly connected methods.
- *Access to Foreign Data* (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods.

We used three types of defects: blob, spaghetti code and functional decomposition.



WMC	TCC	ATFD	Type
HigherThan(4)	HigherThan(7)	HigherThan(1)	Equal(1)

Fig 3. An example of the ith harmony

The default operator used is the intersection (AND). The other operator (union or OR) can be used as a dimension. The harmony vector presented in Figure 3 generates only one rule. However, a vector may contain many rules separated by the dimension "Type".

4.3 Evaluating Solutions

The *fitness function* quantifies the quality of the generated rules. As discussed in Section 3, the fitness function checks to maximize the number of detected defects in comparison to the expected ones in the base of examples. In this context, we define the fitness function of a solution as

$$f = \sum_{i=1}^{p} a_i \qquad (4)$$

where p is the number of detected classes, and a_i has value 1 if the i^{th} detected classes exists in the base of examples, and value 0 otherwise.

5. Validation

To test our approach, we studied its usefulness to guide quality assurance efforts for an open-source program. In this section, we describe our experimental setup and present the results of an exploratory study.

5.1 Goals and Objectives

The goal of the study is to evaluate the efficiency of our approach for the detection of design defects from the perspective of a software maintainer conducting a quality audit. We present the results of the experiment aimed at answering the following research questions:
RQ1: To what extent can the proposed approach detect design defects?
RQ2: What types of defects does it locate?
To answer RQ1, we used an existing corpus of known design defects to evaluate the precision and recall of our approach. We compared our results to those produced by an existing rule-based strategy [5]. To answer RQ2, we investigated the type of defects that were found.

5.2 System Studied

We used three open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2, Quick UML v2001 and Xerces-J v2.7.0. Gantt is a tool for creating project schedules by means of Gantt charts and resource-load charts. Gantt enables breaking down projects into tasks and establishing dependencies between them. Xerces-J is a family of software packages for parsing and manipulating XML. It implements a number of standard APIs for XML parsing. Table 1 provides some relevant information about the programs.

TABLE I. PROGRAM STATISTICS.

Systems	Number of classes	KLOC
GanttProject v1.10.2	245	31
Xerces-J v2.7.0	991	240
Quick UML v2001	142	19

We chose the Xerces-J, Quick UML and Gantt libraries because they are medium-sized open-source projects and were analysed in related work. The version of Gantt studied was known to be of poor quality, which has led to a new major revised version. Xerces-J and Quick UML, on the other hand, has been actively developed over the past 10 years and their design has not been responsible for a slowdown of their developments. Consequently, we used some of the classes in Gantt for our example set of design defects in our experiments. The examples were manually validated by a group of experts [17].
In [5], Moha et al. asked three groups of students to analyse the libraries to tag instances of specific antipatterns to validate their detection technique, DECOR. For replication purposes, they provided a corpus of describing instances of different antipatterns that includes blob classes, spaghetti code, and functional decompositions. Blobs are classes that do or know too much; spaghetti Code (SC) is code that does not use appropriate structuring mechanisms; finally, functional decomposition (FD) is code that is structured as a series of function calls. These represent different types of design risks. In our study, we verified the capacity of our approach to locate classes that corresponded to instances of these antipatterns. As previously mentioned in Introduction, we used a 3-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining two systems as base of examples. For example, Xerces-J is analyzed using some defects examples from Gantt and Quick UML.
The obtained results were compared to those of DÉCOR. Since [5] reported the number of antipatterns detected, the number of true positives, the recall (number of true positives over the number of design defects) and the precision (ratio of true positives over the number detected), we determined the values of these indicators when using our algorithm for every antipattern in Xerces-J, Quick UML and Gantt,

5.2 Experimental Setting

To set the parameters of the HS algorithm, we started with commonly found values in the literature [10] and adapted them to the particularities of the design defects detection problem. The final parameters values were set as follows:

- The harmony memory size (HMS), or number of solution vectors in each iteration, was set to 50. We found this number to provide a good balance between population diversity and the quantity of used metrics.
- The harmony memory considering rate (HMCR), the pitch adjusting rate (PAR) and the bandwidth (bw) were set to 1.1, 1.4 and 0.8, respectively.
- The maximum number of iterations was set to 500. This is a generally accepted heuristic [9].
- Since two different executions of a search heuristic may produce different results, we decided to take the best result from 5 executions.

The list of metrics used for our experiments can be found in [18].

5.3 Results

Figures 3, 4 and 5 summarize our findings. Each class is presented with the associated defect types. Theses classes correspond to about 5% of the classes in the system. For Gantt, our average antipattern detection precision was 87%. DÉCOR, on the other hand, had a combined precision of 59% for the same antipatterns. The precision for Quick UML was about 86%, over twice the value of 42% obtained with DECOR. In particular, DÉCOR did not detect any spaghetti code in contradistinction with our approach. For Xerces-J, our precision average was 81%, while DECOR had a precision of 67% for the same dataset. However, the recall score for both systems was less than that of DECOR. In fact, the rules defined in DECOR are large and this is explained by the lower score in terms of precision. In the context of this experiment, we can conclude that our technique was able to identify design anomalies more accurately than DECOR (answer to research question RQ1 above).

Class	*Spaghetti*	*Blob*	*F.D.*
Access			x
CSVSettingsPanel	x		
Document	x		
FindPossibleDependeesAlgorithmImpl			x
GanttApplet			x
GanttCellListRenderer	x		
GanttCSVExport	x		
GanttDialogPerson	x		
GanttGraphicArea		x	
GanttOptions	x		
GanttProject		x	
GanttTaskPropertiesBean	x		
GanttTree		x	
GanttTXTOpen			x
GanttXFIGSaver	x		
GanttXMLSaver	x		
GraphicPrimitiveContainer	x		
GregorianTimeUnitStack			x
NewProjectWizard			x
RecalculateTaskCompletionPercentage			x
ResourceLoadGraphicArea	x	x	
Shape			
TaskHierarchyManagerImpl			x
Precision	9/11=82%	4/4=100%	7/8=87%
Recall	10/11=90%	4/4=100%	8/17=47%

Fig. 3. Results for Gantt

Class	*Spaghetti*	*Blob*	*F.D.*
MetaMethod			x
GeneralizationTool			x
MetaParameter			x
CardinalityTool			x
AbstractBuilder			x
Precision	100%	100%	3/5=60%
Recall	100%	100%	3/8=38%

Fig. 4. Results for Quick UML

Class	***Spaghetti***	***Blob***	***F.D***
AbstractDOMParser	x		
CharacterDataImpl			x
CoreDocumentImpl	x	x	
DFAContentModel	x		
DOMNormalizer		x	
DOMSerializerImpl	x		
DTDConfiguration		x	
DTDGrammar		x	
ElementSchemePointer			x
HTMLMapElementImpl	x		
HTMLTextAreaElement	x		
NodeIteratorImpl			x
NonValidatingConfiguration		x	
ObjectFactory	x		
ParserConfigurationSettings			x
RegexParser			x
SAXParser			x
SchemaDOM			x
SymbolTable		x	
Token	x		
Util	x		
WMLTimerElement			x
XIncludeHandler		x	
XML11Configuration	x		
XML11DTDConfiguration		x	
XML11DTDValidator			x
XML11EntityScanner	x		
XML11NonValidatingConfiguration		x	
XMLDTDValidator		x	
XMLEntityManager		x	
XMLEntityScanner	x		
XMLNSDTDValidator			x
XMLParser			x
XMLSchemaValidator		x	
XMLSerializer	x		
XMLVersionDetector			x
XPathMatcher	x		
XSAttributeChecker		x	
XSAttributeGroupDecl	x		
XSDAbstractTraverser			x
XSDAttributeTraverser	x		
XSDFACM	x		
XSDHandler		x	x
XSFacets			x
XSFacets			x
XSModelImpl			x
Precision	**14/17=82%**	**13/14=93%**	**13/17=76%**
Recall	**16/19=84%**	**15/16=94%**	**13/22=60%**

Fig. 5. Results for Xerces-J

We noticed that our technique does not have a bias towards the detection of specific anomaly types. In Xerces-J, we had an almost equal distribution of each antipattern (14 SCs, 13 Blobs, and 13 FDs). On Gantt, the distribution was not as balanced, but this is principally due to the number of actual antipatterns in the system. We found all four known blobs and nine SCs in the system, and eight of the seventeen FDs, four more than DECOR. In Quick UML, we found three out five FDS; however DÉCOR detected three out of ten FDs.

The detection of FDs by only using metrics seems difficult. This difficulty is alleviated in DÉCOR by including an analysis of naming conventions to perform the detection process. However, using naming conventions leads to results that depend on the coding practices of the development team. We obtained comparable results without having to leverage lexical information. The complete results of our experiments, including the comparison with DÉCOR, can be found in [18].

5.4 Discussion

The reliability of the proposed approach requires an example set of bad code. It can be argued that constituting such a set might require more work than identifying, specifying, and adapting rules. In our study, we showed that by using Gantt or Quick UML or Xerces-J directly, without any adaptation, the technique can be used out of the box and will produce good detection and recall results for the detection of antipatterns for the studied systems.

The performance of this detection was superior to that of DECOR. In an industrial setting, we could expect a company to start with Xerces-J or Quick UML or Gantt, and gradually migrate its set of bad code examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

Another issue is the rules generation process. The detection results might vary depending on the rules used, which are randomly generated, though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions for rules generation. We observed an average precision of 84% for Gantt, 80% for Quick UML and 81% for Xerces-J. Furthermore, we found that the majority of defects detected are found in every execution (69%, 80% and 62% of average recall scores respectively for Gantt, Quick UML and Xerces-J). We consequently believe that our technique is stable, since the precision and recall scores are approximately the same for different executions.

Another important advantage in comparison to machine learning techniques is that our HS algorithm does not need both positive (good code) and negative (bad code) examples to generate rules like, for example, Inductive Logic Programming [19].

Finally, since we viewed the design defects detection problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed our algorithm on a standard desktop computer (Pentium CPU running at 2 GHz with 2GB of RAM). The execution time for rules generation with a number of iterations (stopping criteria) fixed to 500 was less than two minutes (1min49s). This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of used metrics and the size of the base of examples. It should be noted that more important execution times may be obtained than when using DECOR. In any case, our approach is meant to apply mainly in situations where manual rule-based solutions are not easily available.

6. Related Work

Several studies have recently focused on detecting design defects in software using different techniques. These techniques range from fully automatic detection to guided manual inspection. The related work can be classified into three broad categories: metric-based detection, detection of refactoring opportunities, visual-based detection.

In the first category, Marinescu [7] defined a list of rules relying on metrics to detect what he calls design flaws of OO design at method, class and subsystem levels. Erni et al. [20] use metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, *n*-tuples of metrics expressing a quality criterion (e.g., modularity). The main limitation of the two previous contributions is the difficulty to manually define threshold values for metrics in the rules. To circumvent this problem, Alikacem et al. [21] express defect detection as fuzzy rules, with fuzzy labels for metrics, e.g., small, medium, large. When evaluating the rules, actual metric values are mapped to truth values for the labels by means of membership functions. Although no crisp thresholds need to be defined, still, it is not obvious to determine the membership functions.

The previous approaches start from the hypothesis that all defect symptoms could be expressed in terms of metrics. Actually, many defects involve notions that cannot be quantified. This observation was the foundation of the work of Moha et al. [5]. In their DÉCOR approach, they start by describing defect symptoms using an abstract rule language. These descriptions involve different notions, such as class roles and structures. The descriptions are later mapped to detection algorithms. In addition to the threshold problem, this approach uses heuristics to approximate some notions which results in an important rate of false positives. Khomh et al. [4] extended DECOR to support uncertainty and to sort the defect candidates accordingly. Uncertainty is managed by Bayesian belief networks that implement the detection rules of DECOR. The detection outputs are probabilities that a class is an occurrence of a defect type. In our approach, the above-mentioned problems related to the use of rules and metrics do not arise. Indeed, the symptoms are not explicitly used, which reduces the manual adaptation/calibration effort.

In the second category of work, defects are not detected explicitly. They are so implicitly because the approaches refactor a system by detecting elements to change to improve the global quality. For example, in [22], defect detection is considered as an optimization problem. The authors use a combination of 12 metrics to measure the improvements achieved when sequences of simple refactorings are applied, such as moving methods between classes. The goal of the optimization is to determine the sequence that maximizes a function, which captures the variations of a set of metrics [23]. The fact that the quality in terms of metrics is improved does not necessary means that the changes make sense. The link between defect and correction is not obvious, which make the inspection difficult for the maintainers. In our case, we separate the detection and correction phases. In [8], we have proposed an approach for the automatic detection of potential design defects in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. Taking inspiration from artificial immune systems, we generated a set of detectors that characterize different ways that a code can diverge from good practices. We then used these detectors to measure how far the code in the assessed systems deviates from normality.

The high rate of false positives generated by the automatic approaches encouraged other teams to explore semiautomatic solutions. These solutions took the form of visualization-based environments. The primary goal is to take advantage of the human ability to integrate complex contextual information in the detection process. Kothari et al. [24] present a pattern-based framework for developing tool support to detect software anomalies by representing potentials defects with different colors. Later, Dhambri et al. [25] propose a visualization-based approach to detect design anomalies by automatically detecting some symptoms and letting others to the human analyst. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Still, the visualization approach is not obvious when evaluating large-scale systems. Moreover, the information visualized is for the most part metric-based, meaning that complex relationships can still be difficult to detect. In our case, human intervention is needed only to provide defect examples.

7. Conclusion

In this article, we presented a novel approach to the problem of detecting design defects. Typically, researchers and practitioners try to characterize different types of common design defects and present symptoms to search for in order to locate the design defects in a system. In this work, we have shown that this knowledge is not necessary to perform the detection. Instead, we use examples of design defects to generate detection rules. Our study shows that our technique outperforms DECOR [5], a state-of-the-art, metric-based approach, where rules are defined manually, on its test corpus.

By ignoring the detection of specific defect types, we avoid two problems with existing detection techniques. First, the detection of most defects is difficult to automate because

their definitions are expressed informally; second, even with a precise definition, it may be difficult to express these symptoms as rules.
The proposed approach was tested on open-source systems and the results are promising. The detection process uncovered different types of design defects more efficiently than DECOR. For example, for Xerces-J, the average of our precision is 81%. DECOR on the other hand has a combined precision of 67% for its detection of the same set of antipatterns. Furthermore, DECOR needed an expert to define rules, while our results were achieved without any expert knowledge, relying only on the bad structure of Gantt to guide the detection process.
As part of future work, we plan to explore the second step: correction of the detected design defects (refactoring). We also need to extend our base of examples with additional badly-designed code in order to take into consideration more programming contexts.

