Handling High-Level Model Changes Using Search Based Software Engineering

by

Usman Mansoor

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information Systems Engineering)
in the University of Michigan Dearborn
2017

DEDICATION

I dedicate this thesis to my beautiful wife, Kholla. None of this would have been possible without her by my side.

I also dedicate this to my parents, Naeema Mansoor and Mansoor Ahmed Tariq for their constant encouragement and support. Their love and sacrifices have made me the man I am today.

And to all my immediate and extended family, the completion of this PhD is owed greatly to your continued prayers. Thank you.

ACKNOWLEDGEMENTS

Completing this PhD would not have been possible without the guidance of Professor Kessentini. He has been an extraordinary advisor and it was because of his assistance, time, and energy that I have been able to complete this thesis.

I would also like to thank all the committee members, Professor William Grosky, Professor Ghassan Kridli, Professor Bruce Maxim, and Professor Qiang Zhu, for their additional guidance and mentorship throughout the different stages of completing this degree.

I want to especially thank Emily Wang and Don Barbarci for always being there for us international students. They have not only provided support throughout any obstacle, but have created a family of friends that has been invaluable during times of homesickness.

And finally, I would like to thank all the SBSE lab members, particularly Mohamed Wiem Mkaouer, Josselin Dea, and Dilan Sahin. As well as all my other friends who have helped me along the way.

PREFACE

The research that led to this thesis was performed at Search-Based Software Engineering Laboratory at the Department of Computer and Information Science, University of Michigan-Dearborn, with Prof. Marouane Kessentini as the main advisor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Model-Driven Engineering (MDE) considers models as first-class artifacts during the software lifecycle. The number of available tools, techniques, and approaches for MDE is increasing as its use gains traction in driving quality, and controlling cost in evolution of large software systems. Software models, defined as code abstractions, are iteratively refined, restructured, and evolved. This is due to many reasons such as fixing defects in design, reflecting changes in requirements, and modifying a design to enhance existing features.

In this work, we focus on four main problems related to the evolution of software models: 1) the detection of applied model changes, 2) merging parallel evolved models, 3) detection of design defects in merged model, and 4) the recommendation of new changes to fix defects in software models.

Regarding the first contribution, a-posteriori multi-objective change detection approach has been proposed for evolved models. The changes are expressed in terms of atomic and composite refactoring operations. The majority of existing approaches detects atomic changes but do not adequately address composite changes which mask atomic operations in intermediate models.

For the second contribution, several approaches exist to construct a merged model by incorporating all non-conflicting operations of evolved models. Conflicts arise when the application of one operation disables the applicability of another one. The essence of the problem is to identify and prioritize conflicting operations based on importance and context – a gap in existing approaches. This work proposes a multi-objective formulation of model merging that aims to maximize the number of successfully applied merged operations.

For the third and fourth contributions, the majority of existing works focuses on refactoring at source code level, and does not exploit the benefits of software design optimization at model level. However, refactoring at model level is inherently more challenging due to difficulty in assessing the potential impact on structural and behavioral features of the software system. This requires analysis of class and activity diagrams to appraise the overall system quality, feasibility, and inter-diagram consistency. This work focuses on designing, implementing, and evaluating a multi-objective refactoring framework for detection and fixing of design defects in software models.

KEYWORDS

Search-based Software Engineering, Model Merging, Model-Driven Engineering, Multi-Objective Optimisation, Refactoring, Defect Detection.

Chapter 1: Introduction

1.1 Model Driven Engineering

The majority of industrial software companies are nowadays dealing with projects involving high number of requirements, hard deadlines, and high expectations in terms of efficiency and quality of the resulting software [1]. Thus, Model-Driven Engineering (MDE) [2] is applied increasingly to cope with the complexity of software systems by raising the level of abstraction. To address the size of software systems, teams of developers have to cooperate and work in parallel on software models. Consequently, techniques to support building models collaboratively are a necessity.

Model-Driven Engineering (MDE) [3] considers models as first-class artifacts during the software lifecycle. The number of available tools, techniques, and approaches for MDE is increasing and more attention is paid to the evolution aspects in MDE [3] along with the growing importance of modeling in software development. In fact, software models, defined as code abstractions, are iteratively refined, restructured, and evolved due to many reasons such as correcting errors in design, reflecting changes in requirements, and modifying a design to enhance existing features. Thus, effective techniques to evolve models as well as to understand their evolution are a must.

Models are used to generate primary reusable software artifacts. This helps in generating generic code for the platform along with schematic code based on application based model transformations. Therefore, MDE, if properly deployed and maintained in a software development process, has the potential to increase productivity.

In this work, we use Unified Modeling Language (UML) to define the studied software models. UML is an industry standard for conceptualizing, defining, detailing, and documenting software systems. We use both the Class Diagrams and Activity Diagrams in analyzing the studied models to not only incorporate class and object information that make the system, but also system states and conditions that the system assumes as it executes over time. This allows for specifying both static and dynamic constraints as well as implemented workflow of the system.

1.2 Detection of Changes at Model Level

The transition from code to models induces the need for adequate techniques to cope with the continuous and concurrent evolution of models. One of the most important tasks in this realm is the detection and analysis of operations that have been applied between two versions of a model. In general, we may distinguish between two categories of operations. The first category concerns *atomic operations*, such as additions, deletions, updates, and moves. The second category comprises *composite operations* [4] consisting of a set of cohesive atomic operations, which are applied within one transaction to achieve one common goal. Such operations usually have well-defined pre and post conditions that specify their applicability and their intended effect. The most prominent class of composite operations concerns refactorings introduced by Opdyke [5]. However, composite operations are not limited to refactorings; they may be used to implement any kind of in-place model transformation for supporting a specific purpose, such as model completion, refinement, and evolution. As reported in [6], the availability of the information on applied composite operations is a crucial prerequisite for automating several model management tasks. These tasks include model versioning and merging, the comprehension of a model's evolution, and the co-evolution of models, such as the migration of models to new meta-model versions and the synchronization of models and multiple views on them.

1.3 Merging Models

When models are changed in parallel, they have to be merged eventually to obtain a consolidated model. Therefore, several approaches have been proposed for detecting the operations that have been applied in parallel by developers. Once the applied operations are available, conflict detection algorithms are used to identify pairs of operations that interfere with each other [7]. In this regard, a conflict denotes a pair of operations, where one operation masks the effect of the other (i.e., they do not commute) or one operation disables the applicability of the other. An example for the former is a pair of parallel operations that update the same feature in the model with different values. The latter case is at hand if one operation's preconditions are not valid anymore after applying the operations of the other developer. Such a scenario frequently occurs if composite operations (a sequence of cohesive atomic changes), e.g., model refactorings [5] [8], are applied because they may have potentially complex preconditions that may easily be invalidated by parallel operations.

In general, two kinds of merge approaches can be distinguished [9]. First, *state-based* merge approaches aim at merging two model versions by combining their model elements into one merged model. Second, *operation-based* merge approaches in contrast do not reason about the models' states, but consider recorded operation histories and apply the combination of the parallel histories to the common initial version to compute the merged version.

For both approaches, the notion of conflict is essential, because when having two parallel evolutions of one model, not all operations may be combined to compute one unique merged model. Basically, we can distinguish between two kinds of conflicts. First, two operations are conflicting if one operation masks the effect of the other operation in the merged version: e.g., for update/update conflicts, the latter update in the operation sequence applied on the model is effective, while the former update is lost. Thus, such conflicting operations are not confluent, i.e., different operation sequences result in different models. Second, a conflict also occurs if one operation disables the applicability of the other. Every operation has specific preconditions, e.g., an update of an element can only be performed when the element still exists; otherwise a delete/update conflict is raised.

Empirical studies [10] showed that users prefer to work with a tentative merged model acting as a basis for reasoning about possible conflict resolutions, instead of working with the list of operations in terms of choosing to reject one or the other conflicting operation for creating a merged model. A few approaches respect this preference and produce a merged model by applying all non-conflicting operations; conflicting operations are omitted. However, especially in case of a large number of conflicts, many operations are not merged with this strategy, leading to a tentative merged model that does not reflect the maximal combined effect of the parallel operations. Furthermore, the majority of existing works [11] [12] [6] [13] [14] [15] [10] treat the applied refactoring operations to be merged with equal importance. However, in a real world scenario these operations have different importance scores that can depend on the type of the refactoring and the context of the refactoring application. Thus, in existing work the developer cannot integrate her/his priority preferences concerning the importance of some refactorings that should be included in the merging process.

1.4 Detection and Correction of Design Defects

Refactoring is a widely used technique to improve the overall quality of systems by altering design structure while preserving the overall functionalities and behavior [5].

A refactoring operation is a transformation aimed at mitigating a code smell or anti pattern without altering the behavior of the software. Therefore improving the structure of code does not only require identification of the afflicting smell but also carefully proposing the refactoring transformations which fix the smell without the expense of new smell or even worse a software bug. Usually a code smell is identified by using a set of detection rules.

These detection rules tend to look for certain anomalies and characteristics in the code e.g. number of attributes in a class, number of methods etc to locate code smells in the source code. Manual formulation of detection rules is a no easy task. It requires careful inspection and analysis to formulate a set of detection rules which detect the code smells while minimizing the false positives.

A variety of refactoring work has been proposed in the literature [5] [16] and the majority of them focus only on the source code level. Despite its importance, model refactoring is still in its infancy [17] [18] [19]. In fact, model refactoring is more difficult and challenging than code refactoring for several reasons. First, the evaluation of the impact of refactorings in the model level is difficult. At the source code level, traditional code quality metrics are used to evaluate the quality of a system after applying a sequence of refactorings. However, applying refactoring on a specific model such as class diagrams have an impact on related other diagrams such as activity diagrams, sequence diagrams, etc. Sometimes, an improvement of class diagram quality metrics may decrease the quality of an activity diagram. Thus, it is important to evaluate the impact of suggested refactorings not only on one diagram but also other related diagrams to estimate the overall quality. Second, some refactorings suggested at the model level cannot be applied to the source code level. For example, a move method between two classes can be applicable at the class diagram level but cannot be applied in the source code one. Such situations can be detected using an activity diagram that can evaluate the feasibility of some refactorings. Third, it is difficult to check if a refactoring applied to a class diagram preserves the behavior or not without the use of some related behavioral diagrams such as an activity diagram.

1.5 Contributions

The main focus of this work is to address the challenges in software development when evolving software models. A model can evolve into multiple parallel models. Therefore, detecting the list of changes in an evolved model is essential to keep software development consistent. Parallel evolved models are always merged together to consolidate requirements and determine conflicts. Merged models need to ensure the quality specification and should not have design defects. The four components in model driven software development are shown in Figures 1-5. The contribution of this work is divided into four sections, and each section specifically focuses on each of these stages and uses search based optimization to address challenges in change detection, merge models, defect detection, and defect correction in software models. In each of the following sections we present the block diagram for each stage.

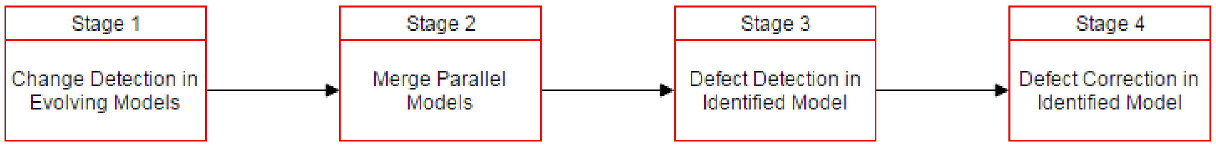

Stage 1
Change Detection in Evolving Models
Stage 2
Merge Parallel Models
Stage 3
Defect Detection in Identified Model
Stage 4
Defect Correction in Identified Model

Figure 1: The four main componenents of Model Driven Software Development : (1) Change Detection (2) Model Merging (3) Defect Detection, and (4) Defect Correction

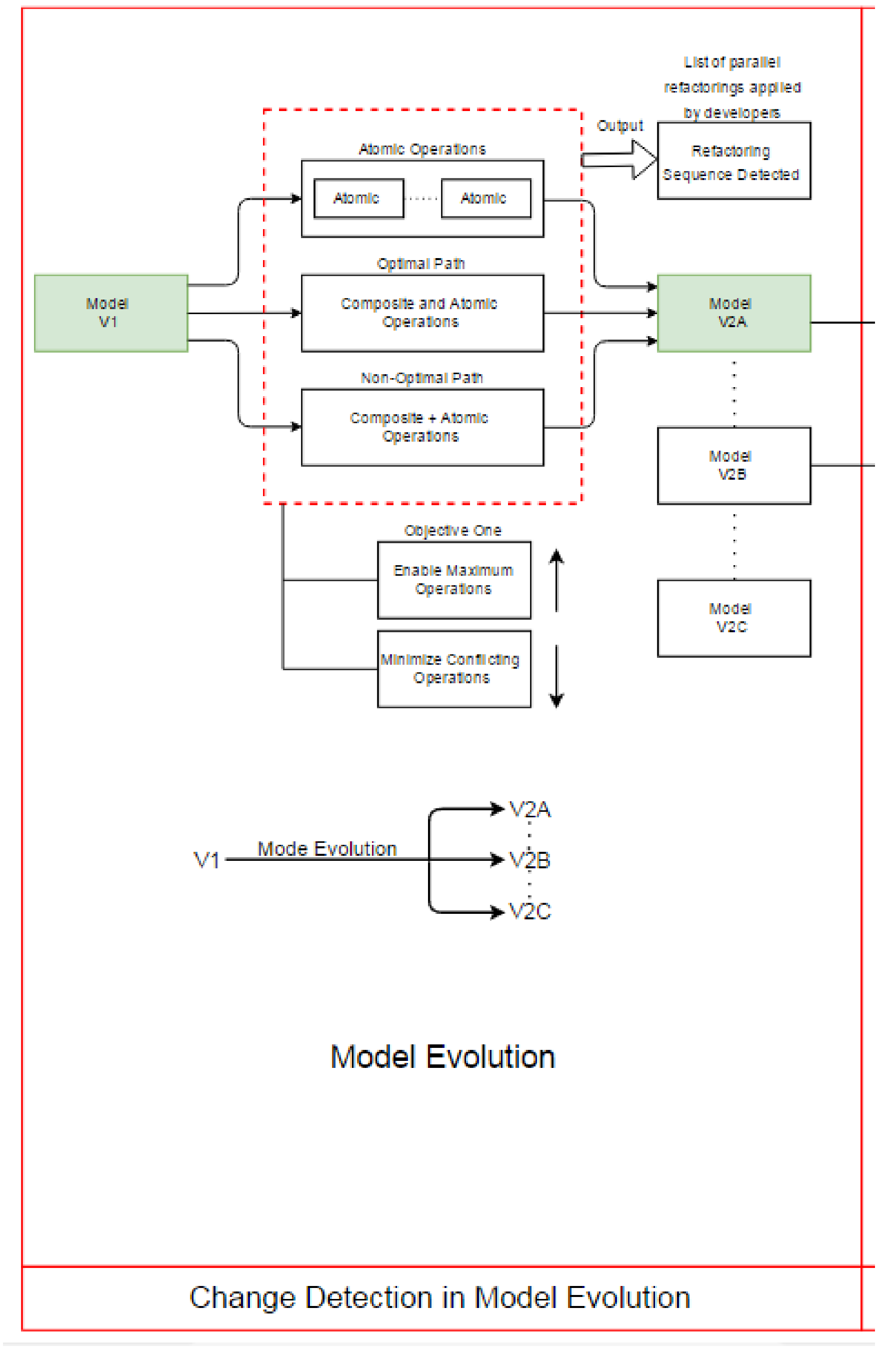

List of parallel refactorings applied by developers
Output
Refactoring Sequence Detected
Atomic Operations
Atomic
Atomic
Optimal Path
Composite and Atomic Operations
Non-Optimal Path
Composite + Atomic Operations
Model V1
Model V2A
Model V2B
Model V2C
Objective One
Enable Maximum Operations
Minimize Conflicting Operations
V1
Mode Evolution
V2A
V2B
V2C
Model Evolution
Change Detection in Model Evolution

Figure 2 : Detailed Model for Change Detection in Evolving Models. Model V1 is initial Model and Model V2A, V2B ..V2C are the parallel evolved models. The rectangular block represents the application of search based detection scheme to detect the sequence of changes between initial and evolved models.

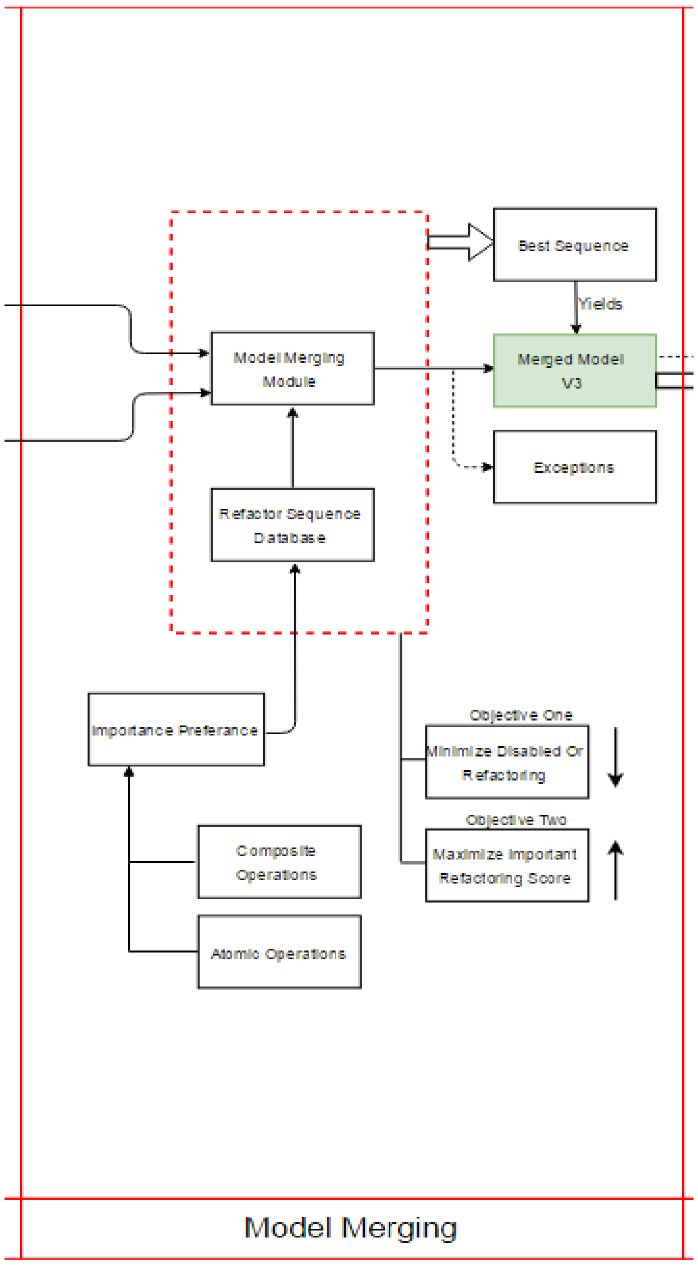


Figure 3 : Model Merging using search based merging techniques. Model V3 is the merged Model which uses Model V2A as initial model. The rectangular block represents the application of search based merge scheme merge two models

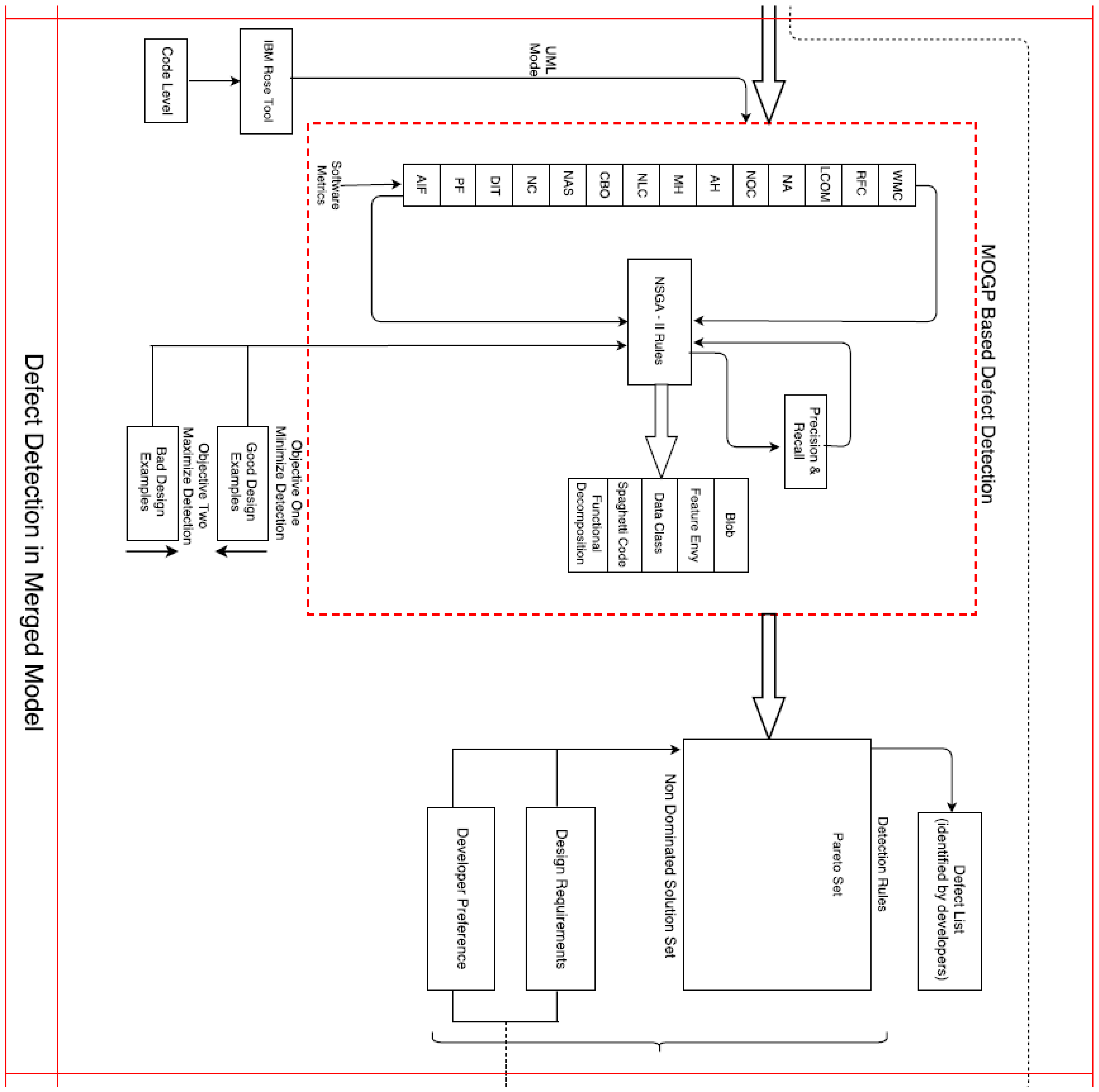


Figure 4: Defect Detection in Merged Model. The rectangular block represents the application of search based defect detection technique which uses set of software metrics to detect design defects.