References

- W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray: Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1st ed. John Wiley and Sons, March 1998.
- M. Fowler: Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley, June 1999.
- N. Fenton and S. L. Pfleeger: Software Metrics: A Rigorous and Practical Approach, 2nd ed. London, UK: International Thomson Computer Press, 1997.
- F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui: A Bayesian Approach for the Detection of Code and Design Smells, n Proc. of the ICQS'09.
- N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur: DECOR: A method for the specification and detection of code and design smells, Transactions on Software Engineering (TSE), 2009, 16 pages.
- H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao: Facilitating software refactoring with appropriate resolution order of bad smells, in Proc. of the ESEC/FSE '09, pp. 265–268.
- R. Marinescu: Detection strategies: Metrics-based rules for detecting design flaws, in Proc. of ICM'04, pp. 350–359.
- Kessentini, M., Vaucher, S., and Sahraoui, H.:. Deviance from perfection is a better criterion than closeness to evil when identifying risky code, in *Proc. of the International Conference on Automated Software Engineering*. ASE'10, 2010.
- Lee, K. S. and Geem, Z. W.: A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice, *Comput Method Appl M*, 194(36-38), 3902-3933, 2005.
- Lee, K. S., Geem, Z. W., Lee, S. H. and Bae, K. W.: The harmony search heuristic algorithm for discrete structural optimization, *Eng Optimiz*, 37(7), 663-684, 2005.
- http://ganttproject.biz/index.php
- http://xerces.apache.org/
- A. J. Riel: Object-Oriented Design Heuristics. Addison-Wesley, 1996.
- Gaffney, J. E.: Metrics in software quality assurance, in *Proc. of the ACM '81 Conference*, ACM, 126-130, 1981.
- M. Mantyla, J. Vanhanen, and C. Lassenius: A taxonomy and an initial empirical study of bad smells in code, in Proc. of ICSM'03, IEEE Computer Society, 2003..
- W. C. Wake: Refactoring Workbook. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- http://www.ptidej.net/research/decor/index_html
- http://www.marouane-kessentini/CSMR11.zip
- Raedt, D.:Advances in Inductive Logic Programming, 1st. IOS Press, 1996.
- K. Erni and C. Lewerentz: Applying design metrics to object-oriented frameworks, in Proc. IEEE Symp. Software Metrics, IEEE Computer Society Press, 1996.
- H. Alikacem and H. Sahraoui: Détection d'anomalies utilisant un langage de description de règle de qualité, in actes du 12e colloque LMO, 2006.
- M. O'Keeffe and M. . Cinnéide: Search-based refactoring: an empirical study, Journal of Software Maintenance, vol. 20, no. 5, pp. 345–364, 2008.
- M. Harman and J. A. Clark: Metrics are fitness functions too. in IEEE METRICS. IEEE Computer Society, 2004, pp. 58–69.
- S. C. Kothari, L. Bishop, J. Sauceda, and G. Daugherty: A pattern-based framework for software anomaly detection, Software Quality Journal, vol. 12, no. 2, pp. 99–120, June 2004.
- K. Dhambri, H. A. Sahraoui, and P. Poulin: Visual detection of design anomalies, in Proc. of CSMR'08. IEEE, 2008, pp. 279–283.
- http://sourceforge.net/projects/quj

Part 3: Model Transformation Correctness

Due to the critical role that model transformations play in software development, validation techniques are required to ensure their correctness. A fault in a transformation can introduce a fault in the transformed model, which, if undetected and not removed, can propagate to other models in successive development steps. As a fault propagates further, it becomes more difficult to detect and isolate. Since model transformations are meant to be reused, faults present in them may result in many faulty models. Several studies have investigated static verification techniques for model transformations. For example, Küster [24], focuses on the formal proof of the termination and confluence of graph transformation. Another alternative is transformation testing where the goal is to find the majority of errors instead of formally proving that all errors are detected. This solution is more adaptable for large-scale projects in industry.

Model transformation testing typically consists of synthesizing a large number of different models as test cases, running the transformation mechanism on them, and verifying the result using an oracle function. In this context, two important issues must be addressed: the efficient generation/selection of test cases and the definition of the oracle function to assess the validity of transformed models. This work is concerned with the latter.

Defining the oracle function for model transformation testing is a challenge [86]. Many problems need to be solved. First, the definition of reference models to compare with the transformation outputs is not obvious [84]. Second, for large models, if the candidate transformation errors are given without any risk quantification, inspecting them could be time and resource-consuming. Finally, transformation errors can have different causes such as transformation logic (rules) or source/ target meta-models. Finally, to be effective, the testing process should allow identification of the error causes.

We propose an oracle function that compares target test cases with a base of examples containing good quality transformation traces and assigns a risk level - which will define the oracle function - to the former based on the dissimilarity between the two, as determined by an artificial immune system-based algorithm. As a result, we no longer need to define an expected model for each test case. Also, the traceability links help the tester to understand the origin of an error, and the detected faults are ordered by degree of risk to help address them.

The previous chapters are concerned with the definition of transformation mechanisms. In the next chapter we detail our proposal to test these mechanisms based on the use of examples. These examples represent good quality of transformation traces and each deviation of the evaluated traces from them is considered to be risky.

Chapter 7: Testing Transformation by Example

7.1 Introduction

One of the major challenges after defining a transformation mechanism is how to validate it. One of the efficient techniques is testing. In this chapter, we describe our solution to the problem of testing model transformation. As mentioned previously, this problem has two steps. The first one is the generation of test cases and is out of the scope in this work. We focus on the second step that consists of defining an oracle function to detect errors. Our solution is based on the use of examples of good transformation traces and on considering each deviation from these examples to be risky. This contribution has been accepted to the Journal of Automated Software Engineering (JASE) [78]. The paper, entitled "Example-based Model-Transformation Testing", is presented in the next section.

7.2 Testing Transformation by Example

Example-based Model-Transformation Testing

MAROUANE KESSENTINI[1], HOUARI SAHRAOUI[1] AND MOUNIR BOUKADOUM[2]

Abstract. A major concern in model-driven engineering is how to ensure the quality of the model-transformation mechanisms. One validation method that is commonly used is model transformation testing. When using this method, two important issues need to be addressed: the efficient generation/selection of test cases and the definition of oracle functions that assess the validity of the transformed models. This work is concerned with the latter. We propose a novel oracle function for model transformation testing that relies on the premise that the more a transformation deviates from well-known good transformation examples, the more likely it is erroneous. More precisely, the proposed oracle function compares target test cases with a base of examples that contains good quality transformation traces, and then assigns a risk level to them accordingly. Our approach takes inspiration from the biological metaphor of immune systems, where pathogens are identified by their difference with normal body cells. A significant feature of the approach is that one no longer needs to define an expected model for each test case. Furthermore, the detected faulty candidates are ordered by degree of risk, which helps the tester inspect the results. The validation results on a transformation mechanism used by an industrial partner confirm the effectiveness of our approach.

Keywords: *Model transformation testing, artificial immune system, traceability*

1 Introduction

Model-Driven Engineering (MDE) aims to provide automated support for the creation, refinement, refactoring, and transformation of software models [1]. One of the major challenges of MDE is to automate these procedures while preserving the quality of the produced models [2]. In particular, efficient techniques and tools for validating model transformations are needed. One of them is model transformation testing [4].

Model transformation testing typically consists of synthesizing a large number of different models as test cases, running the transformation mechanism on them, and verifying the result using an *oracle* function. In this context, two important issues must be addressed: the efficient generation/selection of test cases and the definition of the oracle function to assess the validity of transformed models. This work is concerned with the latter.

Defining the oracle function for model transformation testing is a challenge [3, 22]. Many problems need to be solved. First, the definition of reference models to compare with the transformation outputs is not obvious [3, 4, 5]. Second, for large models, if the candidate transformation errors are given without any risk quantification, inspecting them could be time and resource-consuming [22]. Finally, transformation errors can have different causes such as transformation logic (rules) or source/ target metamodels [23]. Finally, to be effective, the testing process should allow identification of the error causes [22].

The primary contribution of this paper is to generate an oracle function "by example" that addresses the above-mentioned issues. The presented work draws an analogy between the detection of transformation errors and the detection of pathogens in the human body. In the human immune system, the process relies on detecting abnormal conditions; the more abnormal something is, the riskier it is considered. By analogy, we propose an oracle function that compares target test cases with a base of examples containing good quality transformation traces, and then assigns a risk level to the former, based on the dissimilarity between the two as determined by an artificial immune system-based algorithm [15]. Consequently, one no longer needs to define an expected model for each test case, and the traceability links help the tester understand the error origins. Furthermore, the detected faults are ordered by degree of risk to help the tester perform further analysis. For this, a custom tool was developed to visualize the risky fragments found in the test cases in different colors, each related to an obtained risk score.

The proposed approach is illustrated and evaluated with the known case of transforming UML class diagrams (CD) to relational schemas (RS). The choice of CD-to-RS transformation is motivated by the fact that it has been investigated by other means and is reasonably complex; this allows focusing on describing the technical aspects of the approach and comparing it with alternatives.

The remainder of this paper is as follows: Section 2 presents the relevant background and the motivation for the presented work; Section 3 describes the AIS-based algorithm; an evaluation of the algorithm with industrial validation is explained and its results are discussed in Section 4; the benefits and also the limitations of the approach are presented in Section 5; Section 6 is dedicated to related work. Finally, concluding remarks and future work are provided in Section 7.

2 Background and Motivation

As showed in Figure 19, a model transformation mechanism takes as input a model to transform, the *source model*, and produces as output another model, the *target model*. The source and target models must conform, respectively, to specific metamodels and, usually, relatively complex transformation rules are defined to insure this.

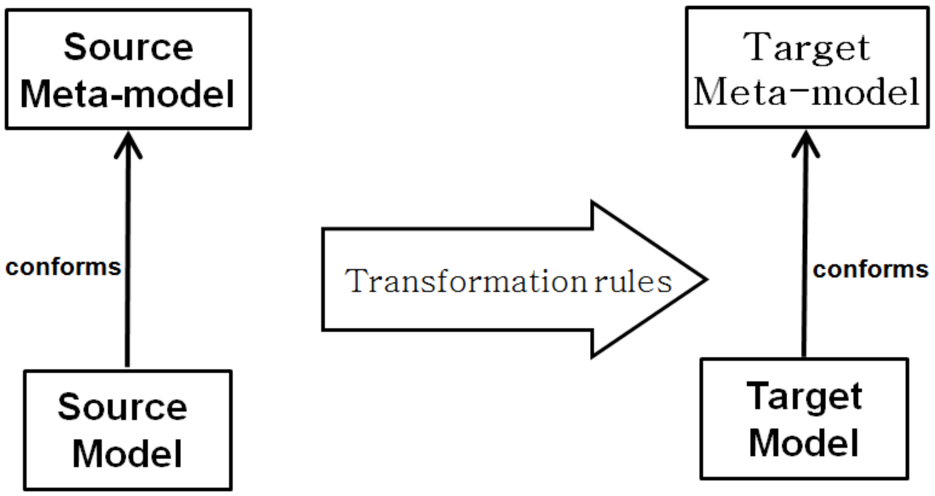


Figure 19 Model transformation mechanism

We can illustrate this definition of the model transformation mechanism with the case of class diagram to relational schema transformation. Figure 20 shows a simplified metamodel of the UML class diagram [24], containing concepts like class, attribute, relationship between classes, etc. Figure 21 shows a partial view of the relational schema metamodel [24], composed of table, column, attribute, etc. The transformation mechanism, based on rules, will then specify how the persistent classes, their attributes and their associations should be transformed into tables, columns and keys.

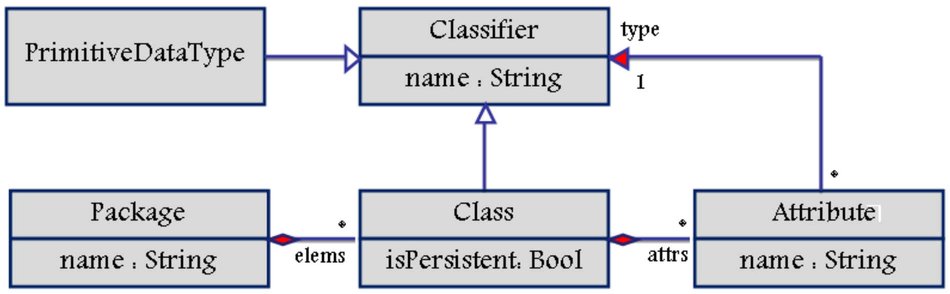


Figure 20 Class diagram metamodel

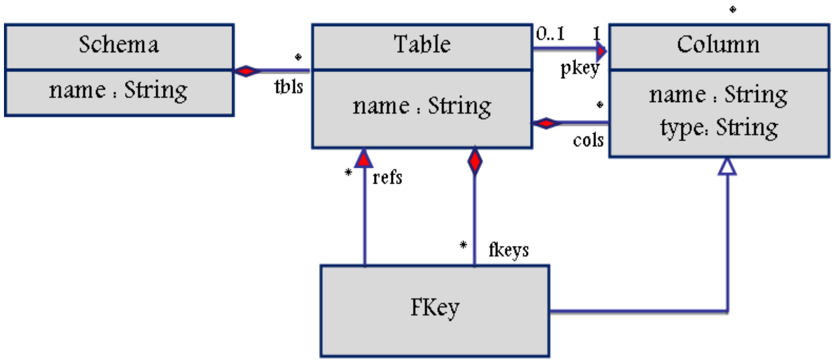


Figure 21 Relational schema metamodel

Once defined, the transformation mechanism needs to be tested to detect potential errors. As described in Figure 22, the basic testing activities consist of designing test cases, executing the model transformation on them, and examining the obtained results [4]. This requires an oracle function that analyzes the validity of the transformed models.

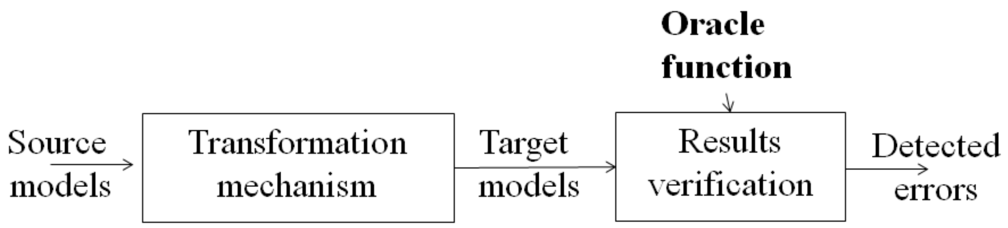


Figure 22 Model transformation testing process

Much work has addressed the automatic generation of test cases [4, 8, 9, 10]. This paper focuses on the complementary issue of defining the oracle function, assuming that a set of test data can be provided. There are many different ways to define this function, depending on the effort provided and the amount of information that is available (formal specification, expected output, etc.) [3]. We distinguish between two main categories of oracle function definitions for model transformation testing: model comparison [5] and specification-conformance checking [25, 22].

For the first category, current MDE technologies and model repositories store and manipulate models as graphs of objects. Thus, when the expected output model is available, the oracle compares two graphs. In this case, the oracle definition problem has the same complexity as the graph isomorphism problem, which is NP-hard [6]. In particular, we can find a test case output and an expected model that look different (contain different model elements) but have the same meaning. So, the complexity of these data structures makes it difficult to provide an efficient and reliable tool for comparison [22]. Still, several studies have proposed simplified versions with a lower computation cost [12]. For example, Alanen et al. [12] present a theoretical framework for performing model differencing. However, they rely on the use of unique identifiers for the model elements.

To illustrate the specification conformance category, we present two contributions: design by contract [25] and pattern matching [22].

For design by contract, the idea is that the transformation of source models into target models is coupled with a contract consisting of pre- and post-conditions. Hence, the transformation is tested with a range of source models that satisfy the pre-conditions to ensure that it always yield target models that satisfy the post-conditions. If the transformation produces an output model that violates a post-condition, then the contract is not satisfied and the transformation needs to be corrected. The contract is defined at the metamodel level and conditions are generally expressed in OCL.

The second method of specification-conformance checking uses patterns that are defined as model fragments, instead of pre-conditions, and for each pattern, a set of post-conditions. Then, the process of pattern matching consists in checking the presence of a pattern in a source model. When a pattern is present, the oracle function evaluates the associated post-conditions on the output model. The difference with design by contract approaches is that both patterns and post-conditions are specified in terms of example of models rather than in terms of metamodel concepts.

Specification-based oracles are difficult to define. Indeed, the number of constraints to define can be very large to cover all transformation possibilities [22]. This is especially the case of contracts related to one-to-many mappings. Moreover, being formal specifications, these constraints are difficult to write in practice [25]. In pattern matching, the constraints are described at the model level and may lead to a fastidious task to define them for each possible instance of the source metamodel [25].

To address the preceding issues, we propose a new oracle definition inspired from the immune system (IS) paradigm that will be described in the next section.

3 Approach

This section describes the principles that underlie the proposed method for model transformation testing. It starts by presenting the metaphor that inspired our work, the artificial immune system (AIS). Then, we provide the details of the approach and our adaptation of the AIS algorithm to the model transformation testing problem.

3.1 Immune System Metaphor

The role of an immune system (IS) is to protect its host organism against harmful disease caused by invaders (pathogens) and/or malfunctioning cells. A biological immune system reacts to adverse environmental changes by ***identifying*** and ***eliminating*** antigens, which are substances or organisms that are recognized by the body as foreign, and which stimulate the immune ***response***. A detailed presentation of the biological immune system is provided in [13]. This paper adapts the first phase of AIS operation to ***identify/detect*** transformation traces that present a high-risk of containing errors, when testing a transformation outcome.

The main task of the immune system is to survey the organism using ***detectors,*** in search of malfunctioning cells and invaders such as bacteria or viruses. Every element that is recognizable by the immune system is called an ***antigen***. The original body cells that are harmless to it are termed ***self*** (or self antigens) while the disease-causing elements are named ***non-self*** (or antigens). The immune system is able to sort them out.

The classification process into self/non-self is complex and produces a large number of randomly created detectors. A ***negative selection mechanism*** eliminates detectors that match the cells in a protected environment where only self cells are assumed to be present. Non-eliminated detectors become naive detectors and die after some time. Furthermore, detectors that do match an antigen are quickly multiplied; this accelerates the response to further attacks. Also, the newly-produced detectors are not exact replicates of each other, with the mutation rate being an increasing function of detector-antigen **affinity** [14].