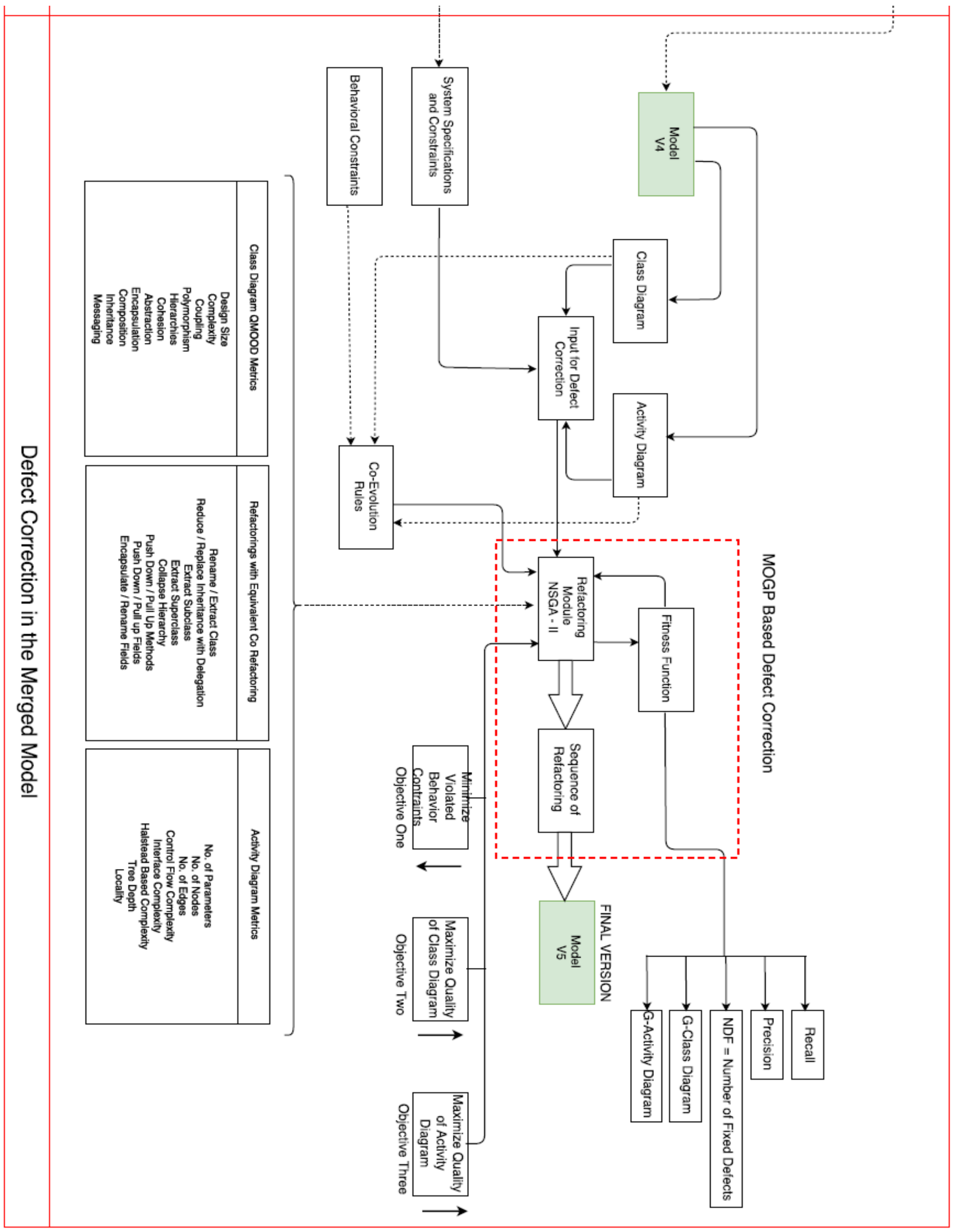


Figure 5: Defect Correction in the identified Model. The dotted rectangular block represents the application of search based defect correction technique which uses a set constraints as input and finds a final model which meets the quality specifications.

1.5.1 Contribution 1: Detection of Composite Changes at Model Level

This work addresses the problem of the detection of composite changes at Model Level as a multi objective optimization problem. The approach takes as input an initial and a revised model, and as controlling parameters, an exhaustive list of refactoring operations. Refer to Figure 2 for detailed block level overview. The approach generates a set of refactoring sequences that represents the evolution from the initial model to the revised model. The process of detecting model changes can be viewed as the mechanism that finds the best way to combine refactoring operations from the exhaustive list of possible refactorings, in such a way to *(i)* maximize the similarity between the revised model and the resulting model when applying the detected refactorings on the initial model and *(ii)* minimize the number of refactorings. In other words, this process aims at finding the best tradeoff between these two conflicting criteria.

1.5.2 Contribution 2: Model Merging

This work attempts to treat the problem of model merging as a combinatorial optimization problem. The goal of our approach is to construct a tentative merged model that minimizes the number of disabled operations and maximizes the number of important enabled ones. Therefore, we use a multi-objective optimization algorithm to compute an optimal sequence of merging operations in terms of finding trade-offs between minimizing the number of operations that are disabled by preceding operations and maximizing the number of important operations that are enabled. Figure 3 shows a detailed block level overview. The process of generating a solution can be viewed as the mechanism that finds the best order among all possible operation sequences.

1.5.3 Contribution 3: Design Defect Detection

The defect detection problem involves searching for the metric combinations among the set of candidates, which constitutes a huge search space. The solution of defect detection problem is a set of rules (metric combination with their thresholds values). Figure 4 shows a detailed block level overview. We address the problem of defect detection as an optimization problem which involves tuning of thresholds in detection rules to maximize the capture of design defects or anti patterns in the analyzed model.

1.5.4 Contribution 4: Design Defect Correction

After having identified the design defects in the merged model the next logical step would be to suggest possible refactoring solutions to minimize the defects in the model. The proposed approach aims to find the best sequence of refactorings that provides a good trade-off between maximizing the quality of class diagrams and activity diagrams while preserving the behavioral constraints defined in activity diagrams.

Figure 5 shows an overview of the proposed refactoring solution. Therefore, we use a multi-objective optimization algorithm to compute an optimal sequence of refactorings which tend to not only improve the structural integrity of the design but also preserves the system behavior.

Chapter 2: Related Work

2.1 Detecting Changes at the Model Level

In general, existing approaches propose to detect differences between model versions using pre and post conditions specified for each refactoring. In this regard, we distinguish the existing work in two categories: the first category [20] [21] [22] [23] produces only atomic differences and the second category [24] [25] [26] is able to produce composed differences such as detecting refactorings. Our work can be classified in the second category. In this case, the specified conditions are related to the possible changes that can be detected by comparing between source and revised models. However, some problems limit their effectiveness. First, it is not possible to find the applied refactorings, if they have been performed in an overlapping sequence, because their pre and post conditions might not be valid due to preceding or succeeding refactorings when only considering the initial and the final versions of a model in the absence of each intermediate version after every single refactoring. Second, the list of possible changes and their combination may be very large for some modeling languages, e.g., consider UML where most refactorings of object-oriented programming may be applied on class diagrams [27]. Thus, it is a fastidious task to specify detection rules for each possible refactoring combination, which is currently required by most of the existing approaches. Third, the evolution of a model from an initial version to a revised version can be described using different refactoring sequences having the same result. In fact, some complex refactorings are equivalent to a specific composition of basic refactorings. For instance, an Extract Class refactoring is a combination of a set of atomic operations such as Create Class, Move Field, Move Method, etc. Thus, some criteria have to be used to choose the best solution from equivalent ones. One of the important criteria, in addition to correctness of detected changes, is the number of refactorings used to describe the model evolution. In general, minimizing the number of refactorings is equivalent to maximizing the number of large composite refactorings used to describe the evolution.

Similar to change tracking for programming environments, the usage of change tracking in model environments has been proposed in [6] coming with the same advantages and disadvantages as discussed before for program code. The state-based comparison of models attracted much attention in the last years that led to a huge list of different model comparison approaches which has been surveyed in [28].

Most approaches for model comparison apply a two step-process: they first identify the correspondences between model elements of two different model versions, and second, based on these correspondences, the differences between the two model versions are derived. In the context of software evolution, difference calculation has been investigated intensively as witnessed by a number of approaches ranging from simple text comparisons based on the XMI serializations of models to dedicated model differencing techniques. Table 1 depicts a summary of the approaches discussed in the following and illustrates a comparison of them with our proposed approach.

A specialized differencing method has been introduced to compare UML models by Xing and Stroulia [29]. Another UML specific differencing approach is proposed by Nejati et al. [4] which is specifically tailored for matching UML state machines. SiDiff [30] started also as a comparison tool for UML models but has been further generalized to detect changes for arbitrary models by having an internal generic graph representation to which rarity languages can be mapped. Alanen and Porres [31] discussed how to detect differences between MOF-based models. Rivera and Vallecillo [32] propose to compare model based on identifier as well as structural similarities by using rules defined in Maude. Thus, before the models are compared, they are translated to corresponding Maude representations on which the comparison is actually performed. DSMDiff [33] is another protagonist for computing differences between models, irrespectively of their metamodel. Finally, EMF Compare [34] is an Eclipse plug-in for comparing EMF-based models providing out-of-the-box matching and differencing support as well as several extension points to develop tailored matching and differencing support. All the approaches mentioned in this paragraph derive only atomic changes, namely additions, deletions, updates, and some of them, moves.

A few approaches use an additional third step in the model comparison process to combine atomic changes to composite changes by using dedicated aggregation rules. In particular, the output of the second phase of model comparison is used as input for the third phase, i.e., the earlier detected atomic changes are rewritten to composite ones by applying the aggregation rules.

The starting point is the approach by Xing and Stroulia [27] for detecting refactorings in evolving software models which is integrated in UMLDiff. For detecting refactorings, change pattern queries have to be developed for each refactoring. These change pattern queries are executed on a difference model obtained by a state-based model comparison to report refactoring applications. The approach is tailored to UML models and the detection of hidden changes is not mentioned.

In Langer et al. [35], an approach based on graph transformation rules to detect refactorings in software models is presented. In contrast to [27], no additional rules have to be developed for detecting changes. On the contrary, the rules for executing the operations are reused for detecting applications. Using an iterative and incremental operation detection approach, overlapping operation sequences are supported as long as the post conditions of preceding operations that enable other succeeding operations are fulfilled in the revised model. However, operation sequences, in which the post conditions are not fulfilled in the revised model, are not supported.

Kehrer et al. [26] follows a similar path by proposing to derive dedicated detection rules from graph transformation rules that represent composite operations. The derived detection rules are then matched on difference models containing the atomic operations that have been applied between two versions of a model. The approach by Vermolen et al. [24] copes with the detection of evolution steps between different versions of a metamodel to allow for a higher automation in model migration, i.e., to adapt the existing models to the new metamodel version. They use a difference model comprising atomic changes as input and calculate composite changes. The approach is tailored to the core of object-oriented metamodeling languages, but follows a similar methodology as UMLDiff. However, the approach requires developing an additional detection rule for every possible change combination which represents a practical challenge if a larger set of refactoring operations is used.

In the area of business process modeling, there is the work of Küster et al. [25] for calculating hierarchical change logs for business process models including compound changes in the absence of recorded change logs. The authors apply the concept of Single-Entry-Single-Exit fragments to calculate the hierarchical change logs after computing the correspondences between two process models. Thereby, several atomic changes are hidden behind one compound change representing an introduction or deletion of a model fragment. However, changes that cross-cut the containment hierarchy are not considered.

In summary, the detection of composite operations and hidden operations on models is currently only discussed by Vermolen et al. [36] and Langer et al. [37]. Nevertheless, when following [36] for detecting hidden refactorings, additional rules have to be developed by hand for each possible combination which makes the elaboration of the complete search space practically impossible due to the huge size of possible refactoring combinations and even more challenging due to eventually overlapping atomic changes. In [37], no additional rules are needed, but some refactorings are not detectable when the operation's post conditions are not valid in the final model version.

Table 1: Comparison of Selected Comparison Approaches for Programs, Models, and Ontologies

	Language-specific	Composite changes Detectable	Composite changes Additional Rules	Hidden Changes Detectable	Hidden Changes Additional Rules
REF-FINDER	yes	yes	yes	no	n.a.
Dig et al.	yes	yes	yes	no	n.a.
Weissgerber et al.	yes	yes	yes	no	n.a.
Demeyer et al.	yes	yes*	yes	yes*	no
EMFCompare	no	no	n. a.	n.a.	n.a.
Alanen & Porres	no	no	n.a.	n.a.	n.a.
DSMDiff	no	no	n.a.	n.a.	n.a.
SiDiff	no	no	n.a.	n.a.	n.a.
Barret et al.	no	no	n.a.	n.a.	n.a.
Riviera & Vallecillo	no	no	n.a.	n.a.	n.a.
SiLift	no	yes	no	no	n.a.
ETL	no	yes	yes	no	n.a.
UMLDiff	yes	yes	yes	no	n.a.
Küster et al.	yes	yes	yes	no	n.a.
Vermolen et al.	yes	yes	yes	yes°	yes
Langer et al.	no	yes	no	yes°	no
Hartung & Rahm	yes	yes	yes	no	n.a.
GenDiff	**no**	**yes**	**no**	**yes**	**no**

Legend:
* an estimation possible refactorings is reported
° hidden operations are only partially supported

2.2 Model Merging Approaches

Research on versioning systems has a long tradition in software engineering dating back to the early 1970s [38]. Conradi & Westfechtel [39] proposed so-called version models to characterize and document the diversity of existing versioning approaches.

With respect to the scope of work in this thesis, namely to integrate two parallel operation histories into one operation sequence that maximizes the number of successfully applied important operations, related work dates back to the early 1990s. Before that time, merging has been mostly achieved based on the states of the artifacts under version control [40]. The origin work on operation-based merging has been published by Lippe [15]. He pointed out several advantages of operation-based merging over purely state-based merging and contributed the important notion of frontier set [15]. The frontier set is constructed by building a two-dimensional space, where one axis enumerates the changes by one developer and the other axis the changes done in parallel by another developer, and subsequently collecting the points in this space which do not exactly correspond to one possible merge solution. It is an indicator of extent of conflicts that exist when trying to combine the given changes. Thus, the frontier set, including the so-called frontier points, is an indicator how far one can merge two sequences of operations. One goal is to shift the frontier points as far away from the original model version as possible to maximize the applicability of the performed operations. One way to shift the frontier points is to reorder the atomic operations, i.e., to apply all non-conflicting atomic operations before the conflicting ones. What we have contributed with our search-based approach is a mechanism to reduce the critical points in the merge process where the users have to be involved, even when composite operations, such as refactorings, have been applied.

Operation-based merging has been heavily applied in asynchronous collaborative graphical editing. Edwards [14] has defined several strategies for combining two operation sequences into one sequence. The strategies range from fully automatic merging by computing each possible sequence of non-conflicting operations to interactive merging allowing the user to decide how each operation of a conflicting operation pair should be incorporated in the finally merged model. Ignat & Norrie [9] have compared an operation-based approach and a state-based approach for merging operation logs of collaborative graphical editors. They distinguish "real" conflicts from resolvable conflicts. The latter may be resolved by finding an appropriate order to incorporate the operations to the final merged model. For finding the appropriate order, priority lists for operation types have to be defined. In our approach, we also use priorities, i.e., the importance score, for operations, but only for "real conflicts" and not for resolvable conflicts as proposed by Ignat & Norrie. The latter are resolved by searching for the sequences that allow applying the operations of resolvable conflicts.

Munson & Dewan present a flexible framework for merging arbitrary objects, which may be configured in terms of merge policies [41]. Merge policies may be tailored by users to their specific needs and include rules for conflict detection and rules for automatic conflict resolution. Actions for automatic conflict resolution are defined in merge matrices and incorporate the kinds of atomic operations made to the object and the users who performed those operations. Thus, it may be configured, e.g., that operations of specific users always dominate operations of others, or that updates outpace deletions.

With the advent of MDE, the research topic collaborative modeling is gaining momentum. Several state-based approaches for model versioning have been proposed (see [40] for an overview), as well as a few operation-based approaches. Koegel et al. [6] record operations in modelling editors and provide conflict detection for two sequences of recorded operations. They also support composite operations, but only consider how these operations are built up from atomic operations while explicit preconditions are disregarded. If they detect that a composite operation is in conflict with an atomic operation, they let the user decide which one to take. Similarly, Barret et al. [42] discuss pushing the frontier points as far as possible by incorporating all non-conflicting operations to produce a merged model and then let again the user decide which operation of a conflict pair to prioritize. Other operation-based approaches for models have been presented in [43] [44] [41], but no dedicated reordering strategies have been discussed. In [45], the authors present a fixed merge policy with a dedicated focus on raising the number of enabled parallel operations applied on ordered features by interleaving insert operations of both developers and relaxing the constraint that the inserted elements have to be exactly at the given index as long as the relative order of them is still given. In [46], the authors mention that finding an appropriate sequence for unifying the operations of two parallel operation sets may be considered as an optimization problem, but they based their approach on manual conflict resolution during the merge process.

Cicchetti et al. [47] propose an adaptable merging algorithm by defining conflict patterns describing specific difference patterns which are supplemented with a reconciliation strategy. Such strategies state how the conflicts should be automatically resolved by specifying either which side should be preferred in the merge process or by introducing a transformation to resolve the conflict. While policy-based approaches require user intervention in certain conflict cases where no policy is at hand, [10] present a formal merge approach based on graph transformation theory, yielding a merged model by construction and deferring the resolution of conflicts.

In summary, maximal arbitrary application orders of the operations is not considered by existing operation-based merge approaches. State-of-the-art approaches mostly reside on a two-phase process: first, they apply the non-conflicting operations and then let the user select the operation to be prioritized out of two conflicting operations. In contrast, our approach explores arbitrary sequences and the result is the most applicable sequence of operations found by NSGA-II. Thus, we are able to minimize the critical and labor-intensive tasks involving user interaction in the merge process going beyond existing state-of-the-art approaches.

2.3 Detection of Design Defects

Software metrics can be used to capture the structural and semantic attributes of the software, and can be a reliable indicator of the quality of design. These quality indicators can then be used to quantitatively estimate and reflect the design signatures of a software architecture in terms of many metrics including coupling, cohesion, cyclic complexity etc. The code smell detection process usually involves finding the fragments of code which violate these software metrics. Chidamber and Kemerer [48] has performed one of the benchmarking studies in classifying these software metrics for object oriented architectures. We selected and used in our experiments the following quality metrics, namely Weighted Methods per Class (WMC), Response For a Class (RFC), Lack of Cohesion Of Methods (LCOM), Cyclomatic Complexity (CC), Number of Attributes (NA), Attribute Hiding factor (AH), Method Hiding factor (MH), Number of Lines of Code (NLC), Coupling Between Object classes (CBO), Number of Classes (NC), Depth of Inheritance Tree (DIT), Polymorphism Factor (PF), Attribute Inheritance Factor (AIF) and Number Of Children (NOC). Table 2 lists the relevant structural metrics along with their desirable value ranges.

Table 2: Structural Metrics, their relevance and desirable value ranges.

Weighted Method Class	It measures the effort needed to maintain a class by counting the number of methods in class along with their normalized complexity [49].	Desirable value should be low
Response for a class	It is the count of number of methods and constructors that can be invoked in a class [50].	Desirable value should be low. Usually a value less than 50 is recommended.
Lack of Cohesion Method	It estimates the cohesiveness of methods present in a class. Classes with disjoint methods with non intersecting functionalities are prone to bugs and higher effort during maintenance. Non cohesive classes are good candidates for sub division into more cohesive smaller classes [51].	Lower values are more desirable since it indicates absence of non-cohesive characteristics among methods in a class.
Cyclomatic Complexity	It estimates the program complexity by measuring the number of linearly independent paths. A high cyclomatic complexity indicates that that program structure is too complex resulting in increased risk in maintenance [52].	Low values are desired.
Number of Attributes	Number of attributes in a class counts the number of fields in a class. It helps in estimating the risk potential in future extensions and maintainability. A class with too many attributes is a high risk class and is a possible candidate ExtractClass [50].	Low values are desired. Values higher than 10 may pose potential risks.
Attribute Hiding Factor	It is a measure of attribute hiding. Ideally all attributes should be hidden and be visible only	It is measured in ration and can have maximum value of 1 for

	to corresponding class methods. This not only improves encapsulation but also implementation refinement in the design [53].	best case scenarios. Values close to zero indicates complete absence of encapsulation in the design.
Method Hiding Factor	It is a measure of functionality in class methods. It indicates a tradeoff between functionality and abstractness. A good design incorporates as much functionality in its class methods while not compromising on abstractness [53].	The value is a tradeoff. A high value indicates high functionality. However, high functionality might be achieved at the expense of abstractness in the design.
Number of Lines of Code	It is the count of number of physical lines in the code. In our experiment we include every line in the source code including comments but excluding blanks.	It can be used as a metric to measure the size of program, class or a method. It is used in conjunction with other metrics.
Coupling between Objects	It is a count of number of classes a considered class is coupled. A high coupling indicates that methods in one class are excessively used in the other classes. This indicates lack of modular design and potentially high maintenance costs [48].	The desired value for this metric is low. Inter object class coupling should be kept to minimum to allow for less complex, modular and encapsulated design.
Number of Classes	It is the number of classes in the source code. It is to be used in conjunction with other supporting metrics such as response of a class, weighted method etc.	Though no desired value can be ascribed to this metric, the objective is to have number of classes in the design such that complexity and cohesion is reduced with minimal tradeoff on coupling.
Depth of Inheritance Tree	It quantitatively measures the depth of a class in class hierarchy. Since a change in an ancestor class might trigger a change in inherited class, a class lying deep in inheritance tree usually requires higher maintenance cost [48].	A bigger inheritance tree usually results in more complexity. DIT should be small with recommended value range of less than 5.
Polymorphism Factor	It is a measure of use of parametrized classes in the system. This allows for flexible and refined design.	It is measured in ratio with 1 indicating maximum usage of parametrization.
Attribute Inheritance Factor	A class which inherits a lot of attributes from its ancestor results in high attribute inheritance [54]. It indicates the level of reuse in the system.	Higher value indicates reuse of ancestral attributes while low value indicates redefinition. The value is upper bounded to 50%.

Number of Children	It is the count of number of directly subordinate classes in class hierarchy tree.	Greater number of children might indicate improper abstraction, deeper inheritance as well as more complex design.

The manual definition of rules to identify is difficult and can be time-consuming. One of the main issues is related to the definition of thresholds when dealing with quantitative information. For example, the 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. Thus, the manual definition of detection rules sometimes requires a high calibration effort. Furthermore, the manual selection of the best combination of metrics that formalize some symptoms of code-smells is challenging. In fact, multiple metrics can detect same symptoms while tracking their corresponding code smell trigger signature. For example, a very deep inheritance tree might also be accompanied by unsettling values in number of children, coupling between objects, cohesion etc. Thus different possible metric combinations need to be explored to fully exploit the visibility of smell in multiple metrics. In addition, the translation of code-smell definitions into metrics is not straightforward. Some definitions of code-smells are confusing and it is difficult to determine which metrics to use to identify such design problems.

2.4 Correction of Design Defects

With respect to the contribution of this work, we organize related approaches using three categories of related work: *(i)* refactoring approaches working solely on the model level, *(ii)* refactoring working on model and code level that may be also considered as a kind of multi-view refactoring, and *(iii)* widely related approaches working solely on the code level.

2.4.1 Model Level Refactoring

Two surveys concerning model refactorings are available [55] [56] that discuss different research trends and classifications for model refactoring. One of the first investigations in this area was done by Sunyè et al. [57] who define a set of UML refactorings on the conceptual level by expressing pre- and post-conditions in OCL. Boger et al. [58] present a refactoring browser for UML supporting the automatic execution of pre-defined UML refactorings. While these two approaches focus on pre-defined refactorings, approaches by Porres [59], Zhang et al. [60], and Kolovos et al. [61] discuss the introduction of user-defined refactorings by using dedicated textual languages for their implementation. A similar idea is followed in [62] [63]where graph transformations are used to describe refactorings and graph transformation theory is applied for analyzing model refactorings. Pattern-based refactoring for UML models with model transformations is presented in [64].

The mentioned approaches cover mostly single-view refactorings and focus on the implementation of semi-automatically executable refactorings. Only some approaches for tackling consistency between different views in the context of refactorings have been presented. For instance, [65] [66] proposed to refactor UML class diagrams, also adapting attached OCL constraints. Another approach that considers the effect on refactorings of UML class diagrams on operations implemented in OCL with respect to behavioral equivalence is presented in [67]. A constraint-based refactoring approach for UML is presented in [68] which considers well-formed rules and translates refactorings to CSP to eventually compute the additional changes required for a semantic-preserving model refactoring.

Reuse of model refactorings for different languages is discussed in [69] by specifying generic role-based refactorings that can be bound to specific languages. Another approach aiming for generic model refactorings is present in [70] by using a combination of aspect weaving and model typing. Refactorings are developed on a generic metamodel and may be reused for specific metamodels which fulfill the model typing relationship to the generic metamodel.

In [71] tool support for defining model metrics, smells, and refactorings is presented. In particular, language specific and project specific metrics, smells, and refactorings for the design-level may be defined based on graph transformations. A refactoring approach considering performance optimization of models, i.e., runtime-level, is presented in [19]. In this context, refactorings are used to eliminate anti-patterns that may have a negative impact on performance aspects.

Related to multi view refactoring is the field of multi-view consistency [72]. We have early works on multi-view consistency [61] [17] using a generic representation of modifications and relying on users to write code to handle each type of modification in each type of view. This idea influenced later efforts on model synchronization frameworks in general [73] [74] and in particular bi-directional model transformations [75] [64]. Other approaches use so-called correspondence rules for synchronizing models in the contexts of RM-ODP and model-driven web engineering [76] [77] [70]. All these approaches have in common that they consider only atomic changes when reconciling models and not refactorings. In [70], coupled transformations to refactor different views were presented by automatically executing the coupled transformations when initial transformations are executed.

In related work [78] it has been proposed to use Interactive Genetic Algorithm (IGA) for model refactoring allowing the modelers to provide feedback during refactoring focusing on single-view improvements.

To abridge, all these mentioned model refactoring approaches are mostly considering a single view during refactoring. If multiple views are considered by approaches from multi-view synchronization, the only quality aspect that is taken care of is having consistency between the different viewpoints.

2.4.2 Multi View Refactoring

The synchronization of models and code is a considerably challenging issue when it comes to refactoring. In [79], distributed graph transformations are used to specify coupled refactorings on UML models and Java code. Van Gorp et al. [74] have presented an extension of the UML metamodel which allows expressing the pre and post conditions for refactorings as well as for representing method implementations in UML class diagrams based on the UML action semantics – a predecessor of fUML. Furthermore, they use OCL to detect code smells on the model level and propose to refactor designs independent of the underlying programming language on the model level by applying the following transformation chain: reverse engineering, model refactoring, and forward engineering. The approach for constraint-based model refactoring [68] discussed before, has been also extended for dealing model/code co-refactoring by so-called bridge constraints that capture the correspondences between model elements and the code elements [80].