The elements of the natural immune system that are used in our model transformation testing procedure are mapped as follows.

- **Body**: the transformation mechanism to evaluate.
- **Self-Cells**: model transformation traces without faults.
- **Non-Self Cells** (Antigen): model transformation traces that present a high-risk of having faults.
- **Detector**: example of transformation trace that is very dissimilar to all "clean" traces (self-cells).
- **Affinity**: similarity between a detector and a model transformation trace to evaluate.

The next section presents the principle of our AIS-inspired approach.

3.2 Traceability-based Approach for Model Transformation Testing

We start by describing the overall process of the proposed procedure, illustrating it with the case of class diagram to relational schema transformation. Then, we detail our adaptation of the negative selection algorithm to the model transformation testing problem.

3.2.1 Overview

As showed in Figure 23, our approach can be divided into three important components: the input/output of the testing process, the base of examples, and the main algorithm. We describe these components next.

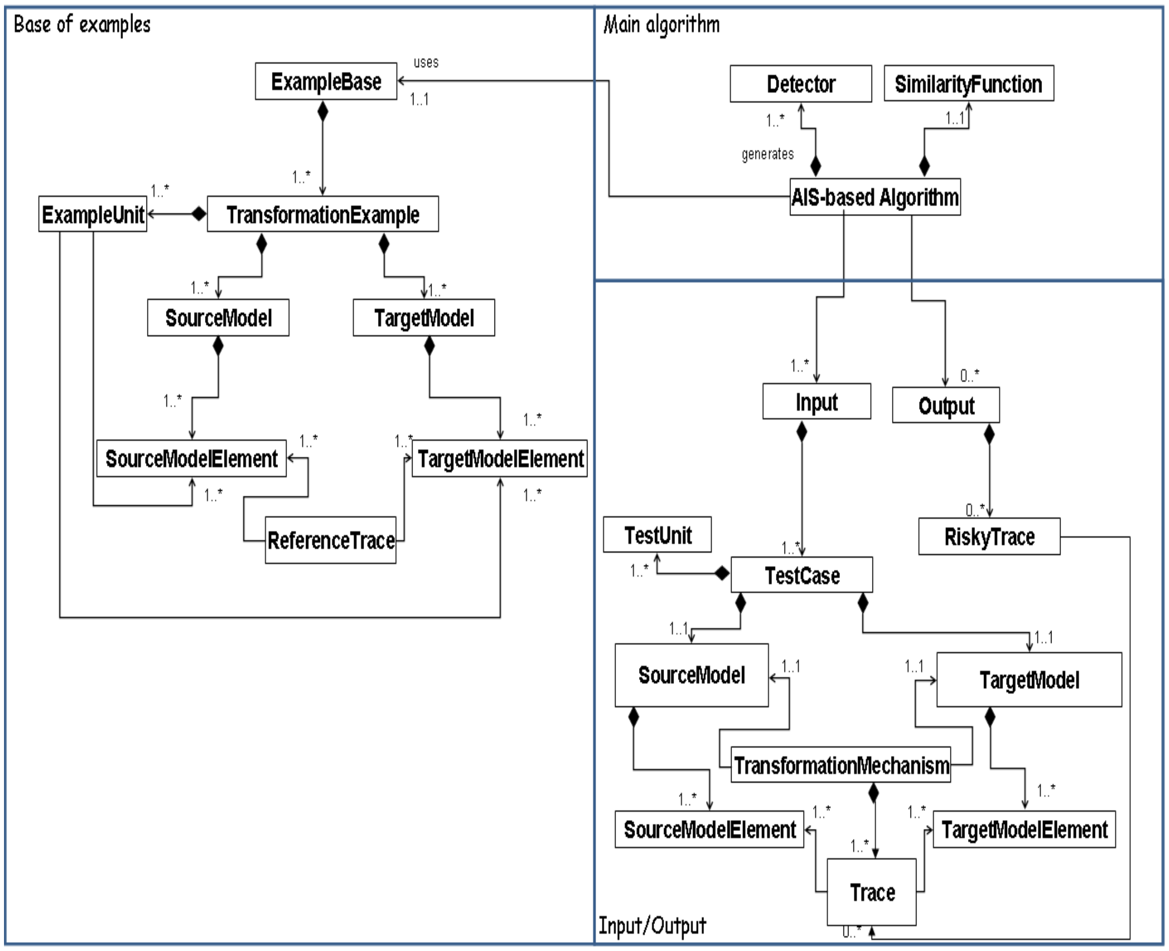


Figure 23 Overall process of our approach

3.2.1.1 Input/Output

The ***Input*** of our testing mechanism is a ***test case*** (TC). A TC includes a ***source model***, its equivalent ***target model*** generated using the ***transformation mechanism*** to test, and the ***traceability links*** between the two models. More formally, a TC is a triple <*SMT, TMT, UT*>, where SMT denotes the source model to test, TMT denotes the generated target model, and UT is a set of ***test unit***s. A **test unit** defines the mappings to produce a particular element in the target model (Thus, there exists one test unit per element). Since a model element (*e.g.*, Table) may contain sub-elements (*e.g.*, Columns), an element test unit also includes the mapping for the sub-elements.

The creation of a database schema from a UML class diagram, as described in the example of Figure 24, is a TC where SMT is the class diagram and TMT is the relational schema generated by the transformation mechanism to evaluate. This TC contains five test units UT that correspond to the number of tables.

To ease manipulation of the test cases, the source and target models are described using a set of predicates that encode the included elements. The predicate types correspond to the different concepts of the source and target metamodels (class, attribute, etc. for class diagrams). The definition of their parameters has to be decided according to the properties and relationships of these concepts. For example, Class *Position* in Figure 24 is described as follows:

Class(Position).

Attribute(Title, Position, String, _).

Attribute(SalaryMin, Position, Int, _).

Attribute(SalaryMax, Position, Int, _).

The first predicate indicates that *Position* is a class, and the second that *Title* is an attribute of that class with a non-unique value ("_" instead of "unique"). The two other predicates describe the remaining two attributes of class *Position*.

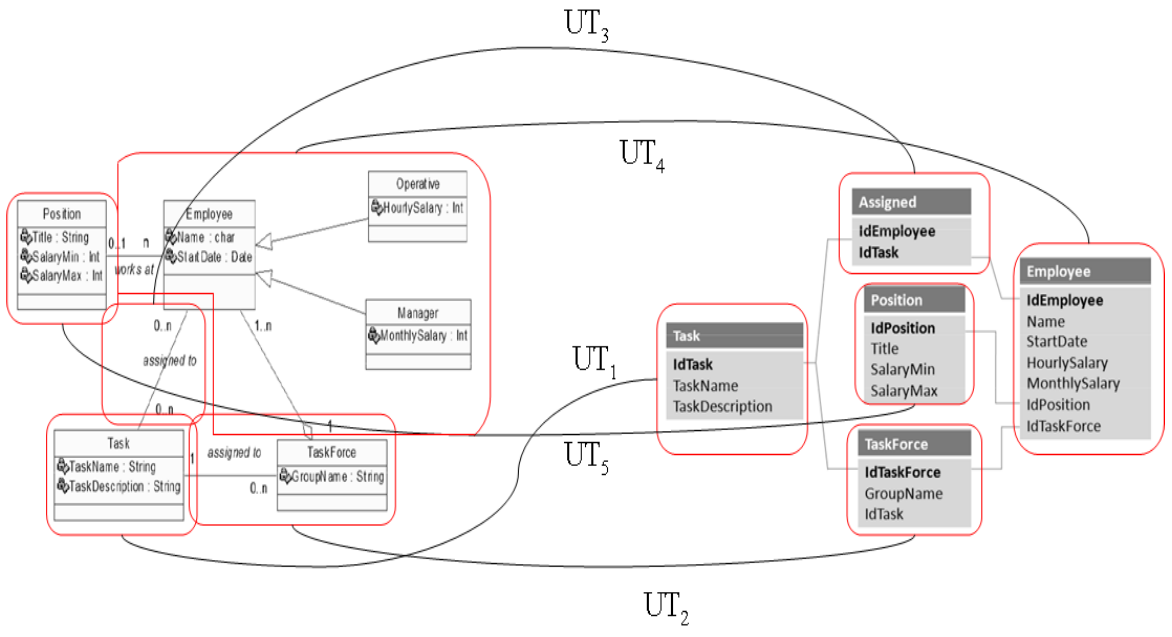


Figure 24 Test case

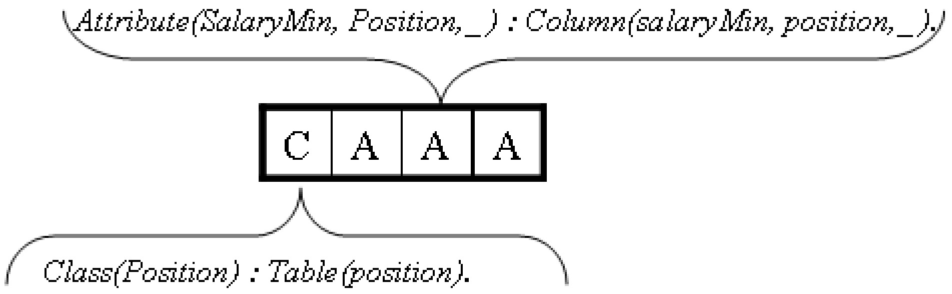

Attribute(SalaryMin, Position,_) : Column(salaryMin, position,_).
C
A
A
A
Class(Position) : Table(position).

Figure 25 Transformation unit coding

The traceability links relate the predicates in the source model to their equivalents in the target model. In our work, these links are automatically generated by adapting an existing metamodel, implemented in Kermeta [32]. An example traceability link that relates an association link to a column is as follows:

Association(0,1, ,n,_ , Position, Employee) : Column(idPosition, employee, fk).

The mappings are specified by the sign ":". For instance, the mapping between *Association(0,1, ,n,_ , Position, Employee)* and *Column(idPosition, employee, fk)* means that the association link between *Position* and *Employee* maps to the primary-foreign key (pfk) *idPosition* in table *Employee.*

The different test units are sets of these mappings. For example, UT_5 is described as follows:

Begin UT5

Class(Position) : Table(position).

Attribute(SalaryMin, Position, Int,_) : Column(idPosition, position, pk),

Column(salaryMin, position,_).

Attribute(SalaryMax, Position, Int,_) : Column(salaryMax, position,_).

Attribute(Title, Position,String,_) : Column(title, position,_).

End UT5

Each test unit can be viewed as a sequence (string) composed of the following predicate types: class (C), attribute (A), method (M), generalization (G), aggregation (F), and association (S). For example, in Figure 25, we present U_5 as the sequence of predicates CAAA, which corresponds to the transformation of a class with three attributes.

The sequence of predicates must follow the specified order of predicate types (C, A, M, G, F, S) to ease the comparison between predicate sequences. When several predicates of the same type exist, we order them according to their parameters. For example, if a class contains several attributes, the corresponding predicates are ordered by considering first the uniqueness, and then the types. In the example of class *Position*, as all the attributes are not unique, the predicates of SalaryMin and SalarityMax (*Int*) appear before the one of Title (*String*).

The ***output*** of our transformation mechanism is a set of test units containing ***risky traces,*** *i.e. traces* with potential transformation errors. Their risk score is determined by an AIS-based algorithm based on dissimilarity with the base of examples. These two components of our approach are described in the next subsections.

3.2.1.2 Base of Examples

The **base of examples** (BE) is composed of a set of ***transformation examples*** (TE). A transformation example is a mapping of **model elements** from a source model to a target model. Similar to a test case, a TE is essentially made of transformation **units.** Thus, it is a triple <*SME, TME, UE*>, where SME denotes the **source-model** example, TME denotes the corresponding **target model**, and UE is a set of **example units** that relate model elements in SME to their equivalents in TME. The definition of a transformation example is similar to that of a test case, and the same predicates representation is used, as described above. However, the target model and the test units of TC are generated by the transformation mechanism whereas those of TE exist independently from the mechanism to test.

3.2.1.3 Main Algorithm

Figure 26 gives the overview of our **AIS-based algorithm**. The detection process has two main steps: **detector generation** and **risk estimation** (similarity function). Detectors are a set of units generated from those in the base of examples. These units define the reference for good transformation traces. The detector generation process is accomplished by using a heuristic search that simultaneously maximizes the difference between the detectors and the units, and between the detectors themselves. The same set of detectors could be used to evaluate different transformation mechanisms based on different formalisms, and it could be updated as the base of examples grows.

The second step of the detection process consists of comparing the test case units to all the detectors. A test case unit that shows similarity with a detector is considered to be risky; the higher the similarity, the riskier the test case unit is. Both detector generation and risk estimation steps use similarity scores. Before detailing the two steps, we describe the similarity function used in this work.

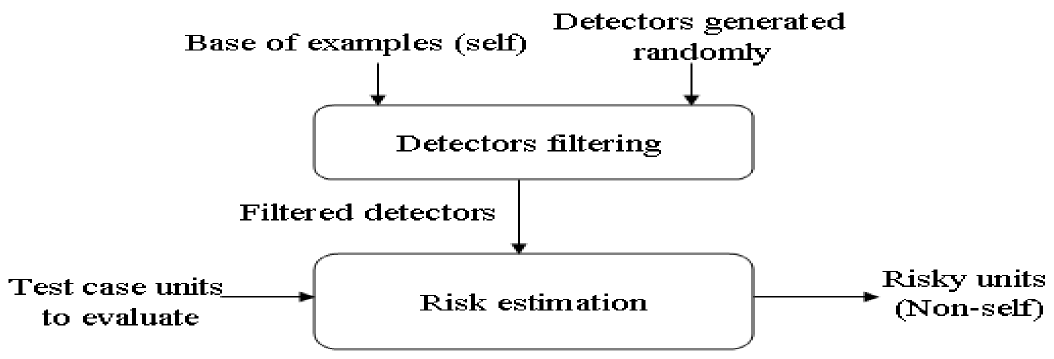


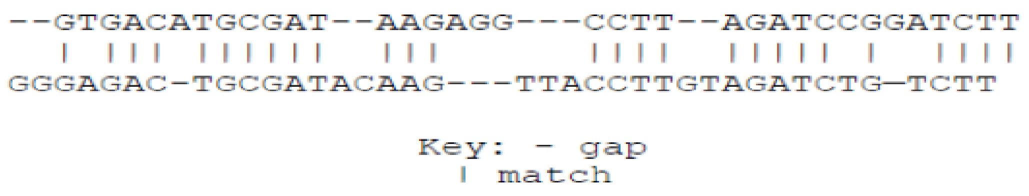
Figure 26 AIS-based algorithm overview

3.2.2 AIS-based Algorithm

In this section, we start by explaining how to determine the similarity between two units. The resulting similarity score is used for detector generation and risk estimation as described later.

3.2.2.1 Similarity between Transformation Units

To calculate the similarity between two units, we adapted to our context a dynamic programming algorithm used in bioinformatics to find similar regions between two sequences of DNA, RNA or proteins: the Needleman-Wunsch alignment technique [27]. Figure 27 provides an illustration of the algorithm.



```
--GTGACATGCGAT--AAGAGG---CCTT--AGATCCGGATCTT
  | ||| ||||||   |||       ||||   ||||| |   ||||
GGGAGAC-TGCGATACAAG---TTACCTTGTAGATCTG—TCTT

                Key: - gap
                 | match
```

Figure 27 Global alignment algorithm [27]

The Needleman-Wunsch global alignment algorithm recursively updates a matrix S of similarity scores for already-matched sub-sequences. The dimensions of S are set by the lengths of the sequences to align. For two sequences a = (a_1,...,a_n) and b = (b_1,...,b_m), S is of dimensions n x m, and each of its element $s_{i,j}$ corresponds to the best alignment score for sub-sequences of a and b, a_i to b_j, of lengths i et j, respectively, considering the previously aligned elements of the sequences. The algorithm can introduce gaps (represented by ”-”) to improve sub-sequence matching. The number of introduced gaps corresponds to the number of times that the maximum value for each line in the matrix is not in the diagonal. The alignment algorithm depends on the predicate order in the sequences, hence the precise order that is described in the previous section.