To sum up, there are some approaches that consider refactorings on both model and code level, but we are not aware of any approach considering the models aligned with code as a multi-objective optimization problem.

Chapter 3: Search Based Problem Solving

3.1 Search Based Optimization Algorithm – NSGA-II

3.1.1 NSGA-II Definitions

Before describing the adaptation of NSGA-II to solve the multi objective optimization problems, some background definition related to multi-objective optimization is described based on [81]:

Definition 1 (MOP). A multi-objective optimization problem (MOP) consists in minimizing or maximizing an objective function vector under some constraints. The general form of a MOP is as follows:

$$\begin{cases} Min\ f(x) = [f_1(x), f_2(x), ..., f_M(x)]^T \\ g_j(x) \geq 0 & j = 1,...,P; \\ h_k(x) = 0 & k = 1,...,Q; \\ x_i^L \leq x_i \leq x_i^U & i = 1,...,n. \end{cases}$$

where M is the number of objective functions, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond to the lower and upper bounds of the variable x_i. A solution x_i satisfying the (P+Q) constraints to be feasible and the set of all feasible solutions defines the feasible search space denoted by Ω. In this formulation, we consider a minimization MOP since maximization can be easily turned to minimization based on the duality principle by multiplying each objective function by -1. The resolution of a MOP consists in approximating the whole Pareto front.

Definition 2 (Pareto optimality). A solution $x^* \in \Omega$ is Pareto optimal if $\forall\, x \in \Omega$ and $I = \{1,...,M\}$ either $\forall\ m \in I$ we have $f_m(x) = f_m(x^*)$ or there is at least one $m \in I$ such that

$f_m(x) > f_m(x^*)$.The definition of Pareto optimality states that x^* is Pareto optimal if no feasible vector x exists which would improve some objective without causing a simultaneous worsening in at least another one. Other important definitions associated with Pareto optimality are essentially the following:

Definition 3 (Pareto dominance). A solution $u = (u_1,u_2,..., u_n)$ is said to dominate another solution $v= (v_1,v_2,...,v_n)$ (denoted by $f(u) \preceq f(v)$) if and only if $f(u)$ is partially less than $f(v)$. In other words, $\forall\, m \in \{1,...,M\}$ we have $f_m(u) \le f_m(v)$ and $\exists\, m \in \{1,...,M\}$ where $f_m(u) < f_m(v)$.

Definition 4 (Pareto optimal set). For a MOP $f(x)$, the Pareto optimal set is $P^* = \{x \in \Omega \mid \neg\exists x' \in \Omega, f(x') \preceq f(x)\}$.

Definition 5 (Pareto optimal front). For a given MOP $f(x)$ and its Pareto optimal set P^*, the Pareto front is $PF^* = \{f(x), x \in P^*\}$.

3.1.2 Overview of NSGA-II

NSGA-II [82] is one of the most widely used multi-objective evolutionary algorithms (EAs) in tackling real world problems including software engineering ones [83] [84] to find trade-offs between different objectives simultaneously. It begins by generating an offspring population from a parent one by means of variation operators (crossover and mutation) such that both populations have the same size. After that, it ranks the merged population (parents and children) into several non-dominated layers, called fronts, as depicted in Figure 6. Non-dominated solutions are assigned a rank of 1 and constitute the first layer. Non-dominated solutions according to the population truncated of the layer 1 are assigned rank of 2 and constitute the layer 2. This process is continued until the ranking of all parent and children individuals is complete. After that, each solution is assigned a diversity score, called crowding distance, front wise. This distance corresponds to the half of the perimeter of the cuboid having the two closest neighboring solutions to the considered individual as vertices. It is important to note that extreme solutions are assigned an infinite crowding score since they are of great importance for diversity. The fitness in NSGA-II is not a scalar value. In fact, it is a couple (rank, crowding distance). Solutions having better ranks are emphasized. Among solutions having the same rank (belonging to the same layer), solutions having larger crowding distances are emphasized since they are less crowded than the others. Once all individuals of the merged population are assigned a rank and a diversity score, we perform the environmental selection to form the parent population for the next generation. Indeed, solutions belonging to the best layers are selected. Figure 6 illustrates this process where the last selected layer is the 4th one. Usually, the cardinality of the last layer (layer 4 in Figure 6) is greater than the number of available slot in the parent population of the next generation. As denoted by Figure 6, solutions of the 4th layer are selected based on their crowding distance values. In this way, most crowded solutions are the least likely to be selected; thereby emphasizing population diversification. To sum up, the Pareto ranking encourages convergence and the crowding factor procedure emphasizes diversity, therefore NSGA-II is an elitist multi-objective EA which is today widely used to address real-world problems including software engineering ones.

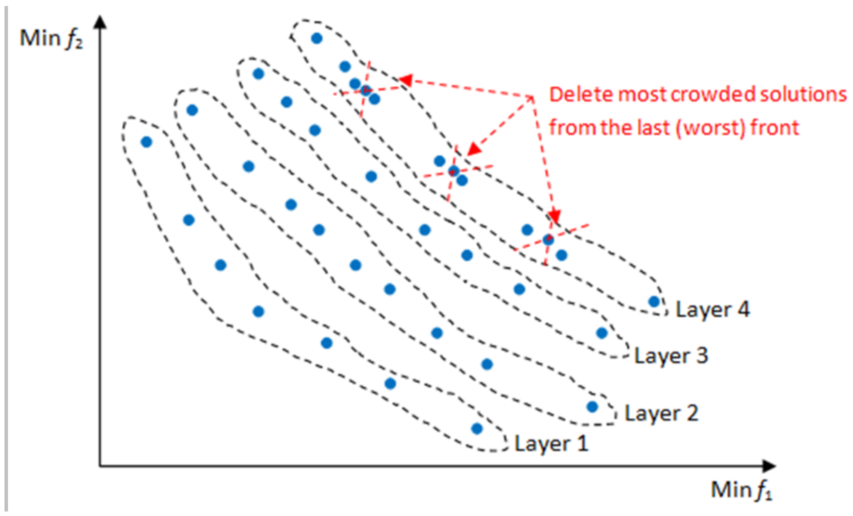


Figure 6 : NSGA-II replacement scheme.

3.2 Quality Metrics for Search Based Optimization

When comparing two mono-objective algorithms, it is usual to compare their best solutions found far during the optimization process. However, this is not applicable when comparing two multi-objective evolutionary algorithms since each of them gives as output a set of non-dominated (Pareto equivalent) solutions. For this reason, we use the three following performance indicators [85] when comparing NSGA-II with any other multi objective optimization scheme:

Hypervolume **(*IHV*)** corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. A Pareto front or a set is a set up non dominated solutions among which every solution meets the selection criteria. It is difficult to improve the quality of a pareto solution without adversely affecting the quality of other solutions in the front. Larger values for this metric mean better performance. The most interesting features of this indicator are its Pareto dominance compliance and its ability to capture both convergence and diversity. The reference point used in this study corresponds to the nadir point [86] of the Reference Front

(*RF*), where the Reference Front is the set of all non-dominated solutions found so far by all algorithms under comparison as shown in Figure 7.

Inverse Generational Distance **(*IGD*)** is a convergence measure that corresponds to the average Euclidean distance between the Pareto front Approximation (*PA)* provided by the algorithm and the reference front *RF*. The distance between *PA* and *RF* in an *M*-objective space is calculated as the average *M*-dimensional Euclidean distance between each solution in *PA* and its nearest neighbour in *RF*. Lower values for this indicator mean better performance (convergence).

Contribution **(*IC*)** corresponds to the ratio of the number of non-dominated solutions the algorithm provides to the cardinality of *RF*. Larger values for this metric mean better performance.

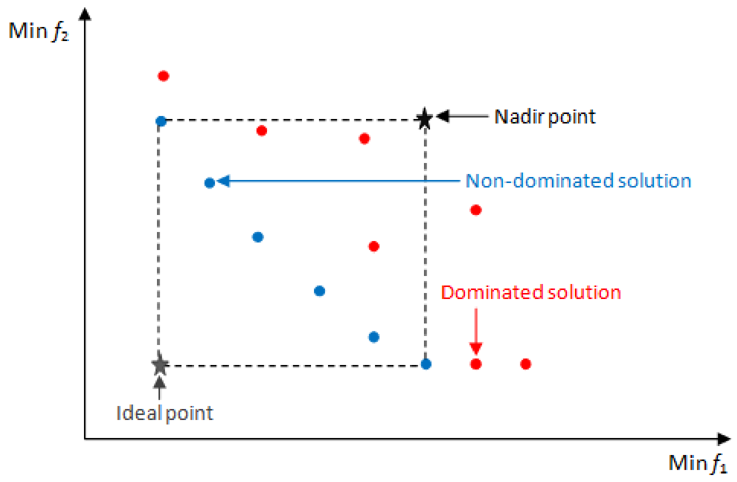


Figure 7 : A Demonstration of Nadir Point in a set of non dominated solutions in a Pareto Front.

Throughout this work we use these three metrics to compare performance of different MOGP schemes and evaluate the quality of the selected solution sets.

Chapter 4: Changes Detection at Model Level

4.1 Introduction

Models evolve during software development process. It can be incremental evolution due to design improvement, requirement changes, implementation bottlenecks and work around, feedback from end users, budget dependencies, or software milestones. Furthermore, parallel models can evolve from main trunk due multiple product lines, new products, software module reuse etc. By enabling optimum model reuse among different product lines, code reuse is increased, and therefore cost of developing, testing, delivering and maintaining software decreases. However, model reuse is not a simple problem. The fundamental knot in enabling model reuse among different product lines is to detect changes between parallel models. If these changes can be broken down into atomic or composite blocks, it will allow the software development to estimate the cost of changes, develop glide path to reach new model from existing model, and implement and deploy the changes. This strategy of change detection allows existing models to be reused heavily, therefore reducing the cost of developing and maintaining customized software specific to needs to multiple product lines.

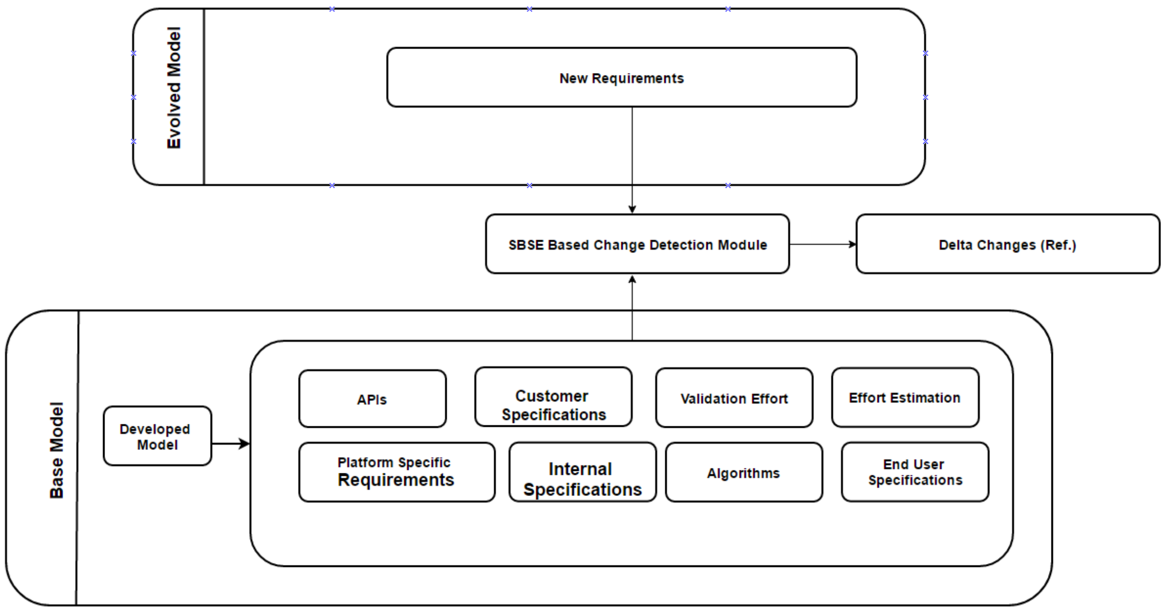


Figure 8 : A Base Model with set of defined APIs and Specifications is being compared using SBSE techniques with an evolved Model.

4.1.1 Model Comparison : A Motivating Example

To illustrate the issue of overlapping sequences of refactorings, we present in this section a motivating example for illustrating atomic operations and discuss the challenges of finding an optimal sequence of composite operations that best describes the evolution history of a model.