The algorithm operates as follows: If a gap is inserted in a or b, it introduces a penalty of g in the similarity assessment (see below). In our adaptation, we choose the widely-used value of 1 for the penalty g [27]. Then the algorithm attempts to match the predicates of each pair of sub-sequences a_i and b_j, by using a similarity function $sim_{i,j}$ to return the reward or cost of matching a_i to b_j, and the similarity score for a_i and b_j is updated. Formally, $s_{i,j}$ is defined as follows :

$$s_{i,j} = Max\begin{cases} s_{i-1,j} - g & \text{//insert gap for } b_j \\ s_{i,j-1} - g & \text{//insert gap for } a_i \\ s_{-1,j-1} + sim_{i,j} & \text{//match} \end{cases}$$

Where $s_{i,0} = 0$ and $s_{0,j} = 0$.

Our adaptation of the Needleman-Wunsch algorithm is straightforward. We simply assign a value to g and a way to measure similarity between individual predicates to derive $\mathrm{sim}_{i,j}$. Since our model description uses predicate logic, we define a predicate-specific function to measure similarity. First, if the types differ, the similarity is 0. Since we manipulate sequences of predicates, and not strings, $\mathrm{sim}_{i,j}$ behaves as a predicate-matching function PM_{ij} that measures the sought similarity in terms of the parameters of predicates p_k and q_k associated to the different characters of a_i and b_j. This similarity is the ratio of common parameters in both predicates. Formally, $\mathrm{sim}_{i,j}$ is defined as follows:

$$sim_{ij} = \frac{PM_{ij}}{\max(|a_i|, |b_j|)}$$

where,

$$PIM_{ij} = \sum_{k=1}^{\max(|a_i|,|b_j|)} \frac{\text{number of equivalent predicates parametres}(\mathrm{p_k}, \mathrm{q_k})}{\max(|\mathrm{p_k}|, |\mathrm{q_k}|)}$$

The similarity between sequences a and b is obtained by normalizing this absolute measure $s_{n,m}$ with respect to the maximum of their lengths n and m:

$$Sim(a,b) = \frac{s_{n,m}}{\max(n,m)}$$

To illustrate the use of the global alignment algorithm, consider the evaluation of test unit UC_5 described previously, based on its similarity to unit UE_{15} taken as an example unit (reference traces). UE_{15} is defined as follows:

Begin UE15

Class(Teacher) : Table(Teacher).

Attribute(Level, Teacher, String,_) : Column(Level, Teacher,_).

Attribute(Name, Teacher, String,_) : Column(Name, Teacher,_).

Generalization(Person, Teacher) : Column(IDTeacher, Person, _).

End UE15

	C	A	A	A
C	**1**	0	0	0
A	1	**1.66**	1	1
A	1	**1**	**2.66**	1
G	1	1	**1**	**2.66**

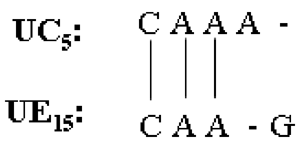


Figure 28 Best sequence alignment between U_5 and T_{15}

Using the sequence coding described in Subsection 3.2.1.1, the predicate sequence for UC_5 is CAAA and the one for UE_{15} is CAAG. The alignment algorithm finds the best sequence alignment as shown in Figure 28. There are three matched predicates between UC_5 and UE_{15}: one class (C), and two attributes (A). If we consider the second matched predicates *Attribute(Title, Position, String,_) : Column(idPosition, position, pk), Column(title, position,_)* from UC_5 and *Attribute(Level, Teacher, String,_) : Column(Level, Teacher,_)* from UE_{15}, their matching corresponds to element (2, 2) in the matrix. The attribute predicates (and their parameters) are similar, but not the transformation of these attributes since we do not have a primary key created in the second trace. The resulting similarity is consequently (1+1+0)/3=0.66, and this value is added to the maximum of elements (1, 2), (1, 1) and (2, 1) which is 1. Thus, the value of the matching is 1.66.
In our example, we have after normalization:

$$Sim(UC_5, UE_{15}) = s_{4,5}/\max(4,4) = 2.66/4 = 0.65.$$

3.2.2.2 Detectors Generation

This section describes how a set of detectors is produced starting from the base of examples. The generation is inspired by the work of Gonzalez and Dasgupta [29], and follows a genetic algorithm [28]. The idea is to produce a set of detectors that best covers the possible deviations from the base of examples. As the set of possible deviations can be very large, its coverage may require a huge number of detectors, which is infeasible in practice. For example, pure random generation was shown to be infeasible in [29] for performance reasons.

We therefore consider detector generation as a search problem. A generation algorithm should seek to optimize the following two objectives:

- Maximize the generality of the detector to cover the non-self by minimizing the similarity with the self.
- Minimize the overlap (similarity) between detectors.

These two objectives define the cost function that evaluates the quality of a solution and, then, guides the search. The cost of a solution D (set of detectors) is evaluated as the average cost of the included detectors. We derive the cost of a detector d_i as an average between the scores of the lack of generality and the overlap, respectively. Formally, we have:

$$\cos t(d_i) = \frac{LG(d_i) + O(d_i)}{2}$$

The lack of generality is measured by the matching score $LG(d_i)$ between the predicate sequence of a detector d_i and those of all units UE_j in the base of examples (BE). It is defined as the average value of the alignment scores $Sim(d_i, UE_j)$ between d_i and units UE_j in BE:

$$LG_{d_i} = \frac{\sum_{UE_j \in BE} Sim(d_i, UE_j)}{|BE|}$$

Similarly, the overlap O_i is measured by the average value of the individual $Sim(d_i, d_j)$ between detector d_i and all the other detectors d_j in solution D:

$$O_i = 1 - \frac{\sum_{d_j, j \neq i} Sim(d_i, d_j)}{|D|}$$

The preceding cost function is used in our genetic-based search algorithm. Genetic algorithms (GA) implement the principle of natural selection [28]. Roughly speaking, a GA is an iterative procedure that generates a population of individuals from the previous generation using two operators, crossover and mutation. Individuals having a high fitness have higher chances to reproduce themselves (by crossover), which improves the global quality of the population. To avoid falling in local optima, mutation is used to randomly change individuals. Individuals are represented by chromosomes containing a set of genes.

For the particular case of detector generation, we use the predicate sequences as chromosomes, with each predicate representing a gene. We start by randomly generating an initial population of detectors. The size of this population will be discussed in Section 4. It is maintained constant during the evolution. The fitness of each detector is evaluated by the inverse function of cost.

The fitness determines the probability of being selected for crossover. We implement the selection process using a wheel-selection strategy [28]. In fact, for each crossover, two detectors are selected by applying the wheel selection twice. Even though detectors are selected, crossover only happens with a certain probability. Sometimes, based on a set probability, no crossover occurs and the parents are directly copied to the new population.

The crossover operator allows creating two offspring o_1 and o_2 from the two selected parents p_1 and p_2. We used the 1-point crossover procedure, defined as follows:

- A random position k, is selected in the predicate sequences.
- The first k elements of p_1 become the first k elements of o_1. Similarly, the first k elements of p_2 become the first k elements of o_2.
- The remaining elements of, respectively, p1 and p2 are added as second parts of, respectively, o_2 and o_1.

For instance, if k = 2, p1 = ***CC***AAGS and p2 = CA***AAS***, then o1 = ***CCAAS*** and o2 = CAAAGS.

The mutation operator consists of randomly changing the traceability links associated to some characters. For example, we change a trace that transforms a class to table by another one that transforms an association link to a table.

3.2.2.3 Risk Estimation

The second step for detecting a potential transformation error is risk assessment. Since the test units are also represented by predicate sequences, each sequence is compared to the detectors obtained in the previous step by using the alignment algorithm. The risk for potential errors associated to test unit UC$_i$ is defined as the average value of the alignment scores Sim(UC$_i$, d$_j$), obtained by comparing UC$_i$ to respectively all the detectors of a set D. Formally,

$$risk_{UC_i} = \frac{\sum_{d_j \in D} Sim(UC_i, d_j)}{|D|}$$

By using the previous definition, the test units can be ranked according to their risks of containing potential transformation errors.

4 Evaluation

To evaluate our approach, we conducted an experiment with industrial data. We start this section by presenting the two kinds of transformation errors we considered in this study. Then we describe our experimental setting. Finally, we report and discuss the obtained results.

In addition to our oracle performance, we evaluate the impact of the example base size on transformation error detection quality. Furthermore, we show how a human tester can easily validate the detected faults using our visualization tool. Finally, we discuss the benefits and limitations of the proposed approach to model transformation testing.

4.1 Considered Transformation Errors

We considered errors belonging to the two following categories:

4.1.1 Metamodel Coverage

This type of error occurs when the transformation is defined without a complete coverage of the metamodel elements. This leads to the problem that parts of some input models cannot be transformed. To illustrate metamodel coverage errors, consider the class diagram metamodel presented in Figure 20. Figure 29 shows a class diagram instance that conforms to this metamodel. Suppose that the transformation mechanism does not include rules transforming the metamodel element *Association*. When executing the transformation mechanism, we have these two incomplete traces:

Association(payable_by, Command, Bill, 1..n, _) : _

Association(pays, Client, Bill, 1, _) : _

However, in our base of examples all association links have corresponding transformations. Thus, one of the generated detectors has an example of this faulty trace. The result is that this trace will be considered to be risky.

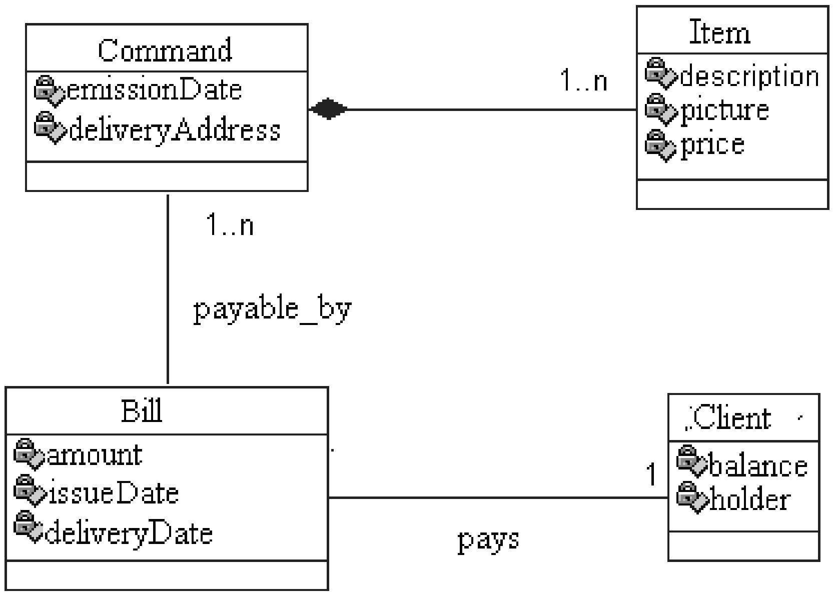


Figure 29 Transformation input: class diagram

4.1.2 Transformation Logic Errors

These errors happen when the transformation, or part of it, is not implemented correctly. This can lead to models that do not conform to the target metamodel. This includes constraints violation. For example, an important constraint in relational models is that each table should have a primary key. Consider a transformation with a rule that maps attributes to columns and another rule that maps unique attributes to primary keys. If we consider class Bill in Figure 29, this does not contain a unique attribute. We end-up then with a table without a primary key:

Class(Bill) : Table(Bill).

Attribute(Amount, Bill,_) : Column(Amount, Bill,_).

Attribute(IssueDate, Bill,_) : Column(IssueDate, Bill,_).

Attribute(DeliveryDate, Bill,_) : Column(DeliveryDate, Bill,_).

However, in our base of examples, all tables have primary keys. Thus, one of the generated detectors has an example of this faulty trace. Thus this trace will be considered to be risky.

4.2 Experimental Setting

We used 12 examples of CD-to-RS transformations, provided by an industrial partner acting in the beverage industry, to build an example base EB = {<SME_i, TME_i, UE_i> | $1 \leq i \leq 12$ }. This company decided to migrate all its existing applications to distributed ones (intra-web) with a common database. As a result, different database schemas had to be generated from the existing applications written in object-oriented code. To this end, the development and maintenance department started by reverse-engineering these projects to class diagrams. Then they transformed the obtained diagrams to relational schema using a commercial tool. In a third step, they completed and corrected the schemas manually.

The projects we obtained from the company are related to three application domains: product management, marketing, and fleet management including geolocalization. For each transformation example, we had the class diagram and the manually corrected relational schema. After receiving the examples, we inspected them manually to ensure that they were free of transformation errors.

As Table 1 shows, the size of class diagrams varies from 28 to 92 elements, with an average of 58. Altogether, the 12 examples defined 193 test units corresponding to the number of tables in the 12 schemas (ref. section 3).

We selected as transformation mechanism to test, MTIP, a tool written in Kermeta [34]. Kermeta implements a state-of-the-art declarative model transformation language suitable for Model-Driven Development (MDD) and data transformation. It is implemented as an Eclipse plugin that leverages the Eclipse Modelling Framework (EMF) to handle models based on MOF, UML2, and XML Schema. The transformation traces are collected automatically by adapting an existing metamodel in Kermeta [32].

We used a 12-fold cross validation procedure. For each fold, we manually introduced different transformation errors into the transformation mechanism (rules) and subsequently transformed one of the 12 examples (test case <SMT_k, TMT_k, UT_k>). The 11 remaining ones formed the base of examples for the testing ({<SME_j, TME_j, UE_j> | $j \neq k$ }). Thus, each fold concerned one different example. The test units were ranked by order of risk, and those that were reported to have a risk higher than 0.75 were checked for correctness. The correctness of our testing method was based on precision and recall capabilities assessments. These were defined as follows:

$$\text{Precison} = \frac{\text{number of true positive transform ation errors}}{\text{total number of detected transform ation errors}}$$

$$\text{Recall} = \frac{\text{number of true positive transformation errors}}{\text{total number of actual transformation errors}}$$

Are considered as true positive all units that have a risk higher than 0.75 and that were actual errors. For our experiment, we randomly generated 50 detectors (about a quarter of the number of existing units in the base of examples) with a maximum size of 15 predicates (ref. section 3).

4.3 Transformation Errors Detection Results

As showed in Table 1, the riskiest test units detected by our approach contained transformation errors in all folds of the validation procedure. The measured average precision was 91%, with most errors detected with at least 82% precision. The measured average recall of 98% was greater, indicating that nearly all the errors were detected. For over half the total number of folds, 100% recall was obtained, indicating the detection of all expected errors. Furthermore, the precision and recall scores were not correlated with the size of the source model.

Table 1. 12-fold cross validation

Source Model	Number of elements	Number of transformation errors introduced manually	Precision	Recall
SM1	72	13	82%	93%
SM2	83	14	93%	94%
SM3	49	11	92%	100%
SM4	53	16	88%	100%
SM5	38	9	90%	100%
SM6	47	12	100%	100%
SM7	78	16	84%	95%
SM8	34	8	100%	100%
SM9	92	14	82%	93%
SM10	28	9	100%	100%
SM11	59	13	93%	100%
SM12	63	15	94%	100%

Average	58	12	91%	98%

We also investigated the types of transformation errors that were identified. As mentioned previously, the possible error sources were during specification of the model transformation mechanism: (i) the metamodels; (ii) the transformation logic (rules). Table 2 shows that, for fold SM5, chosen because it represent the average size and precision/recall scores, our affinity function (risk score) can be a good estimator for detecting transformation errors. In fact, the units located at the top of the list are all true positive, and the unique incorrect (unexpected) detected error is located last. Furthermore, the units containing two kinds of errors are typically detected with higher risk values (UC_{68} and UC_{69}). The same observations can be drawn for all folds, showing that the used risk score offers an effective and efficient manner for the tester to validate the detected errors.

An important consideration is the impact of the example base size on transformation error detection quality. Drawn for SM5, the results of Figure 30 show that our approach had good precision in situations where only few examples were available. As the results shows, the precision score seems to follow an exponential curve: it rapidly grows to acceptable values and then slows down. First, it improved from 22% to 75% as the example base size increased from 1 to 6 examples. Then, it only grew by an additional 18% as the size went from 6 to 11 examples.

Table 2. Errors detected in SM5

Test units with numbers	**Risk**	**Met-model error**	**Transformation logic error**
UC_{68}	0.93	X	X
UC_{69}	0.91	X	X
UC_{70}	0.96		X
UC_{71}	0.91	X	
UC_{72}	0.89		X
UC_{73}	0.94		X

UC_{74}	0.96	X
UC_{75}	0.89	X
UC_{76}	0.77	

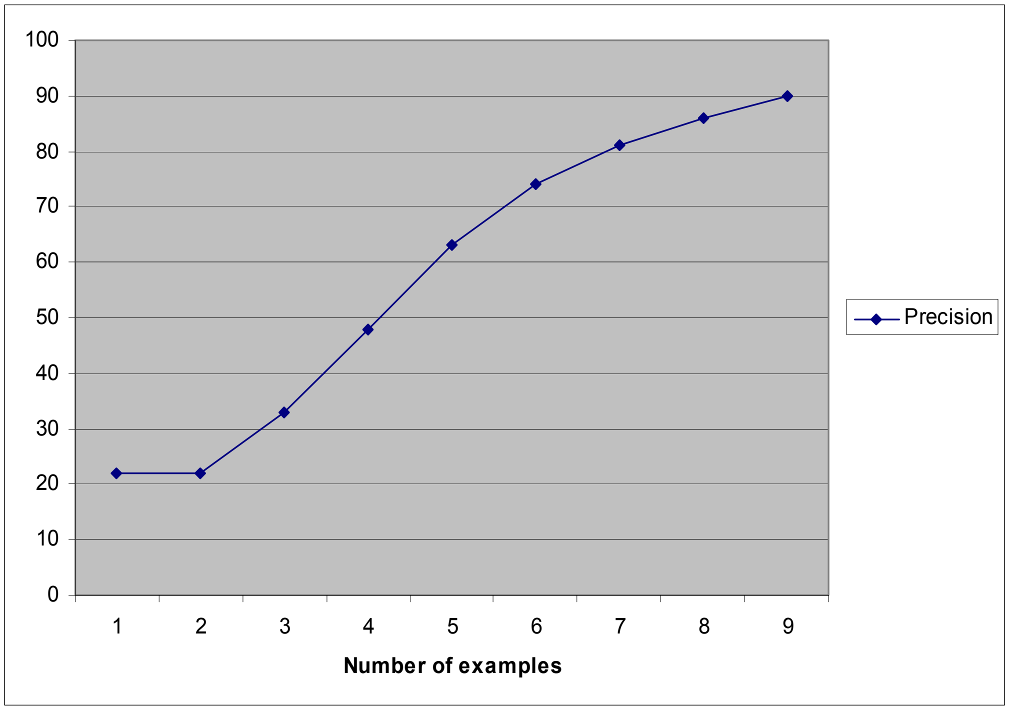


Figure 30 Example-size variation

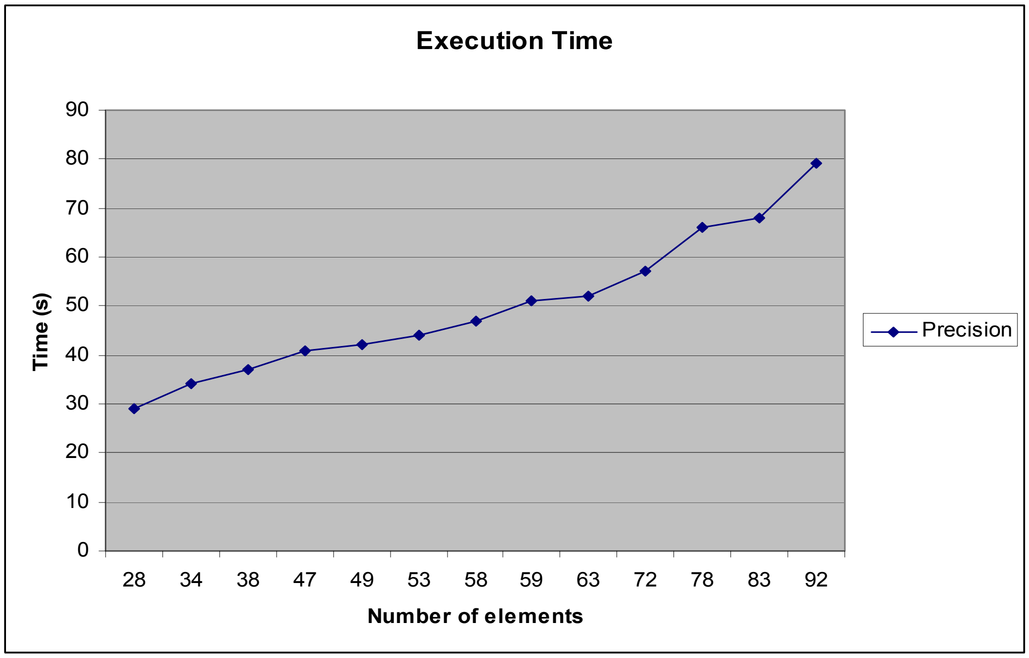
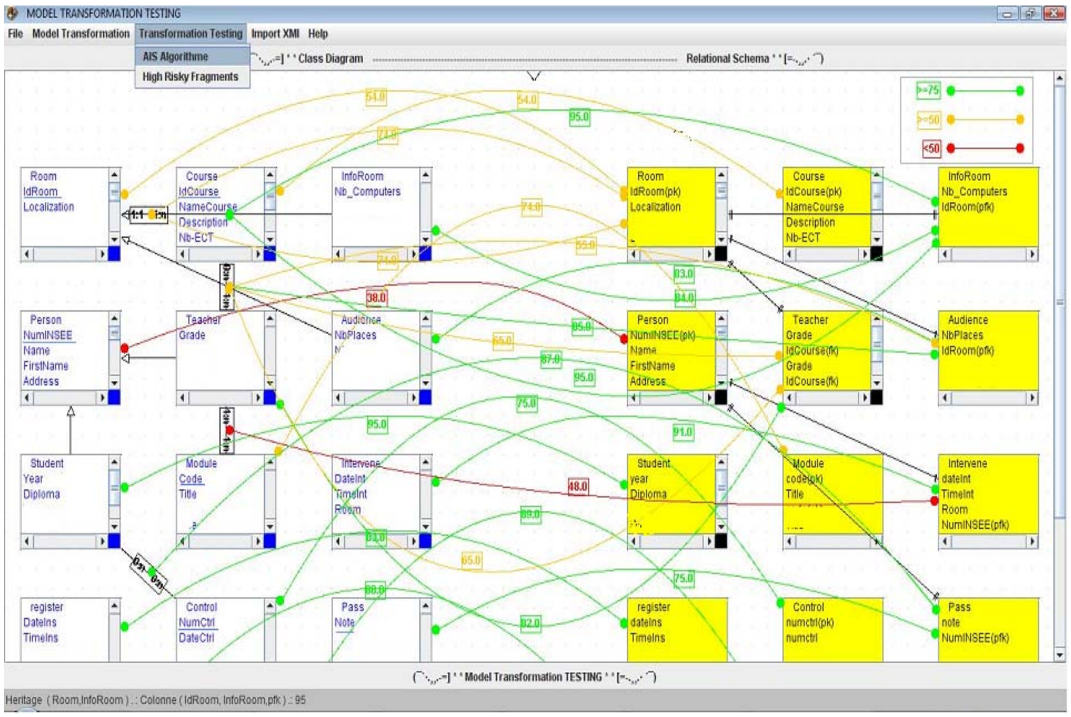


Figure 31 Execution time

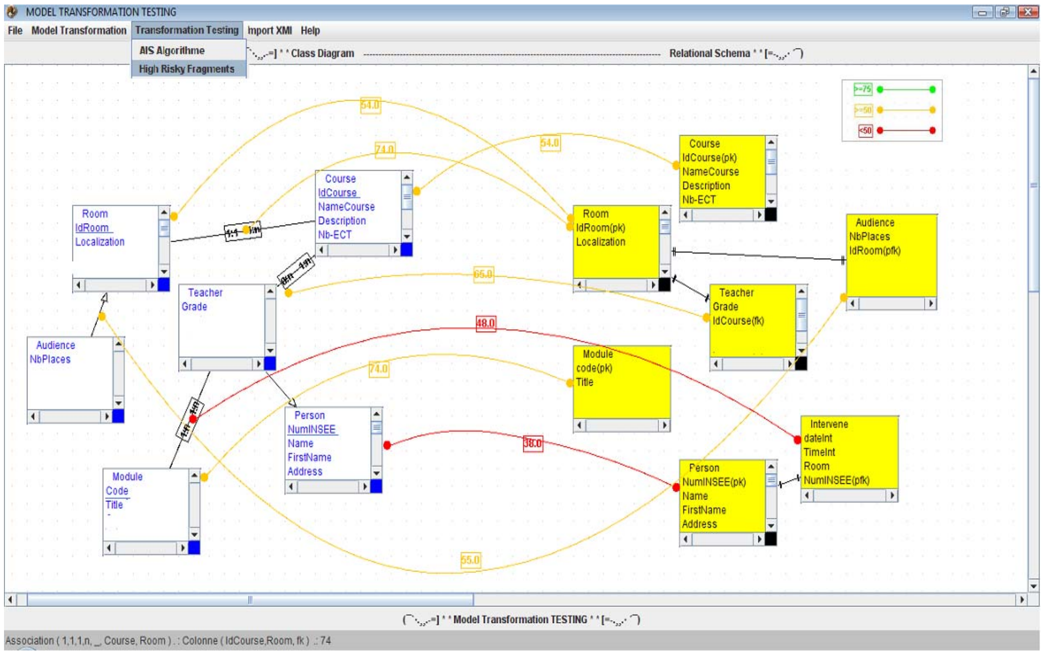
We executed our algorithm on a standard desktop computer (Pentium CPU running at 2 GHz with 1GB of RAM). The execution time is shown in Figure 31. As suggested by the curve shape, the time increased linearly with the number of elements. Thus, our approach appears to be scalable from the performance standpoint. Only a few seconds were needed to test the transformation mechanism to evaluate. This execution time does not include that for detector generation since the detectors are only generated once and can serve to evaluate several transformation mechanisms afterwards. This feature is a major advantage of using detectors versus comparing the test units to all units in the base of examples, which can be infeasible in time when the number of units is very large [15].

As showed in Figure 32, a human tester can analyze the detected risky test units with a graphical visualization tool. We developed a custom utility that displays the risky test units with different colors related to the obtained risk score, and with the “clean” traces colored in green. The human tester can validate, for example, only units that present a potential risk that are colored in red. Furthermore, the traces help the tester understand the origin of an error. To allow dealing with the transformation of large models, the traces can be viewed at different levels of granularity. For example, the tester can only show the links between model elements, or between sub-elements. Furthermore, he can only visualize the traces having potential risk (Figure 32-b).



Heritage (Room,InfoRoom) : : Colonne (IdRoom, InfoRoom,pfk) : 95

(a)



Association (1,1,1,n, _ Course, Room) : : Colonne (IdCourse,Room, fk) : 74

(b)

Figure 32 Interactive transformation errors detection using our tool: (a) all traces and (b) only risky traces

5 Discussion

In this section, we discuss several issues concerning the detection of transformation errors. Especially, we describe some advantages and limitations related to our approach.

In our approach, there is no need to define an expected model for each test-case or to define pre- and post-conditions as oracles; we only use similarity to good transformation examples. The approach can be seen to propose a way to detect and order transformation errors by importance, using a risk score. Moreover, our oracle definition is independent from the transformation mechanism to evaluate or the source/target formalisms, and it helps the tester understand the origin of errors by visualizing the traceability links with different colors.

Still, our approach has issues that need to be addressed. First, its performance depends on the availability of good transformation examples, which could be difficult to collect. Second, the assumption that the base of examples does not contain transformation errors may be too strong, and not easily verified in practice. On the positive side, our results show that a small number of examples may be sufficient to obtain good testing results. This alleviates the two previous limitations, and may even offer a solution because the number of needed examples is small. It consists of generating a few test cases using the transformation mechanism to test and, then, of manually detecting and correcting potential transformation errors. The resulting cases then form the base of examples.

To reduce the number of necessary examples, these examples are decomposed into units. However, the definition of units sometimes depends on the source/target metamodels of the test-case. Thus, our proposed methodology could sometimes be dependent on the source/target metamodels, but this potential dependency is acceptable in comparison to the state of the art that will be discussed in the next section.

Another potentially important aspect of our detection technique is the generation of a sufficient number of detectors. In our experiments, we generated 50 detectors, which corresponds to a quarter of the units present in the base of examples. We evaluated the precision of our approach when varying the number N_d of detectors, with N_d= {20, 50, 90, 120}. Our results, shown in Figure 33, reveal that precision stops improving when the number of detectors is higher than the quarter of the total number of units in the base of examples. In addition, Figure 34 shows the execution time necessary to generate different numbers of detectors. We observe that this time appears to vary linearly with respect to the diagram sizes for all the number of detectors. In conclusion, our experimentation results indicate that a reasonable number of detectors (quarter of the transformation units in the base of examples), generated in less than one minute, is sufficient to obtain good detection results.

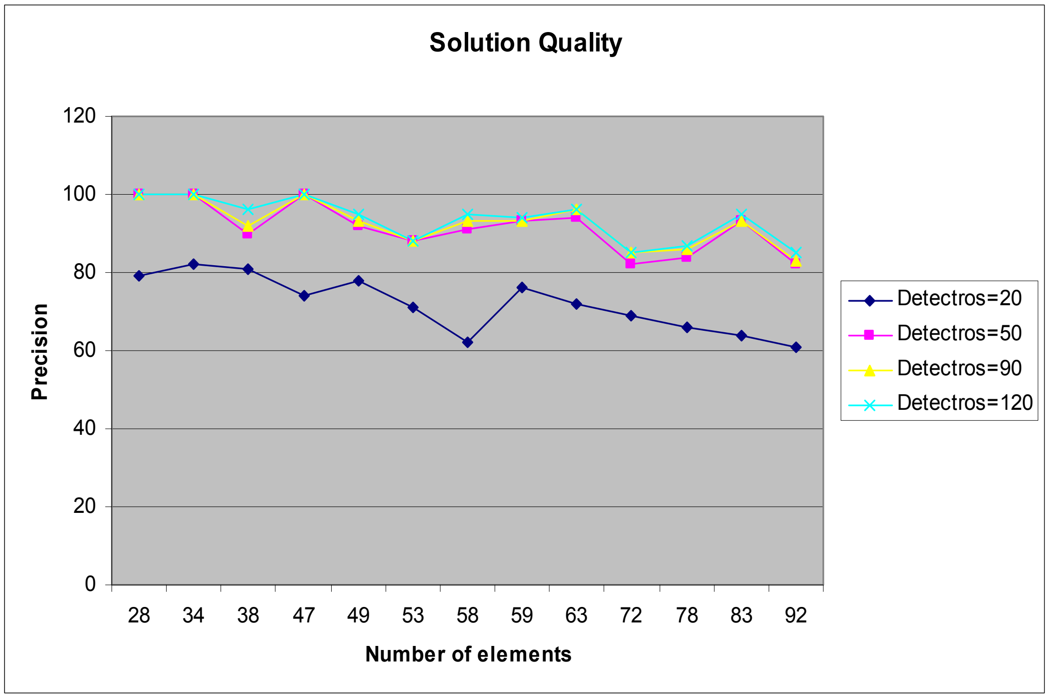


Figure 33 Detectors variation vs solution quality (precision)

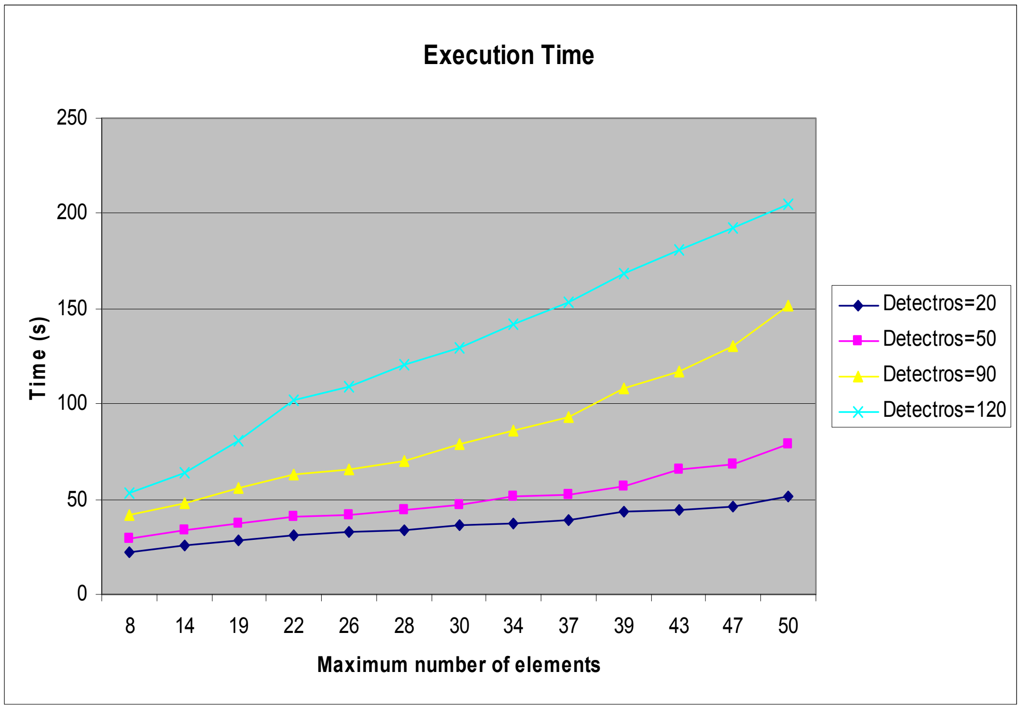


Figure 34 Detectors variation vs. execution time

An additional issue is the selection of interesting detectors since the detection results might vary depending on which detectors are used, and ours were randomly generated (though guided by a meta-heuristic). To ensure that our results are relatively stable, we compared the results of multiple executions for detector generation. We found that approximately the same transformation errors are found after every execution and the differences only exist for low-risk test units. We therefore believe that our technique is stable with regard to detector choice since the result variability only relates to the least risky classes.