Consider the evolution scenario depicted in Figure 10. In this scenario, we have two model versions, called *V1* and *V2*, in the repository. Between these two versions, two refactorings have been applied. First, the user applied the refactoring *Inline Class*. This refactoring moves all features of the class to be inlined into another class that originally references the inlined class. Finally, the inlined class can be deleted without losing any information. The precondition of this refactoring is that the inlined class is referenced through a single-valued reference by the class that eventually contains all features. Applying this refactoring to our example, the attribute version is moved from the class Description to Service and the class Description, as well as the reference description in class Service, is finally deleted.

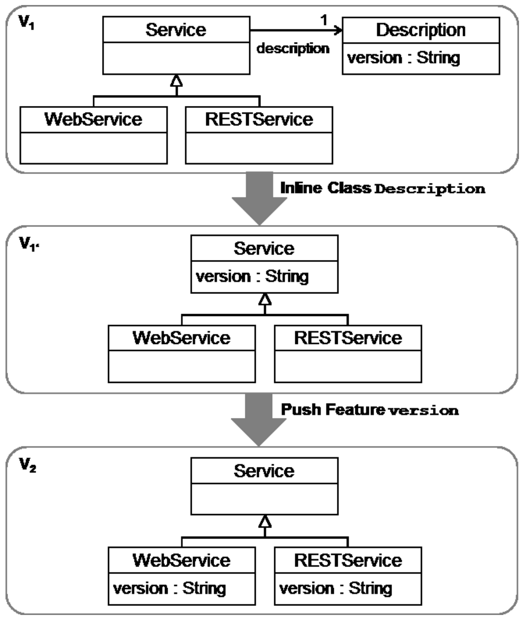
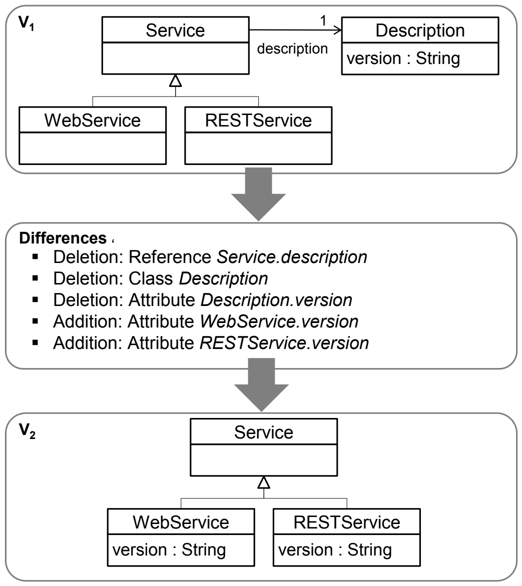


Figure 9: Example of atomic refactoring between evolving models

When applying a model comparison algorithm to the model versions V_1 and V_2, we detect that the class Description has been deleted and the attribute version has been added to WebService and RESTService. However, this situation does not match with any refactoring specification, because neither the postconditions of Inline Class nor the preconditions of Push Feature are fulfilled when only considering the available model versions V_1 and V_2. Thus, no refactoring can be detected, as the second refactoring hides the operation constituting the first refactoring and the intermediate state V_1' is not available in the repository.

Although several remarkable techniques have been proposed for deriving the applied operations a posteriori from two versions of a model. However, most of the existing model comparison tools are capable of detecting only atomic operations and, thus, do not provide the crucial information for enabling several model management tasks that also require the explicit knowledge on applied composite operations.

An example for atomic changes and how they are represented is shown in Figure 10. In this Figure, two versions of a simple UML class diagram representing different kinds of services are shown in the upper and lower half of the Figure, whereas in the middle the atomic operations between the two versions are represented. Although, for this simple example the differences are not too complex to understand, larger examples that involve a higher number of operations are challenging to be understood. Furthermore, for reasoning on the evolution of models, a set of atomic operations solely fails to reveal explicitly the reasons and intentions behind the actual evolution. Therefore, an explicit set of composite operations would be necessary.



Differences
- Deletion: Reference *Service.description*
- Deletion: Class *Description*
- Deletion: Attribute *Description.version*
- Addition: Attribute *WebService.version*
- Addition: Attribute *RESTService.version*

Figure 10: Change list in terms of atomic refactoring operations between V_1 and V_2 models. The lack of presence of composite operations makes it difficult to infer the underlying reasons behind the choice of atomic refactoring operations.

However, only a few approaches have been proposed that also address the detection of composite operations. These approaches search for patterns of atomic operations among all atomic operations obtained from model comparison tools. If a change pattern of a composite operation is found, they evaluate the pre and post conditions of the respective operation before an occurrence of the operation is reported. A detailed case study of such an approach [35] revealed that successful detection of such operations is only achieved as long as there are no overlapping sequences of composite operations. However, in several scenarios the subsequent operations mask preceding operations (e.g., they delete an updated element so that the update is not visible anymore) or render the postconditions of the preceding operations invalid. Also it is not uncommon that multiple composite parallel or overlapping operations might have been applied. As a result, current approaches do not have a high detection rate of composite operations especially when overlapping composite sequences are present. Studies, however, have shown that as much as 20% of model transformation consists of overlapping composite sequences [35]. This highligts the importance of detection mechanism which is resilient to presence of composite operations in all its different personifications including overlapping sequences.

4.1.2 Quality of Detected Operations Sequence

A naive approach to this problem would be to relax the pre and post conditions of refactorings to a certain degree during the detection process. However, this would lead to several wrong refactoring indications (false-positives). Moreover, the pre and post conditions are very specific to certain refactorings and it seems to be impossible to decide on a general basis which conditions can be relaxed and which conditions should hold at any time to accomplish a trustworthy refactoring detection.

A better approach is to search for a possible sequence of operations that lead from the initial model to the given final model, whereas the preconditions of every operation in this sequence are fulfilled when applied at the respective position in the sequence. However, as applying one operation in this sequence may affect the applicability of subsequent operations, finding a valid operation sequence is a combinatorial optimization problem within a huge search space. But not all valid solutions are good solutions in terms of understandability and minimality. Consider, for instance, the alternative operation sequence (Figure 11) which constitutes a valid solution for the same initial version V_1 and revised version V_2 too. In this sequence, we first add the class Description as a superclass of Service and apply the composite operation Imitate Super Class to Service. This refactoring replaces a superclass by a subclass, whereas all its features are maintained by the respective subclass. Thus, in our example, the target of the reference description is changed from Description to Service and the attribute version of Description is moved to Service. Finally, the class Description is deleted as it is now *imitated* entirely by the class Service. Next, we delete the feature description and finally apply the Push Feature refactoring to obtain the same version V_2 as from the original sequence in Figure 10.

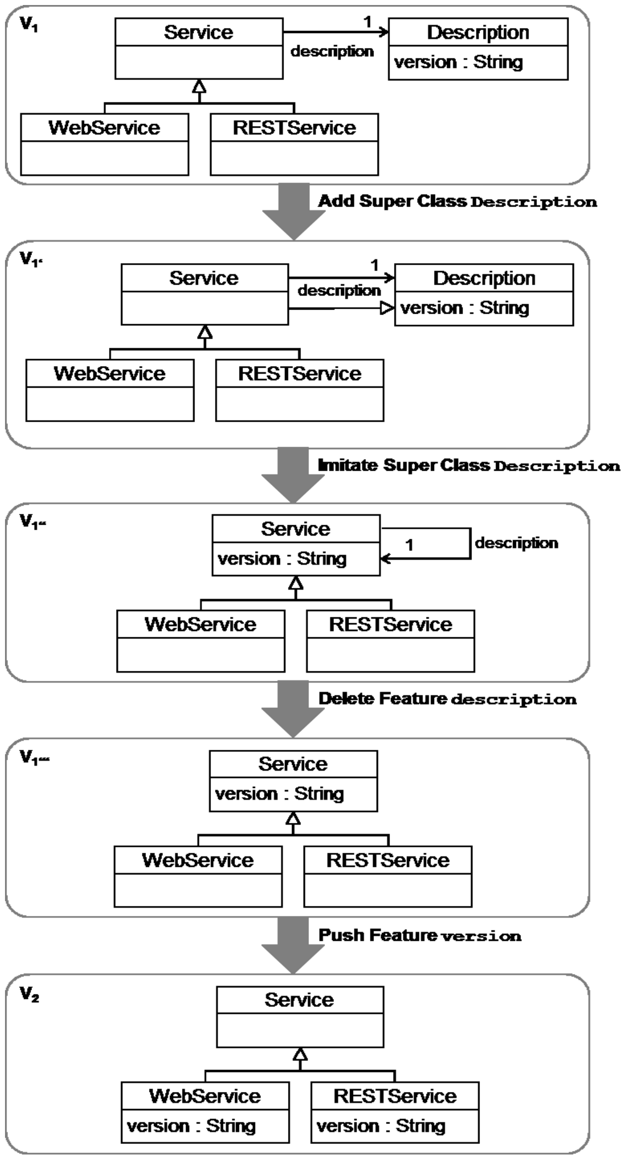


Figure 11: Alternate Operations Sequences

Although this operation sequence is valid in terms of the preconditions of all applied operations and does not contain unnecessary operations (e.g., an addition of an element that is removed again later), it still does not reflect the evolution optimally, because it is not as concise as the original sequence and it still contains an atomic operation that has been originally part of a refactoring (i.e., Delete Feature, which deletes the reference description). Note that even more alternative sequences of operations that also constitute valid transitions from V_1 to V_2 are possible. However, all of these alternative operation sequences contain additional atomic and composite operations instead of showing the most direct transition from V_1 to V_2. In addition, it is easier for developers to understand changes described in terms of a short sequence of composite operations. Thus, these sequences distort the actual evolution, as well as the original intentions of the users. This makes it harder to understand the model evolution and potentially hampers the automation of certain model management tasks that depend on these operation sequences.

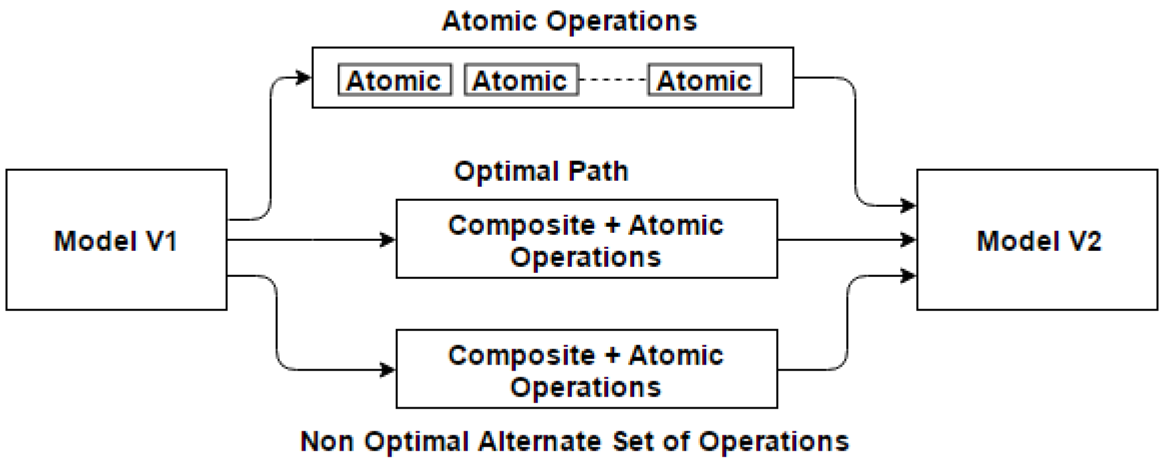

Atomic Operations
Atomic
Atomic
Atomic
Optimal Path
Model V1
Composite + Atomic Operations
Model V2
Composite + Atomic Operations
Non Optimal Alternate Set of Operations

Figure 12 : Possible set of operations (both atomic and composite) which can generate a target model from an initial one.

4.2 Approach Overview

4.2.1 Detection Scheme

The general structure of our approach is introduced in Figure 13. The approach takes as input an initial and revised (evolved) model, and as controlling parameters, an exhaustive list of refactoring operations. The approach generates a set of refactoring applications that represents the evolution from the initial model to the revised model. The process of detecting model changes can be viewed as the mechanism that finds the best way to combine refactoring operations of the exhaustive list of possible refactorings, in such a way to *(i)* maximize the similarity between the revised model and the resulting model when applying the detected refactorings on the initial model and *(ii)* minimize the number of refactorings. In other words, this process aims at finding the best tradeoff between these two conflicting criteria. In fact, maximizing the correctness of the detected refactorings corresponds to maximizing the similarity between the initial model after applying refactorings and the revised one (expected or evolved model). Minimizing the number of refactoring used to describe the changes corresponds to maximizing the use of a few larger (composite) refactorings instead of using a larger amount of smaller (atomic) refactorings for describing the evolution.

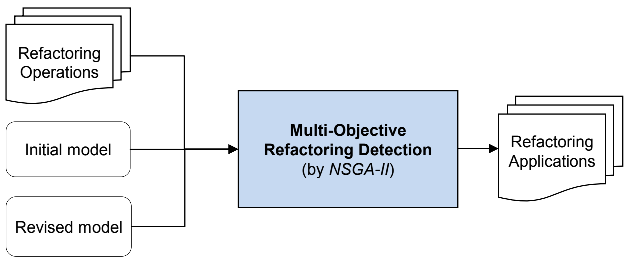


Figure 13 : Input and output of the multi-objective model change detection approach

Due to the large number of possible refactoring solutions and the two conflicting objectives, we consider the detection of refactoring between different model versions as a multi-objective optimization problem instead of a single-objective one [87]. The algorithm explores a huge search space. In fact, the search space is determined not only by the number of possible refactoring combinations, but also by the order in which they are applied. Formally, if m is the number of available refactoring operations, then the number of possible refactoring solutions is given by all the permutations of all the possible subsets and is at least equal to: $(m!)^m$. The shear vastness of search space is shown in Figure 14. It is evident that with just number of three available rfactoring operations the search space can be as large as 300 possible solutions. This number can be bigger, since the same refactoring operations can be applied several times in different model fragments. To explore this huge search space, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best trade-off between our two conflicting objectives [82].

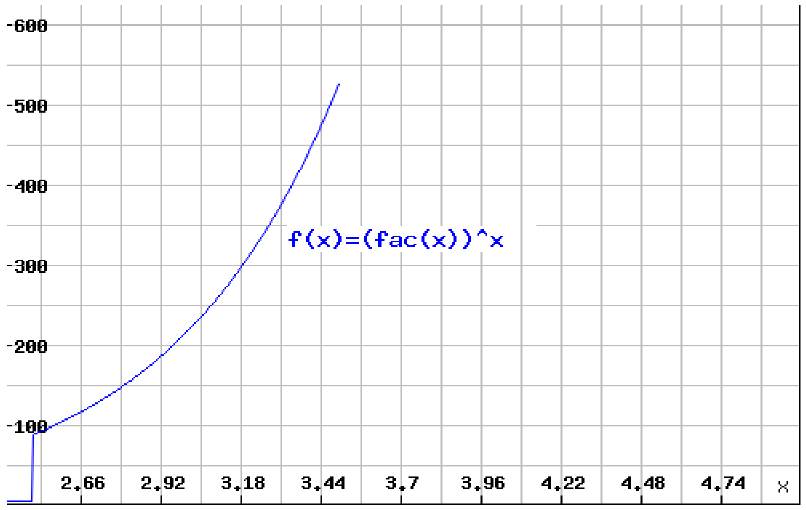


Figure 14 : Size of search space. Graph Plotted for $(m!)^m$.

4.2.2 Solution Coding

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, detecting high-level model changes. We discuss an example where the i^{th} individual (solution) represents a combination of refactoring operations to apply. The order of applying refactorings corresponds to their position in the table (referred to as dimension number in the following). In addition, the execution of the refactorings respects pre and post conditions to avoid conflicts and semantic inconsistencies. Furthermore, it has to be noted that the same type of refactoring operation could be applied several times in the same solution(but to different model fragments).

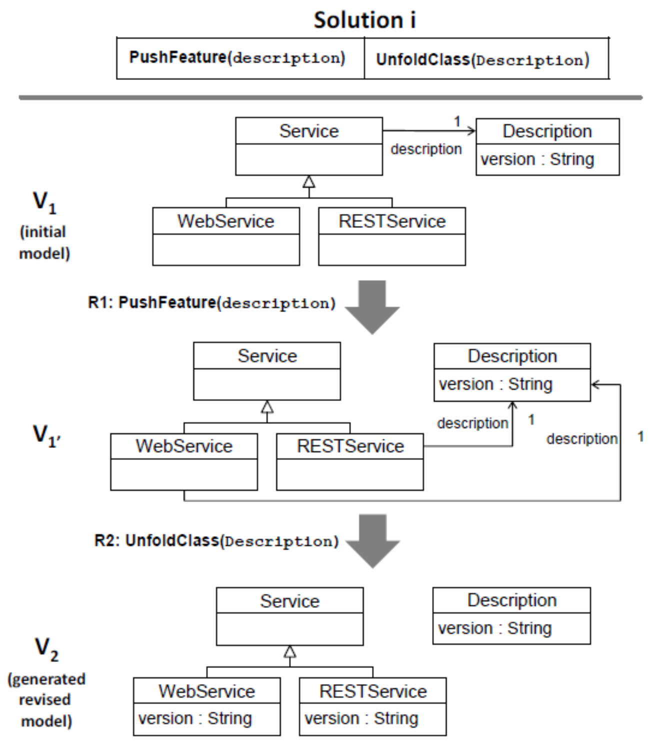


Figure 15 :Solution Coding Example

For instance, the solution represented in Figure 15 is composed by two dimensions corresponding to two refactoring operations to apply in some model fragments. The two refactorings composing the generated solution are *Push* Feature that is applied to the reference *description* and *Unfold Class* that is applied to the class *Description*. After applying the proposed solution we obtain a new model that will be compared with the expected revised model using a fitness function.

When generating a solution, we selected from the exhaustive list of refactorings proposed by Fowler in [88] which can be applied in the model level for UML class diagrams. But it has to be noted that the approach is designed to be generic, thus also other refactorings for various modeling languages may be supported. We considered the following list of refactorings in our experiments:

- **Replace Inheritance with Delegation** Replaces a direct inheritance relationship with a delegation relationship.
- **Replace Delegation with Inheritance** Replaces a delegation relationship with a direct inheritance relationship.
- **Rename Class** Changes the name of a class to a new name, and updates its references.
- **Extract Hierarchy** Adds a new subclass to a non-leaf class C in an inheritance hierarchy.
- **Extract Subclass** Adds a new subclass to class C and moves the relevant features to it.
- **Extract Superclass** Adds a new super class to class C and moves the relevant features to it.
- **Collapse Hierarchy** Removes a non-leaf class from an inheritance hierarchy.
- **Inline Class** Moves all features of a class into another class and deletes it.
- **Extract Class** Creates a new class and moves the relevant features from the old class into the new one.
- **Push Down Method** Moves a method from a class to those subclasses that require it.
- **Pull Up Method** Moves a method from some class(es) to the immediate superclass.
- **Rename Method** Changes the name of a method to a new one, and updates its references.
- **Move Method** Creates a new method with a similar body in the class it uses most. Either turns the old method into a simple delegation, or removes it.
- **Push Down Field** Moves a field from a class to those subclasses that require it.

- **Pull Up Field** Moves a field from some class(es) to the immediate superclass.
- **Move Field** Moves a field from a class to another one which uses the field most.
- **Rename Field** Changes the name of a field to a new name, and updates its references.
- **Encapsulate Field** Creates getter and setter methods for the field and uses only those to access the field.

4.2.3 Solution Search

For NSGA based search algorithm to initialize, the first step is to select an initial set of population. This population set will evolve in successive generations based on a set of well defined criteria to iterate towards the optimum solution in search space.

4.2.3.1 Initial Population Set

To generate an initial population, we start by defining the maximum vector length including the number of refactorings. The vector length is proportional with the number of refactorings to use for detecting model changes. Sometimes, a high vector length does not mean that the results are more precise, but that only a few refactorings are sufficient to detect changes. These parameters can be specified either by the user or chosen randomly. Thus, the individuals have different vector length (structure). Then, for each individual we randomly assign one refactoring, with its parameters, to each dimension.

4.2.3.2 Selection Criteria

Since any refactoring combination is possible, we do need to define some conditions to verify when generating an individual. For example, we cannot remove a method from a class that was already moved to another class, thus the pre-conditions and post conditions of the operations have to be fulfilled before their execution.

To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS) [89], in which the probability of selection of an individual is directly proportional to its relative fitness in the population. SUS is a random selection algorithm which gives higher probability to be selected to the fittest solutions while still giving a chance to every solution. For each iteration, we use SUS to select individuals (*population_size/2*) from population P_n for the next population P_{n+1}. These selected individuals (upper half of the ranking) will “give birth” to new individuals (substituting the lower half of the ranking) using the crossover operator.

4.2.3.3 Crossover

When two parent individuals are selected, a random cut point is determined to split them into two sub-vectors. Then, crossover swaps the sub-vectors from one parent to the other. Thus, each child combines information from both parents. This operator must enforce the length limit constraint by eliminating randomly some refactoring operations.

For each crossover, two individuals are selected by applying the SUS selection. Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creation of two offspring : P_1' and P_2' from the two selected parents : P_1 and P_2. A random position k is selected. The first k refactorings of P_1 become the first k elements of P_1'. Similarly, the first k refactorings of P_2 become the first k refactorings of P_2'.

Figure 16 shows an example of the crossover process. In this example, P_1 and P_2 are combined to generate two new solutions. The right sub-vector of P_1 is combined with the left sub-vector of P_2 to form the first child, and the right sub-vector of P_2 is combined with the left sub-vector of P_1 to form the second child.

4.2.3.4 Mutation

The mutation operator consists of randomly changing one or more dimensions (refactoring) in the solution (vector). Given a selected individual, the mutation operator first randomly selects some dimensions in the vector representation of the individual. Then the selected dimensions are replaced by other refactoring. Furthermore, the mutation can only modify the controlling elements of some dimensions without replacing the operation by a new one. Figure 16 illustrates the effect of a mutation that replaced the dimension number one *Push Feature (description*) by *Push Feature (version)*.

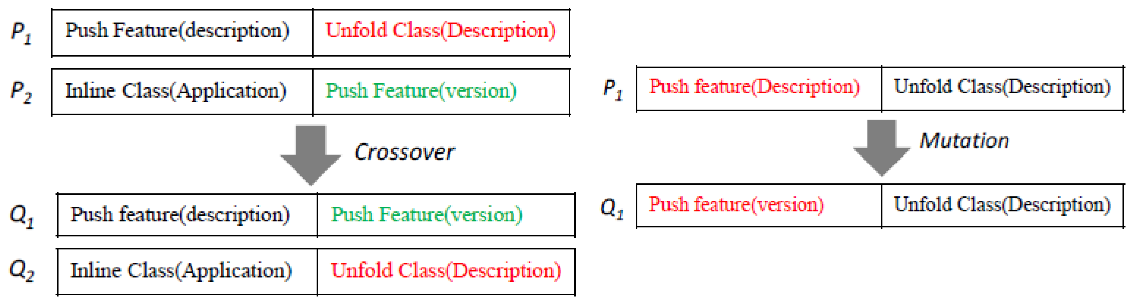

P1
Push Feature(description)
Unfold Class(Description)
P2
Inline Class(Application)
Push Feature(version)
Crossover
Q1
Push feature(description)
Push Feature(version)
Q2
Inline Class(Application)
Unfold Class(Description)
P1
Push feature(Description)
Unfold Class(Description)
Mutation
Q1
Push feature(version)
Unfold Class(Description)

Figure 16 : Example of Cross Over and Mutation Operations

4.2.4 Correctness Function

In general, the encoding of a solution should be formalized as a mathematical function called “fitness function”. The fitness function quantifies the quality of the proposed refactoring sequence. The goal is to define an efficient and simple fitness function in order to reduce the computational complexity. In our work, we are using two fitness functions. The first function is based on a similarity score between the generated models and expected ones to maximize the correctness of detected model changes. The second function counts the number of used refactoring to detect the changes. This function tends to rank the solution higher if it has smaller number of refactoring sequences. In fact, minimizing the number of refactoring improves the comprehension of model evolution by favoring course-grained changes over fine-grained changes and reduces possible redundancies in the change sequence.

The quality of an individual, in terms of correctness, is proportional to the quality of the refactoring operations composing it with respect to the expected revised model. In other words, the best solution is the one that maximizes the similarity between the computed model, obtained after applying the refactoring sequence, and the expected revised model.

In this context, we define the fitness function of a solution, normalized in the range [0, 1], as:

$$f_{norm} = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2}$$

where t is the number of model elements in the expected revised model, p is the number of model elements in the generated model, and a_i has value 1 if the ith element in the generated model exists in the expected one, and value 0 otherwise.