6 Related Work

The work proposed in this paper crosscuts many research topics. In the remainder of this section, we present representative contributions in five of these topics: test-case generation, oracle function definition, search-based testing, by-example model transformation, and traceability and transformation.

6.1Test Case Generation

Fleurey et al. [10, 44] and Steel et al. [16] discuss the reasons why testing model transformation is distinct from testing traditional implementations: the input data are models that are complex in comparison to simple-type data. Both papers describe how to generate test data in MDA by adapting existing techniques, including functional criteria [10] and bacteriologic approaches [9]. Lin et al. [4] propose a testing framework for model transformation, built on their modeling tools and transformation engine, that offers a support tool for test case construction, test execution and test comparison; but the test models are manually developed in their work. As our work does not address test case generation, it can be integrated with the previous approaches without the need to define the expected model for each test case.

One of the most widely-used techniques for test-case generation is mutation analysis. Mutation analysis is a testing technique that aims to evaluate the efficiency of a test set. Mutation analysis consists of creating a set of faulty versions, or mutants, of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. Mottu et al. [35] have adapted this technique to evaluate the quality of test cases. They introduce some modifications in the transformation rules (program-mutant). Then, using the same test cases as input, an oracle function compares the results (target models). If all the results are the same, we can assume that the input cases were not sufficient to cover all the transformation possibilities. In our work, the goal is not to evaluate the quality of a data set but to propose a generic oracle function to detect transformation errors. Our oracle function compares between some potential errors (detectors) and transformation traces to evaluate. However, in mutation analysis, the oracle function compares between two target models, one generated by the original mechanism (rules) and another after modifying the rules. In addition, our technique does not create program variations (rules modifications) but traces variation that differs from good ones. We modified the transformation mechanism to introduce errors artificially only to validate our approach. Finally, the mutation analysis technique needs to define an expected model for each test case in order to compare it with another target model obtained from the same test case after modifying the rules (mutant).

Some other approaches are specific to test case generation for graph-transformation mechanism. Küster [61], addresses the problem of model transformation validation in a way that is very specific to graph transformation. He focuses on the verification of transformation rules with respect to termination and confluence. His approach aims at ensuring that a graph transformation will always produce a unique result. Küster's work is concerned with the verification of transformation properties rather than the validation (testing) of their correctness. Darabos et al. [25] investigate the testing of graph transformations. They consider graph transformation rules as the transformation specification and propose to generate test data from this specification. Their technique focuses on testing the pattern matching activity that is considered the most critical of a graph transformation process. They propose several faulty models that can occur when performing the pattern matching as well as a test-case generation technique that targets those particular faults. Compared to our approach, Darabos' work is specific to graph-based transformation testing. Sturmer et al. [22] propose a technique for generating test cases for code generators. The criterion they propose is based on the coverage of graph transformation rules. The generated test cases consider both individual rules and rule interactions. Sampath et al. [[11]] propose a similar method for the verification of model processing tools such as simulators and code-generators. They use a method that generates test-cases for model processors starting from a metamodel. This method, like the previous contributions, is concerned with test-case generation which is not the goal of our contribution.

6.2 Oracle Function Definition

Mottu et al. [3] describe six different oracle functions to evaluate the correctness of an output model. These six functions can be classified in the three categories discussed in Section 2. Thus, they are completely different from our proposal.

In [8], the authors suggest to manually determine the expected outcome of the transformation and compare it with the actual outcome of the transformation by using a simple graph-comparison algorithm, since the compared models conform to the same metamodel. While this makes model transformation testing feasible, our view is that manually constructing the expected outcome is not an efficient and scalable approach.

Varró et al. [33] have developed a formal framework for describing model transformation. The formal framework relies on models represented as typed attributed graphs. Concerning the transformation correctness, they have developed an approach based on planner algorithms to prove the syntactic correctness of a transformation. Syntactic correctness refers to the property that the result of a transformation corresponds to a certain previously specified syntax, and can be achieved by specifying a graph grammar for both the source and target languages.

More generally, when many test models are necessary, writing an oracle for each test case is time consuming and error prone. Generic oracles are more interesting since they are written only once, and could be used with all the test cases. Another limitation of the existing approaches is that they consider a particular model transformation technique and use its specificities to validate the corresponding transformation mechanisms. This has the advantage of having specific validations but make these approaches difficult to adapt to other transformation techniques. For our approach, the oracle function is generic and independent from the transformation techniques. Moreover, we do not have an explicit specification of the transformation mechanism to evaluate (properties, constraints, or contracts).

6.3 Search-based Testing

Our approach is inspired by contributions in the domain of Search-Based Software Engineering (SBSE) [39]. SBSE uses search-based approaches to solve optimization problems in software engineering. Once a software engineering task is framed as a search problem, many search algorithms can be applied to solve that problem. Search-based techniques are have been used for problems in software testing [40, 41, 42]. Especially, genetic algorithms have been extensively used for test data generation. The general idea behind the proposed approaches is that possible test suites define a search space and that a test adequacy criterion is coded as a fitness function. This later guides the selection of the best test suite in this space. A wide variety of testing problems have been targeted using search techniques, including structural, functional and non functional testing, safety testing, mutation testing, integration testing and exception testing [42]. In our work, we use a genetic algorithm with a completely different perspective. Indeed, the idea is to generate artificial situations that are different from known good-transformation traces. Then, these artificial traces are used not as test cases but as oracle functions.

To our knowledge, there exist very few works in software engineering that use an AIS techniques. The closest one to our work proposes a software defect prediction model by means of an artificial immune recognition system (AIRS) along with correlation-based feature selection (CFS) [30]. In our work, in addition to target a different problem, we do not use AIRS, but the negative selection algorithm.

6.4 By Example Model Transformation

The AIS approach proposed in this paper is based on using examples. Various such by-example approaches have been described in the literature [17, 18, 19, 20, 31, 43]. The most similar one is Model Transformation By Example (MTBE), which was proposed in [18, 31]. Varrò and Balogh [17] propose a semi-automated process for MTBE using Inductive Logic Programming (ILP). The principle of their approach is to derive transformation rules semi-automatically from an initial prototypical set of interrelated source and target models. In a previous work [18, 31, 43] we proposed MOTOE (MOdel Transformation as Optimization by Example), a novel approach to automate model transformation using heuristic-based search. MOTOE uses a set of transformation examples to derive a target model from a source model. The transformation is seen as an optimization problem where different transformation possibilities are evaluated and a quality associated to each one depending on its conformance with the examples at hand. A similar approach to MTBE, called Model Transformation By Demonstration (MTBD), was proposed in [20]. Instead of the MTBE idea of inferring the rules from a prototypical set of mappings, users are asked to demonstrate how the model transformation should be done, through direct editing (e.g. add, delete, connect, update) of the source model so as to simulate the transformation process.

In conclusion, when compared to existing by-example approaches, our proposal appears to present the first contribution that uses examples for model transformation testing.
Despite these efforts in MTBE work, and considering the nature of the algorithms that are used, there is no evidence that a solid base of examples can generate target models without errors.

6.5 Traceability and Transformation

In our approach, the definition of transformation examples is based on traceability [33]. Traceability usually allows tracing artifacts within a set of chained operations, where the operations may be performed manually (e.g. crafting a software design for a set of software requirements) or with automated assistance (e.g., generating code from a set of abstract descriptions). Most work on traceability in MDE uses it for detecting model inconsistency and fault localization in transformations. In our proposal, the goal is not to generate traces but to use clean trace information as input in order to detect transformation errors.

7 Summary

In this article, we presented a new oracle function definition for model transformation testing that does not need to define the expected model for each test case. The technique is based on the metaphor of a biological immune system using negative selection. We propose an oracle function that compares between the targeted test cases and a base of examples containing good quality transformation traces and assigns a risk level, which will define the oracle function to the former based on the dissimilarity between the two. Furthermore, we use a custom tool to help the human tester visualize the detected risky fragments in test cases, using different colors related to the obtained risk scores.
We illustrated our approach with a transformation mechanism for UML class diagrams to relational schemas. In this context, we conducted a validation with real industrial models. The experiment results clearly indicated that the detected risky fragments (transformation errors) are comparable to those detected by a human tester (precision and recall of more than 90%).

Our method also suffers from some limitations as discussed in section 5. In particular, our oracle function may require considerable effort to find and collect transformation examples. Future work should validate our approach with more complex transformation mechanisms like sequence diagram to colored Petri nets in order to conclude about the general applicability of our methodology. Also, in this paper, we only looked at the first step of immune systems: the detection of risk. The second step is problem correction. The colonal selection algorithm [20] could be adapted for finding the best immune response, i.e. the one corresponding to the optimal sequence of corrections to apply for correcting errors by automatically regenerating some rules from examples.

References

1. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. ICSE 2007 : Future of Software Engineering. (2007)
2. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOSPLA 2003, Anaheim, USA (2003)
3. Mottu, J.M., Baudry, B., Traon, Y.L.: Model transformation testing: Oracle issue. In Proc. of ICST08.
4. Y. Lin, J. Zhang, and J. Gray. A Testing Framework for Model Transformations, in Model-driven Software Development. 2005, Springer.
5. Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In Proc. GaMMa, 2006.
6. Samir Khuller and Balaji Raghavachari. Graph and network algorithms. ACM Computing Surveys, pages 43{45, March 1999.
7. F. Azuaje, "Review of artificial immune systems: a new computational intelligence approach" by l.n. de castro and j.timmis (eds) springer, london, 2002.
8. Brottier, E., Fleurey, F., Steel, J., Baudry, B., and Traon, Y. L. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In Proceedings of SSRE 2006
9. B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. L. Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to . net components. In *ASE* 2002.
10. Fleurey, F., J. Steel and B. Baudry, Validation in Model-Driven Engineering: Testing Model Transformations, In *15th IEEE International Symposium on Software Reliability Engineering*. 2004
11. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for uml design models. Technical report, Computer Science Department, Colorado State University, 2006
12. Alanen, M. and I. Porres. Difference and Union of Models. in UML'03, USA.
13. Kuby, Janis, Thomas J Kindt, Barbara A Osborne, & Richard A Goldsby (1997) "Immunology," 3 rd ed, WH Freeman & Co, New York.

14. D. Dasgupta, Z. Ji, and F. Gonzalez, “Artificial immune system (ais) research in the last five years.” in IEEE Congress on Evolutionary Computation (1). IEEE, 2003, pp. 123–130.
15. S. Forrest, A. S. Perelson, L. Allen, and R. C. Kuri, “Self nonself discrimination in a computer,” in Proceedings of the 1994 IEEE Symposium on Resarch in Security and Privacy.
16. Steel, J. and M. Lawley. Model-Based Test Driven Development of the Tefkat Model- Transformation Engine. In ISSRE'04, pp. 151-160, 2004. IEEE.
17. D. Varro and Z. Balogh, Automating Model Transformation by Example Using Inductive Logic Programming, ACM Symposium, 2007 (SAC 2007).
18. M. Kessentini, H.Sahraoui and M.Boukadoum Model Transformation as an Optimization Problem. In Proc.MODELS 2008, pp. 159-173 Vol. 5301 of LNCS. Springer, 2008.
19. M. Wimmer, M. Strommer, H. Kargl, and G. Kramler.Towards model transformation generation by-example. HICSS-40 Hawaii International Conference on System Sciences.
20. Yu Sun, Jules White, and Jeff Gray, "Model Transformation by Demonstration," *MoDELS09*
21. W. Pang and G. M. Coghill, “Modified clonal selection algorithm for learning qualitative compartmental models of metabolic systems,” in GECCO ’07.
22. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. L. Traon. Model Transformation Testing Challenges. In *IMDT workshop*, 2006.
23. J. Kuster and M. Abd-El-Razik. Validation of model transformations- first experiences using a white box approach.In *MoDeVa’06.*
24. J. Bezivin, F. Jouault, and P. Valduriez. On the need for megamodels. In OOPSLA/GPCE 2004 Workshop, 2004.
25. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the Specification of Model Transformation Contracts. *Proceedings of Workshop OCL and MDE*, 2004.
26. Solberg, R. Reddy, D. Simmonds, R. France, and S. Ghosh. Developing Service Oriented Systems Using an Aspect-Oriented Model Driven Framework., 2006.
27. Carrillo H. and Lipman D. The multiple sequence alignment problem in biology. SIAM Journal on Applied Mathematics, 48(5):1072-1082, 1988.
28. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.
29. F. Gonzalez, D. Dasgupta, Anomaly detection using real-valued negative selection, Genetic Programming and Evolvable Machines (2003) 383–403.
30. Catal, C. and Diri, B. 2007. Software defect prediction using artificial immune recognition system. In *Proceedings of IASTED international Multi-Conference*, 285-290,2007
31. M. Kessentini, A.Bouchoucha, H.Sahraoui and M.Boukadoum. Example-based Sequence Diagram to Colored Petri Nets Transformation Using Heurisitc Search. In Proc.ECMFA 2010, Springer, 2010.
32. Jean-Remi Falleri, Marianne Huchard, Clementine Nebut: Towards a traceability framework for model transformations in Kermeta In: Proceedings of the European Conference on MDA Traceability Workshop, Bilbao, Spain (2006)

33. Varró, D., Pataricza, A.: Automated formal verification of model transformations. In: Jürjens, J., Rumpe, B., France, R., Fernandez, E.B. (eds.) CSDUML 2003: critical systems development in UML; proceedings of theUML'03 workshop, Technical Report, pp. 63–78. Technische Universität München, September 2003
34. Bézivin, J., Rumpe, B., Schürr, A., Tratt, L.: MTIP workshop. Available from: http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf (2005)
35. Mottu, J.-M.,Baudry, B.,LeTraon,Y.: Mutation analysis testing for model transformations. In: Proceedings of ECMDA'06 (European Conference on Model Driven Architecture). Bilbao, Spain (2006)
36. Küster, J.M.: Definition and validation of model transformations. Softw. Syst. Model. 5(3), 233–259 (2006)
37. Darabos, A., Pataricza, A., Varro, D.: Towards testing the implementation of graph transformations. In: Proceedings of GT-VMT Workshop Associated to ETAPS'06, pp. 69–80. Vienna, Austria (2006)
38. P. Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar. Testing model-processing tools for embedded systems. In IEEE Real-Time and Embedded Technology and Applications Symposium, pages 203– 214, 2007.
39. M. Harman, The Current State and Future of Search Based Software Engineering, In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), 20-26 May, Minneapolis, USA (2007)
40. A. Baresel, D. W. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In International Symposium on Software Testing and Analysis (ISSTA 2004), pages 108–118, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume29, Number 4.
41. A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pages 1329–1336.
42. P. McMinn. Search-based software test *data* generation: A survey. Software Testing, Verification and Reliability, 14(2):105–156, June 2004.
43. M. Kessentini, H.Sahraoui and M.Boukadoum Search-based Model Transformation by example. In Journal of Software and System Modeling, 2010 DOI: 10.1007/s10270-010-0175-7
44. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.: Qualifying input test data for model transformations. In: Software and Systems Modeling (2008)

Chapter 8: Conclusions

In this chapter, we summarise the results and conclusions of the dissertation. We also discuss opportunities for extending our work.

8.1 Contributions

The main objective of this thesis was to define approaches for automating model and code transformation, and for testing model transformation. We defined as requirements the following properties: (1) The automatic model and code transformation should not require a lot of knowledge (rule definition, exhaustive design defect list, etc); (2) Data and mechanisms used for transformations also should be used to test the correctness of these transformations; and (3)The transformation mechanism should be adaptable to different software artefacts and transformation categories.

The first contribution of the thesis is about defining a new exogenous transformation mechanism that does not require rules definition. This contrasts with the majority of the available work on model transformation that is based on the hypothesis that transformation rules exist and that the important issue is how to express them. In our solution, the transformation process is seen as an optimization problem where different transformation possibilities are evaluated and, for each possibility, a quality is associated depending on its conformance with the examples at hand. The search space is explored using two methods. In the first one, we use Particle Swarm Optimization (PSO) with transformation solutions generated from the examples at hand as particles. Particles progressively converge toward a good solution by exchanging and adapting individual construct transformation possibilities. In the second method, a partial run of PSO is performed to derive an initial solution. This solution is then refined using a local search with simulated annealing (SA). We have illustrated our approach with the transformation of static models (UML class diagrams to relational schemas) and dynamic ones (UML sequence diagram to colored petri net). In this context, we conducted a validation with real industrial models. The experiment results clearly indicate that the derived models are comparable to those proposed by experts (correctness of more than 90% with manual evaluation). They also reveal that some constructs were correctly transformed although no transformation examples were available for them. This was possible because the approach uses syntactic similarity between construct types to adapt their transformations. We also showed that the two methods used for the space search produced comparable results when properly applied, and that PSO alone is enough with small-to-medium models while the combination PSO-SA is more suitable when the size of the models to transform is larger. For both methods, our transformation process derives a good quality transformation in an acceptable execution time. Finally, the validation study showed that the quality of MT improves with the number of examples. However, it reaches a stable score after as few as nine examples.

The second contribution is about endogenous transformation. We have proposed two principale solutions.

In the first one, we proposed a new detection mechanism for design defects detection. The solution is based on the metaphor of biological immune systems using negative selection theory. As with immune systems, the technique does not look for design that follows specific definitions, but rather for abnormal designs. By ignoring the detection of specific defect types, we avoid two problems with existing detection techniques. First, we do not need to code informal specifications into rules. Second, we do not have to cover exhaustively the set of possible defects. To evaluate our approach, we used classes from the JHot-Draw library as our examples of well-designed and implemented code. Two systems, Xerces-J and Gantt, were then analyzed using our approach. Almost all the identified riskiest classes (precision> 90%) were found in a list of classes tagged as defects (blobs, spaghetti code and functional decomposition) in DECOR [88].

In the second solution, we proposed an automated approach for design defect detection rules generation. It exploits an algorithm that automatically finds rules for the detection of possible design defects, thus relieving the designer from doing so manually. Our algorithm derives rules in the form of metric/threshold combinations, from known instances of design defects (defect examples). Due to the large number of possible combinations, we use a music-inspired heuristic that finds the best harmony when combining metrics. We evaluated our approach on finding potential defects in three open-source systems (Xerces-J, Quick UML and Gantt). The detection process uncovered different types of design defects more efficiently than DECOR. For example, for Xerces-J, the average of our precision is 81%. DECOR on the other hand has a combined precision of 67% for its detection of the same set of antipatterns.

After defining a transformation mechanism, one of the challenges is how to validate it. One of the efficient techniques is testing based on an oracle function to detect errors. We presented an approach that does not need to define an expected model for each test case and also without specifying constraints to evaluate. Our oracle function compares target test cases with a base of examples containing good quality transformation traces, and then assigns a risk level to the former, based on dissimilarity between the two. The traces help the tester to understand the origin of an error. We illustrated our approach with a transformation mechanism for UML class diagrams to relational schemas. In this context, we conducted a validation with real industrial models. The experiment results clearly indicated that the detected risky fragments (transformation errors) are comparable to those detected by a human tester (precision and recall of more than 90%).

8.2 Limitations and Future Research Directions

In this section, we discuss some limitations and open research directions related to our proposal. First, all our performance contribution depends on the availability of examples, which could be difficult to collect. However, as we have shown in the experiments, only few examples are needed to obtain good results. Second, due to the nature of our solution, i.e., an optimization technique, the process could be time consuming for large models. Furthermore, as we use heuristic algorithms, different executions for the same input could lead to different outputs. This can be a disadvantage for some model-driven engineering applications, e.g., when model transformation is a deterministic process and the generated target model is unique. Nevertheless, having different and equivalent output models is close to what happens in the real world where different experts may propose different target models.

Different future work directions can be explored. The application of new search-based techniques like artificial immune system to model evolution or model refactoring is challenging. We are working on an extension of our first contribution about exogenous transformation by example. The idea is to generate transformation rules from examples using heuristic search. Our approach starts by randomly generating a set of rules, executing them to generate some target models. Then, it evaluates the quality of the proposed solution (rules) by comparing the generated target models to the expected ones in the base of examples. In this case, the search space is large and heuristic-search is needed.

We are actually working to extend our proposal to other problems. A new technique for predicting "buggy" changes, when modifying an existing version of a model, can be proposed. The idea is to classify the changes as clean or not. The Change classification determines whether a new model change is more similar to prior "buggy" or clean changes in the base of examples. In this manner, change classification can predict the existence of "bugs" in models changes.

Furthermore, we are working on transformation composition using examples. We propose a solution based on a music-inspired approach. We draw an analogy between the transformation composition process and finding the best harmony when composing music. Say, for example, that we have a transformation mechanism M1 that transforms formalism T1 into T2, but the meta-model of T2 evolved into T3, after deleting or adding elements. We want to generate new transformation rules that transform T1 into T3. The idea is to compose two transformation mechanisms T1 to T2 and T2 to T3. To this end, we propose to view transformation rules generation as an optimization problem where rules are automatically derived from available examples. Each example corresponds to a source model and its corresponding target model, without transformation traces from T1 to T3. Our approach starts by composing a set of rules (T1 to T2 and T2 to T3), executing them to generate some target models, and then evaluating the quality of the proposed solution (rules) by comparing the generated target models and the expected ones in the base of examples.

Finally, we can extend our work related to endogenous transformation. In this thesis, we only looked at the first step of immune systems: the discovery of risk. As part of our future work, we plan to explore the other two steps: identification and correction of detected design defects (refactoring) that corresponds to the code transformation step.

Related Publications

I have started my PhD in January 2008. The following is a list of our publications related to this dissertation.

Articles in Journals

1. Kessentini, M., Sahraoui, H., and Boukadoum, M. 2010. Search-Based Model Transformation by Example, *Software and System Modeling* Journal-Special Issue of MODELS08 (Accepted, to appear)
2. Kessentini, M., Sahraoui, H., and Boukadoum, M. 2010. Example-based Model Transformation Testing, *Automated Software Engineering* Journal (Accepted. To appear)
3. Kessentini, M., Vaucher, S., Sahraoui, H., and Boukadoum, M. 2010. Immune-Inspired Approach for Design Defect Detection , *ACM Transactions on Software Engineering and Methodology* (To be submitted-December 2010)
4. Kessentini, M., Vaucher, S., Sahraoui, H., and Boukadoum, M. 2010. Design Defect Detection Rules Generation Using Genetic Programming , *Automated Software Engineering* Journal (To be submitted-December2010)

Book Chapters

1. Kessentini, M., Sahraoui, H., Boukadoum, M., Faunes, M., and Wimmer, M. 2010. Maintenance, Evolution and Reengineering of Software Models by Example. In ”Emerging Technologies for the Evolution and Maintenance of Software Models” book, edited by Jorg Rech and Christian Bunse (Submitted).

Articles in Refereed Conference:

1. Kessentini, M., Vaucher, S., and Sahraoui, H. 2010. Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code. (**Considered as one of best contributions**). The *25th IEEE/ACM International Conference on Automated Software Engineering ASE2010* (acceptance rate : 18%)
2. Kessentini, M., Sahraoui, H., and Boukadoum, M. 2010. Testing Sequence Diagram to Colored Petri Nets Transformation: An Immune System Metaphor. (**Best paper award**) In Proceedings of the *20th Annual International Conference on Computer Science and Software Engineering (CASCON2010).* (acceptance rate : 26%)
3. Kessentini, M., Sahraoui, H., and Boukadoum, M. 2008. Model Transformation as an Optimization Problem. (**Considered as one of best contributions**) In Proceedings of *the 11th international Conference on Model Driven Engineering Languages and Systems* (2008), 159-173. MODELS08 (acceptance rate : 21%)
4. Kessentini, M., Wimmer, M., Sahraoui, H., and Boukadoum, M. 2010. Generating Transformation Rules from Examples for Behavioral Models. (**Best paper Award**) In Proceedings of *Behavioural Modelling - Foundations and Application* (BM-FA 2010)
5. Kessentini, M., Sahraoui, H., Boukadoum, and M. Wimmer, M. 2011. Design Defects Detection Rules Generation : A Music Metaphor, *15th IEEE European Conference on Software Maintenance and Reengineering* CSMR11 (acceptance rate : 28%)
6. Kessentini, M., Sahraoui, H., Boukadoum, and M. Wimmer, M. 2011. Design Defects Detection by Example. 14th IEEE International Conference

on Fundamental Approaches to Software Engineering FASE 2011 (acceptance rate : 28%)

7. Kessentini, M., Bouchoucha, A., Sahraoui, H., and Boukadoum, M. 2010. Example-Based Sequence Diagrams to Colored Petri Nets Transformation Using Heuristic Search. In Proceedings of the *Sixth European Conference on Modelling Foundations and Applications ECMFA2010* (acceptance rate : 28%)

8. Kessentini, M., Sahraoui, H., and Boukadoum, M. 2009. Transformation de modèle par l'exemple: approche par metaheuristique. Actes du 15e conférence francophone sur les Langages et Modèles à Objets, mars 2009. éditions Cépadues.

Bibliography

[1] Aditya Agrawal and G'{a}bor Karsai and Sandeep Neema and Feng Shi and Attila Vizhanyo. The Design of a

[2] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, Gabor Karsai. 2001. In Proc of Composing Domain-Specific Design Environments. IEEE Computer 34(11). pp 44-51.

[3] Akos Schmidt, Dániel Varró. 2003. "CheckVML: A Tool for Model Checking Visual Modeling Languages", In Proc. UML 2003: 6th International Conference on the Unified Modeling Language, LNCS, vol. 2863, pp. 92-95.

[4] Alex Sellink and Chris Verhoef. 1999. An Architecture for Automated Software Maintenance. In Proceedings of the 7th International Workshop on Program Comprehension (IWPC '99). IEEE Computer Society, Washington, DC, USA, 38-.

[5] Alexander Egyed. 2002. Automated abstraction of class diagrams. *ACM Trans. Softw. Eng. Methodol.* 11, 4 (October 2002), 449-491.

[6] Alexander Franz Egyed. 2000. *Heterogeneous View Integration and its Automation*. Ph.D. Dissertation. University of Southern California, Los Angeles, CA, USA. Advisor(s) Barry William Boehm.

[7] Alexander Repenning and Corrina Perrone. 2000. Programming by example: programming by analogous examples. *Commun. ACM* 43, 3 (March 2000), 90-97.

[8] Anantha Narayanan and Gabor Karsai. 2008. Towards Verifying Model Transformations. *Electron. Notes Theor. Comput. Sci.* 211 (April 2008), 191-200.

[9] Andrea Corradini , Reiko Heckel , Ugo Montanari. 2000. "Graphical operational semantics", In Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques.

[10] Andrea Darabos and Andras Pataricza and Daniel Varro. 2008. Towards Testing the Implementation of Graph Transformations. Electronic Notes in Theoretical Computer Science, pp 211.

[11] Andrea Darabos, Andres Pataricza, and Daniel Varro. 2008. Towards Testing the Implementation of Graph Transformations. *Electron. Notes Theor. Comput. Sci.* 211 (April 2008), 75-85.

[12] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. 2002. Transformation: The Missing Link of MDA. In *Proceedings of the First International Conference on Graph Transformation* (ICGT '02), Andrea Corradini, Hartmut Ehrig, Hans Kreowski, and Grzegorz Rozenberg (Eds.). Springer-Verlag, London, UK, 90-105.

[13] Anneke G. Kleppe, Jos Warmer, and Wim Bast. 2003. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[14] Anneliese Amschler Andrews and Robert B. France and Sudipto Ghosh and Gerald Craig. 2006. Test adequacy criteria for uml design models. Technical report, Computer Science Department, Colorado State University.

[15] ATLAS Group. The Atlantic Zoo. http://www.eclipse.org/gmt/am3/ zoos/atlantic Zoo/.

[16] Behrens, Ulf, Flasinski, Marillsz, Hagge, Lars, Jurek, Janusz, and Ohrenberg, Kars. 1996. Recent Developments of the ZEUS Expert System ZEX. IEEE Transactions on Nuclear Science, Vol 43.

[17] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (PLDI '03). ACM, New York, NY, USA, 141-154.

[18] Benoit Baudry, Dinh-Trong, Trung and Mottu, Jean-Marie and Simmonds, Devon, Robert France, Ghosh, Sudipto, Franck Fleurey and Yves Le Traon. 2006. Model Transformation Testing Challenges. In ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.

[19] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. 2002. Automatic Test Cases Optimization Using a Bacteriological Adaptation Model: Application to .NET Components. In Proceedings of the 17th IEEE international

conference on Automated software engineering (ASE '02). IEEE Computer Society, Washington, DC, USA, 253-.

[20] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. 2002. Automatic Test Cases Optimization Using a Bacteriological Adaptation Model: Application to .NET Components. In Proceedings of the 17th IEEE international conference on Automated software engineering (ASE '02). IEEE Computer Society, Washington, DC, USA, 253-.

[21] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. 2009. Barriers to systematic model transformation testing. Commun. ACM 53, 6 (June 2009), 139-143.

[22] Dániel Varró and Andras Pataricza. 2003. “Automated Formal Verification of Model Transformations”, In Critical Systems Development workshop in UML03: 6th International Conference on the Unified Modeling Language, LNCS, vol. 2863, 63-78.

[23] Dániel Varró and Zoltán Balogh, Automating model transformation by example using inductive logic programming. In *Proceedings of the 2007 ACM symposium on Applied computing* (SAC '07). ACM, New York, NY, USA, 978-984.

[24] Dániel Varró. Model transformation by example. 2006. In *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems* (MoDELS '06), Vol. 4199 of LNCS. Springer, pp. 410–424.

[25] Devon Simmonds, Raghu Reddy, Robert France, Sudipto Ghosh, and Arnor Solberg. 2005. An Aspect Oriented Model Driven Framework. In *Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference* (EDOC '05). IEEE Computer Society, Washington, DC, USA, 119-130

[26] Didier Vojtisek and Jean-Marc Jézéquel. 2004. MTL and Umlaut NG: Engine and Framework for Model Transformation. INRIA Tech.Report http://www.ercim.org/publication/Ercim News/enw58/vojtisek.html.

[27] Don Roberts. 1999. Practical Analysis for Refactoring, Ph.D. thesis, University of Illinois at Urbana-Champaign.

[28] Eclipse. Generative Modeling Technologies (GMT) project, 2006.

[29] Eelco Visser. 2005. A survey of strategies in rule-based program transformation systems. J. Symb. Comput. 40, 1 (July 2005), 831-873.

[30] Eelco Visser. 2005. A survey of strategies in rule-based program transformation systems. J. Symb. Comput. 40, 1, 831-873.

[31] Elliot J. Chikofsky and James H. Cross. 1990. Reverse engineering and design recovery: A taxonomy, IEEE Software, vol. 7, no. 1, pp. 13–17.

[32] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. 2006. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *Proceedings of the 17th International Symposium on Software Reliability Engineering* (ISSRE '06). IEEE Computer Society, Washington, DC, USA, 85-94.

[33] Foutse Khomh, Stéphane Vaucher, Yann-Gael Guéneuc, and Houari Sahraoui. 2009. A Bayesian Approach for the Detection of Code and Design Smells. In Proceedings of the 2009 Ninth International Conference on Quality Software (QSIC '09). IEEE Computer Society, Washington, DC, USA, 305-314.

[34] Franck Fleurey, Jim Steel and Benoit Baudry. 2004. Validation in Model-Driven Engineering: Testing Model Transformations. Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on In Model, Design and Validation, pp. 29-40.

[35] Frédéric Jouault and Ivan Kurtev. 2005. Transforming Models with ATL. In J.-M. Bruel, editor, MoDELS Satellite Events, volume 3844 of LNCS, Springer-Verlag, pages 128–138.

[36] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev. 2008. ATL: A model transformation tool. Science of Computer Programming. 72(1-2), pp 31-39.

[37] Frédéric Jouault. 2005. Loosely coupled traceability for ATL. In: Proceedings of the European Conference on ModelDrivenArchitecture (ECMDA) Workshop on Traceability.

[38] Gabor Karsai. 2010 Lessons Learned from Building a Graph Transformation System. In Proc of Graph Transformations and Model-Driven Engineering 2010, pp 202-223

[39] Gabriele Taentzer. 2003. AGG: A graph transformation environment for modeling and validation of software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, AGTIVE, volume 3062 of LNCS, Springer, pages 446–453.

[40] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. 2005. Visualization-based analysis of quality for large-scale software systems. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05). ACM, New York, NY, USA, 214-223.

[41] Hachmi AliKacem and Houari Sahraoui. 2006. Détection d'anomalies utilisant un langage de description de règle de qualité. in actes du 12e colloque LMO, LMO, Ed.

[42] Hartmut Ehrig and Claudia Ermel. 2008. Semantical Correctness and Completeness of Model Transformations using Graph and Rule Transformation. In: Proc. International Conference on Graph Transformation (ICGT'08). Volume 5214 of LNCS., Heidelberg, Springer Verlag, pp 194-210

[43] http://www.eclipse.org/gmt/.

[44] Ismenia Galvao and Arda Goknil. 2007. Survey of Traceability Approaches in Model-Driven Engineering. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference* (EDOC '07). IEEE Computer Society, Washington, DC, USA, 313-.

[45] Iván García-Magariño, Jorge J. Gómez-Sanz, Rubén Fuentes-Fernández. 2009. Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations* (ICMT '09), Richard F. Paige (Ed.). Springer-Verlag, Berlin, Heidelberg, 52-66.

[46] James Kennedy and Russell C. Eberhart. 1995. Particle swarm optimization. In Proceedings of the IEEE International Conference on Neural Networks, pp. 1942–1948.

[47] Jean-Marie Mottu and Benoit Baudry and Yves Le Traon. 2006. Mutation analysis testing for model transformations. In Model Driven Architecture -

Foundations and Applications, Second European Conference, ECMDA-FA 2006, LNCS 4066, pp 376-390. Springer.

[48] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. 2008. Model transformation testing: oracle issue. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop* (ICSTW '08). IEEE Computer Society, Washington, DC, USA, 105-112.

[49] Jean-Remy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clementine Nebut. 2008. Metamodel Matching for Automatic Model Transformation Generation. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems* (MoDELS '08). Springer-Verlag, Berlin, Heidelberg, 326-340.

[50] Jeff Gray. 2002. Aspect-Oriented Domain-Specific Modeling: A Generative Approach Using a Meta-weaver Framework. Phd, Vanderbilt University.

[51] Jochen M. Kuster and Mohamed Abd-El-Razik. 2006. Validation of model transformations: first experiences using a white box approach. In *Proceedings of the 2006 international conference on Models in software engineering* (MoDELS'06), Thomas Kuhne (Ed.). Springer-Verlag, Berlin, Heidelberg, 193-204.

[52] Jochen M. Kuster. 2006. Definition and validation of model transformations. Software and Systems Modeling, vol 5(3), pp 233–259.

[53] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who should fix this bug?. In *Proceedings of the 28th international conference on Software engineering* (ICSE '06). ACM, New York, NY, USA, 361-370.

[54] Juan de Lara and Hans Vangheluwe. 2002. AToM3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber, editors, FASE, volume 2306 of LNCS, Springer, pages 174–188.

[55] K. Czarnecki and S. Helsen. 2006. Feature-based survey of model transformation approaches. *IBM Syst. J.* 45, 3 (July 2006), 621-645.

[56] Kang Seok Lee and Zong Woo Geem. 2005. A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice, Comput Method Appl M, 194(36-38), 3902-3933.

[57] Karim Dhambri, Houari Sahraoui, and Pierre Poulin. 2008. Visual Detection of Design Anomalies. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering* (CSMR '08). IEEE Computer Society, Washington, DC, USA, 279-283.

[58] Karin Erni and Claus Lewerentz. 1996. Applying design-metrics to object-oriented frameworks. In *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results* (METRICS '96). IEEE Computer Society, Washington, DC, USA, 64-.

[59] Karsai, Gabor and Narayanan, Anantha. 2006. On the correctness of model transformations in the development of embedded systems. In Kordon, F., Sokolsky, O., eds.: Monterey Workshop. Volume 4888 of LNCS., Springer.

[60] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems* (LCTES '99). ACM, New York, NY, USA, 1-9.

[61] Keith H. Bennett and Václav T. Rajlich. 2000. Software maintenance and evolution: a roadmap. In Proceedings of the Conference on The Future of Software Engineering (ICSE '00). ACM, New York, NY, USA, 73-87.

[62] Kiczales, Gregor; John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In proceedings of the European Conference on Object-Oriented Programming, vol.1241. pp. 220–242.

[63] Language for Model Transformations. 2006. Journal of Software and System Modeling, pp 261--288.

[64] Maher Lamari. 2007. Towards an automated test generation for the verification of model transformations. In Proceedings of the 2007 ACM symposium on Applied computing (SAC '07). ACM, New York, NY, USA, 998-1005.

[65] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. 2007. Towards Model Transformation Generation By-Example. In *Proceedings of the*

40th Annual Hawaii International Conference on System Sciences (HICSS '07). IEEE Computer Society, Washington, DC, USA, 285b-.

[66] Marcos Didonet Del Fabro and Patrick Valduriez. 2009. Towards the efficient development of model transformations using model weaving and matching transformations. Softw. Syst. Model. 8(3), 305–324.

[67] Marcos Didonet Del Fabro, Jean Bezivin, Frederic Jouault, Erwan Breton, and Guillaume Gueltas. 2005. AMW: A generic Model Weaver. In Int. Conf. on Software Engineering Research and Practice (SERP05).

[68] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In International conference on the unified modeling language UML03, San Francisco CA , ETATS-UNIS , vol. 2863, pp. 2-17.

[69] Mark Harman and John A. Clark. 2004. Metrics are fitness functions too. In IEEE METRICS. IEEE Computer Society, pp. 58–69.

[70] Mark Harman and Laurence Tratt. 2007. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (GECCO '07). ACM, New York, NY, USA, 1106-1113.

[71] Mark Harman. 2001. Search-based software engineering, Information & Software Technology, Vol. 43, No. 14, pp. 833-839.

[72] Mark Harman. 2007 The Current State and Future of Search Based Software Engineering, In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), 20-26 May, Minneapolis, USA.

[73] Mark O'Keeffe and Mel Ó Cinnéide. 2008. Search-based refactoring: an empirical study. Journal of Software Maintenance, vol. 20, no. 5, pp. 345–364.

[74] Marouane Kessentini, Arbi Bouchoucha, Houari Sahraoui and Mounir Boukadoum. 2010. Example-based Sequence Diagram to Colored Petri Nets Transformation Using Heurisitc Search. In Proc. of Modelling Foundations and Applications ECMFA 2010, Volume 6138, Springer, Pages 156-172, 2010.

[75] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2008. Model Transformation as an Optimization Problem. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems* (MoDELS '08),

Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus (Eds.). Springer-Verlag, Berlin, Heidelberg, 159-173.

[76] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2009. Transformation de modèle par l'exemple: approche par métaheuritique. Actes du 15e conférence francophone sur les Langages et Modèles Objets, mars 2009. éditions Cépadués.

[77] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2010. Testing Sequence Diagram to Colored Petri Nets Transformation: An Immune System Metaphor. In Proceedings of the 20th Annual International Conference on Computer Science and Software Engineering (CASCON2010).

[78] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2010. Example-based Model Transformation Testing, In Automated Software Engineering Journal (Accepted. To appear)

[79] Marouane Kessentini, Houari Sahraoui, and Omar Ben Omar. 2010. Search-based Model Transformation by Example. 2010. In SoSym Journal, Special Issue of Models08 (To appear).

[80] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer. 2011. Design Defects Detection Rules Generation : A Music Metaphor, 15th IEEE European Conference on Software Maintenance and Reengineering CSMR11.

[81] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer. 2011. Design Defects Detection by Example. 14th IEEE International Conference on Fundamental Approaches to Software Engineering FASE 2011.

[82] Marouane Kessentini, Stephane Vaucher, and Houari Sahraoui. 2010. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (ASE '10). ACM, New York, NY, USA, 113-122.

[83] Martin Fowler. 1999. Refactoring: Improving the Design of Existing Programs. Addison-Wesley Book

[84] Martin Fowler. 1999. Refactoring: Improving the Design of Existing Programs. Addison-Wesley Book

[85] Marwa Shousha, Lionel Briand, and Yvan Labiche. 2008. A UML/SPT Model Analysis Methodology for Concurrent Systems Based on Genetic Algorithms. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems* (MoDELS '08), Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus V\&\#246;lter (Eds.). Springer-Verlag, Berlin, Heidelberg, 475-489.

[86] Massimo Baleani and Alberto Ferrari and Leonardo Mangeruca and Alberto L. Sangiovanni-Vincentelli and Ulrich Freund and Erhard Schlenker and Hans-Jörg Wolff. 2005. Correct-by-construction transformations across design environments for model-based embedded software development. Design. Automation and Test in Europe Conference and Exhibition 2, pp 1040-1049.

[87] Muller Pierre-Alain, Franck Fleurey, and Jean-Marc Jézéquel. 2005. Weaving Executability into Object-Oriented Meta-Languages, in S. Kent L. Briand, ed., 'Proceedings of MODELS/UML'2005' , Springer, Montego Bay, Jamaica , pp. 264-278 .

[88] Naouel Moha, Yann-Gael Guéneuc, Laurence Duchien, and Anne-F. Le Meur.2010 DECOR: A method for the specification and detection of code and design smells," Transactions on Software Engineering Journal (TSE), 2009, 20-36.

[89] Norman E. Fenton and Shari Lawrence Pfleeger. 1998. Software Metrics: A Rigorous and Practical Approach (2nd ed.). PWS Pub. Co., Boston, MA, USA.

[90] Ó Cinnéide, Mel and Paddy Nixon. 2001. Automated software evolution towards design patterns. In Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01). ACM, New York, NY, USA, 162-165.

[91] OMG. MOF 2.0 Query/Views/Transformation RFP, 2002. OMG document ad/2002-04-10.

[92] OMG. MOF QVT Final Adopted Specification, 2005. OMG Adopted Specification ptc/05

[93] Óscar R. Ribeiro , João M. Fern. 2006. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets, 7th Workshop and Tutorial on Practical Use of

Coloured Petri Nets and the CPN Tools CPN 2006, Jensen K (ed.), Aarhus, Denmark, pp. 237-56.

[94] P Paolo Bottoni and Francesco Parisi-Presicce and Gabriele Taentzer. 2002. "Coordinated distributed diagram transformation for software evolution," Electronic Notes in Theoretical Computer Science, vol. 72, no. 4.

[95] Paul Baker, Mark Harman, Kathleen Steinhofel, and Alexandros Skaliotis. 2006. Search Based Approaches to Component Selection and Prioritization for the Next Release Problem. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (ICSM '06). IEEE Computer Society, Washington, DC, USA, 176-185.

[96] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger. 2009. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)", Springer, LNCS 5795 MoDELS 2009, pp 271-285.

[97] Philip Langer, Manuel Wimmer, and Gerti Kappel. 2010. Model-to-model transformations by demonstration. In *Proceedings of the Third international conference on Theory and practice of model transformations* (ICMT'10), Laurence Tratt and Martin Gogolla (Eds.). Springer-Verlag, Berlin, Heidelberg, 153-167.

[98] Prahladavaradan Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar. 2007. Testing Model-Processing Tools for Embedded Systems. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium* (RTAS '07). IEEE Computer Society, Washington, DC, USA, 203-214.

[99] Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws, in Proceedings of the International Conference on Software Maintenance, pp. 350–359.

[100] Raphaël Marvie.2004. A Transformation Composition Framework for Model Driven Engineering. Technical Report LIFL-2004-10, LIFL.

[101] Ravi Krishnamurthy, Stephen P. Morgan, and Mosh\&\#233; M. Zloof. 1983. Query-By-Example: Operations on Piecewise Continuous Data (Extended Abstract). In *Proceedings of the 9th International Conference on Very Large Data Bases* (VLDB '83), Mario Schkolnick and Costantino Thanos (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 305-308.

[102] Reiko Heckel. 1995. “Algebraic graph transformations with application conditions,” M.S. thesis, TU Berlin.

[103] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. 1987. Optimization by simulated annealing. In *Readings in computer vision: issues, problems, principles, and paradigms*, Martin A. Fischler and Oscar Firschein (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA 606-615.

[104] Salah Bouktif, Houari Sahraoui, and Giuliano Antoniol. 2006. Simulated annealing for improving software quality prediction. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation* (GECCO '06). ACM, New York, NY, USA, 1893-1900.

[105] Samar Abdi and Daniel Gajski. 2006. Verification of system level model transformations. Int. J. Parallel Program. 34, 1, 29-59.

[106] Samir Khuller and Balaji Raghavachari. 1996. Graph and network algorithms. *ACM Comput. Surv.* 28, 1 (March 1996), 43-45.

[107] Sanjay Rawat and Ashutosh Saxena. 2009. Danger theory based SYN flood attack detection in autonomic network. In *Proceedings of the 2nd international conference on Security of information and networks* (SIN '09). ACM, New York, NY, USA, 213-218.

[108] Sendall, S., Kozaczynski, W. 2003. Model transformation – The heart and soul of model-driven software development. IEEE Software, Special Issue on Model Driven Software Development, vol. 20, no. 5, pp. 42-45.

[109] Stephan Lechner and Michael Schrefl. 2003. Defining web schema transformers by example. In Proceedings of DEXA'03 conference. Springer, pp 46-56.

[110] Stephanie Forrest, Alan S. Perelson, Lawrence Allen, and Rajesh Cherukuri. 1994. Self-Nonself Discrimination in a Computer. In *Proceedings of the 1994 IEEE*

Symposium on Security and Privacy (SP '94). IEEE Computer Society, Washington, DC, USA, 202-.

[111] Thierry Millan, Laurent Sabatier, Thanh-Thanh Le Thi, Pierre Bazex, and Christian Percebois. 2009. An OCL extension for checking and transforming UML models. In Proceedings of the 8th WSEAS International Conference on Software engineering, parallel and distributed systems (SEPADS'09), World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 144-149.

[112] Thierry Millan, Laurent Sabatier, Thanh-Thanh Le Thi, Pierre Bazex, and Christian Percebois. 2009. An OCL extension for checking and transforming UML models. In *Proceedings of the 8th WSEAS International Conference on Software engineering, parallel and distributed systems* (SEPADS'09), World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 144-149.

[113] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. Electron. Notes Theor. Comput. Sci. 152 (March 2006), 125-142.

[114] Tom Mens and Tom Tourwe. 2004. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.* 30, 2, 126-139.

[115] Tom Mens, Serge Demeyer, and Dirk Janssens. 2002. Formalising Behaviour Preserving Program Transformations. In *Proceedings of the First International Conference on Graph Transformation* (ICGT '02), Andrea Corradini, Hartmut Ehrig, Hans-Jerg Kreowski, and Grzegorz Rozenberg (Eds.). Springer-Verlag, London, UK, 286-301.

[116] Van Eetvelde, Niels and Janssens, Dirk. 2003. A hierarchical program representation for refactoring. In Proc. of UniGra'03 Workshop.

[117] Varro Daniel. 2004. Automated Formal Verification of Visual Modeling Languages by Model Checking. Journal of Software and Systems Modeling, col. 3(2), pp. 85-113.

[118] Varró, Dániel; Varró, Gergely & Pataricza, András. 2002. Designing the automatic transformation of visual languages. Science of Computer Programming, 44(2):205–227.

[119] William F. Opdyke. 1992. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign.

[120] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. 1998. Antipatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc., New York, NY, USA.

[121] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. 1998. Antipatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc., New York, NY, USA.

[122] Xactium. Xmf-mosaic. http://xactium.com.

[123] Xavier Dolques, Marianne Huchard, Clémentine Nebut, and Philippe Reitz. 2010. Learning transformation rules from transformation examples : An approach based on relational concept analysis. Proceedings of the IEEE EDOC 2010 workshops and short papers.

[124] Yu Sun, Jules White, and Jeff Gray. 2009. Model Transformation by Demonstration. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems* (MODELS '09). Springer-Verlag, Berlin, Heidelberg, 712-726.

[125] Yuehua Lin , Jing Zhang , Jeff Gray. 2005. A Testing Framework for Model Transformations. In Proceedings of Model-Driven Software Development - Research and Practice in Software Engineering.. Springer, pp 219--236.