To illustrate the fitness function, we consider the example of Figures 2 and 6. The model V_2 of Figure 3 is considered as the expected revised model and V_2 of Figure 6 as the generated model after applying the refactoring solution. There are two differences between the two models which correspond to class *Description* and attribute *version*. The fitness function in this case is :

$$f_i = \frac{\frac{6}{6} + \frac{6}{8}}{2} = 0.87$$

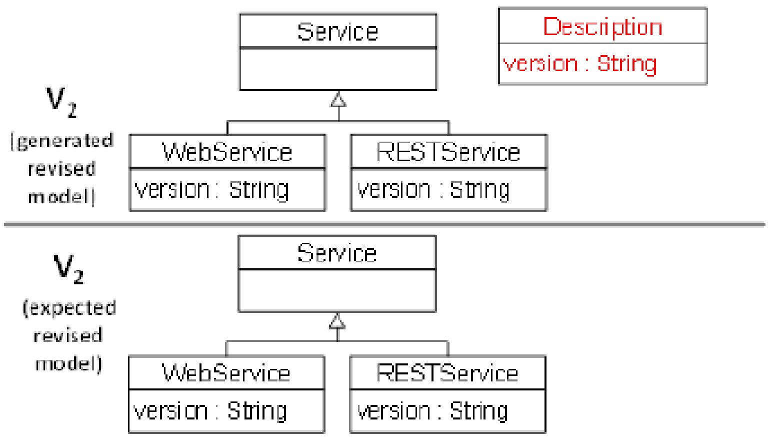


Figure 17 : Comparison between the generated model and the expected model

In our approach, we define the complexity function, to minimize, as the number of refactoring used to detect the changes: $f_{size} = n$, where n is the number of refactorings in the generated solution.

4.3 Validation

4.3.1 Research Questions

In order to evaluate the feasibility of our approach for detecting high-level model changes, we conducted an experiment based on different versions of real-world models extracted from large open source systems [90] [91] [92] [93] [94]. We defined seven research questions that address the applicability and performance in comparison to existing model changes detection approaches, and the usefulness of our multi-objective formulation. The seven research questions are as follows:

RQ1: Search validation (sanity check). To validate the problem formulation of our approach, we compared our NSGA-II formulation with Random Search (RS). If RS outperforms an intelligent search method, we can conclude that there is no need to use a metaheuristic search.

RQ2: To what extent can the proposed approach detect correctly changes between different model versions (in terms of correctness and completeness) and reduce the number of refactorings (reducing redundancies/overlaps and maximizing the use of complex/composite refactorings)?

The next four questions are related to the comparison between our proposal and the state-of-the-art model changes detection approaches.

RQ3.1: How does NSGA-II perform compared to another multi-objective algorithm? It is important to justify the use of NSGA-II for the problem of model changes detection. We compare NSGA-II with another widely used multi-objective algorithm, Multi-Objective Particle Swarm Optimization (MOPSO) [95] using the same fitness functions.

RQ3.2: How does our multi-objective model changes detection formulation perform compared to a mono-objective one? A multi-objective algorithm provides a trade-off between the two objectives where developers can select their desired changes detection solution from the Pareto-optimal front. A mono-objective approach uses a single fitness function that is formed as an aggregation of both objectives and generates as output only one solution of detected changes. This comparison is required to ensure that the solutions provided by NSGA-II and MOPSO provide a better trade-off between the two objectives than a mono-objective approach. Otherwise, there is no benefit to our multi-objective adaptation.

RQ3.3: How does NSGA-II perform compared to existing search-based model changes detection approaches? Our proposal is the first work that treats the problem of model changes detection using a multi-objective approach. However, in [55], a mono-objective genetic algorithm is used for model changes detection using only one objective which is maximizing the similarity with the expected version.

RQ3.4: How does NSGA-II perform compared to existing model changes detection approaches not based on the use of metaheuristic search? While it is very interesting to show that our proposal outperforms existing search-based model changes detection approaches, software developers and engineers will only consider our approach useful [96] if it can outperform other existing tools [37] [96] that are not based on optimization techniques.

The last research question is related to the benefits of our approach for software engineers.

RQ4: Can our multi-objective approach for model changes detection be useful for software engineers in a real-world setting? In a real-world problem, it is important to show that it is useful to consider the correctness, size and understandability of detected model changes. A feed-back from software engineers is required to illustrate the importance of the use of a multi-objective approach for model changes detection in a real-world setting.

4.3.2 Experimental Setup

We chose to analyze the extensive evolution of three Ecore-based metamodels coming from the Graphical Modeling Framework (GMF)[1], an open source project for generating graphical modeling editors. In our case study, we considered the evolution from GMF's release 1.0 over 2.0 to release 2.1 covering a period of two years. We analyzed the revisions of three models, namely the Graphical Definition Metamodel, the Generator Metamodel, and the Mappings Metamodel. Therefore, the respective metamodel versions had to be extracted from GMF's version control system and, subsequently, manually analyzed to determine the actually applied changes between successive metamodel versions. For achieving a broader data basis, we additionally used an existing corpus [97] of 81 releases of four open source Java projects, namely Apache Ant, ArgoUML, JHotdraw and Xerces-J. We used Rational-Rose tool to extract the models from the different open source systems. Apache Ant is a build tool and a library specifically conceived for Java applications. ArgoUML is an open source UML modeling tool. Xerces is a family of software packages for parsing and manipulating XML. It implements a number of standard APIs for XML parsing. JHotdraw is a framework used to build graphic editors. Table 3: Open Source Systems used for studyreports the open source system characteristics of the analyzed systems. The table also reports the number of refactoring operations (as well as the number of different kinds of refactorings) identified for the different systems. More than 17000 refactoring applications on code level have been identified and analyzed manually to filter out those that cannot be applied on the model level. This analysis resulted in 9128 refactoring applications that can be considered on model level and, thus, constitute the input of our experiment.

We choose Xerces-J, JHotDraw, and Apache Ant because they are medium-sized open-source projects and were analyzed in the related work. The initial version of Gantt, GMF, and Apache Ant was known to be of poor quality, which has led to a new major revised version.

[1] http://www.eclipse.org/modeling/gmp/

Xerces-J, ArgoUML, and JHotDraw have been actively developed over the past 10 years, and their design has not been responsible for a slowdown of their developments.

For this experiment, we had to specify all refactoring types manually that have been applied across all metamodel versions. In total, 32 different types of refactoring operations have been applied since we selected only those that can be applied to the model level when using UML class diagram based modeling languages. The evolution of the different models provides a relatively large set of revisions. In total, the evolution of the considered models comprises 81 revisions that involved at least one refactoring operation. Overall, 9128 refactoring operations have been applied, whereas one transition between two revisions contains on average 104 operations. Most of the commits comprise between 1 and 26 refactoring operations. Table 3 describes the number of expected refactorings to be detected by our approach on the different models. These operations cover 32 refactoring types (e.g. move method, move feature, etc.).

Table 3: Open Source Systems used for study

Model	Number of expected refactoring	Number of model elements (min, max)
GMF Map	8	367, 428
GMF Graph	24	277,310
GMF Gen	93	883,1295
Apache Ant	1024	722, 1889
ArgoUML	1839	936, 2092
JHotdraw	1842	2494, 2873
Xerces-J	4423	4323, 5932

To assess the accuracy of our approach, we compute the measured precision and recall originally stemming from the area of information retrieval. When applying precision and recall in the context of our study, the precision denotes the fraction of correctly detected refactoring operations among the set of all detected operations. The recall indicates the fraction of correctly detected refactoring operations among the set of all actually applied composite operations (i.e., how many operations have not been missed). In general, the precision denotes the probability that a detected operation is correct and the recall is the probability that an actually applied operation is detected. Thus, both values may range from 0 to 1, whereas a higher value is better than a lower one.

The quality of our results was measured by two methods: automatic correctness (AC) and manual correctness (MC). Automatic correctness consists of comparing the detected changes to the reference ones, operation by operation using precision (AC-1) and recall (AC-2). AC method has the advantage of being automatic and objective. However, since different refactoring combinations exist that describe the same evolution (different changes but same target model), AC could reject a good solution because it yields different refactoring operations from reference ones. To account for those situations, we also use MC which manually evaluates the detected changes, here again operation by operation. We calculate also NR the number of refactoring used to detect the changes between the different model versions.

4.3.3 Results and Discussion

Our multi-objective approach generates 9504 refactoring operations. Overall, we were able to generate 8948 refactoring operations correctly among all 9128 operations (i.e., around 97%), whereas only less than 500 operations have been incorrectly detected, which leads to a precision of around 94%. It is worth noting that the evolution history of these seven open source systems is very different. The Graphical Definition Metamodel (GMF Graph for short) was extensively modified within only one large revision comprising 24 refactoring operations, but most of them were detected using our technique with a good recall of 91%. The Generator Metamodel (GMF Gen for short) was subjected to 40 revisions. Thus, the evolution of this model is a very representative mixture of different scenarios for the detection of refactoring operations leading to a precision of 95% and a recall of 97%. The evolution of the third model under consideration, the Mappings Metamodel (GMF Map for short), contained four revisions and in each revision at maximum three refactoring operations have been applied.

As the studied evolution of the remaining systems covers a time period of ten years (as opposed to the considered time period of two years of the GMF case study), they have a larger number of refactorings to be detected. However, our multi-objective proposal performs also well on the large evolutions of the remaining systems. For Apache, most of detected changes are correct with 95% and 98% respectively as precision and recall. For ArgoUML and JHotDraw, they have approximately the same number of refactoring to detect and our approach detected most of them with an average of 95% of precision and 96% of recall. Xerces-J is the larger system that we studied with more than 4197 refactoring applied over 10 years and our proposal successes to detect almost all of them with more than 92% of precision and recall. Thus, overall we can conclude that using our approach we could identify most of the applied operations correctly with an average of 93% of precision and 95% of recall. We noticed that our technique does not have a bias towards the types of refactoring since most of them were detected.

With regards to manual correctness (MC), the precision and recall scores for all the seven models were improved since we found interesting refactoring alternatives that deviates from the reference ones proposed by the experts. For instance, MC for the GMF graph model was improved from 95 to 100% and 87 to 92% respectively for the precision and recall. In addition, we found that sequence of applying the refactoring is sometimes different between generated refactoring and reference ones. We found that sometimes a different sequence can reduce the number of refactoring used to detect changes.

Table 4: NSGA-II Detection Correctness for change detection in Evolving Models.

Model	AC1: Precision	AC2: Recall	MC1: Precision	MC2: Recall
GMF Map	7/7= 100%	7/8= 87%	7/7= 100%	7/8= 87%
GMF Graph	20/22= 91%	20/24= 84%	21/22= 95%	21/24= 88%
GMF Gen	90/94= 95%	90/93= 97%	92/94= 97%	92/93= 98%
Apache Ant	1012/1067= 95%	1012/1024 = 98%	1019/1067 = 95%	1019/1024 = 99%
Argo UML	1831/1904 = 96%	1831/1839 = 99%	1834/1904 = 96%	1834/1839 = 99%
JHot-draw	1791/1874 = 95%	1791/1842 = 97%	1809/1874 = 96%	1809/1842 = 98%
Xerces-J	4197/4536 = 92%	4197/4423 = 94%	4227/4536 = 93%	4227/4423 = 95%

The quality of our results was measured by two methods: automatic correctness (AC) and manual correctness (MC). Automatic correctness consists of comparing the detected changes to the reference ones, operation by operation using precision (AC-P) and recall (AC-R). AC method has the advantage of being automatic and objective. However, since different refactoring combinations exist that describe the same evolution (different changes but same target model), AC could reject a good solution because it yields different refactoring operations from reference ones. To account for those situations, we also use MC which manually evaluates the detected changes, here again operation by operation. We calculate also NR the number of refactoring used to detect the changes between the different model versions.

We compared our results with an existing work proposed by Langer et al. [37] that introduced an approach for defining and detecting composite operations for software models based on model transformations by-example techniques. The approach allows defining composite operations on models by demonstrating them on examples from which the general transformations (including the pre and post conditions of the operations expressed as OCL constraints) is semi-automatically derived. These transformations can be used for applying the composite operations as well as detecting applications of them between two versions of a model. A composite operation is detected either if its pre and post conditions are fulfilled or if the pre conditions can be established by an already detected composite operation, but the post conditions always have to be fulfilled by the revised version for all detected composite operations.

We also compared our approach to UMLDiff a refactoring detection technique not based on heuristic search [96]. UMLDiff includes three main steps. First, facts regarding design-level entities and their relations in each individual version of the system are extracted from the source-code versions of the system. Then, the designs of subsequent system versions are pair-wise compared to determine how the basic entities and relations have changed from one version to the next. Finally, queries, corresponding to typical refactorings change patterns, are applied to the change-facts database to extract instances of particular refactorings and their participant elements.

To answer the first research question RQ1, we implemented a "random" solution in which refactorings were randomly generated at each iteration. The obtained Pareto fronts were compared for statistically significant differences with NSGA-II using *IHV*, *IGD* and *IC*.

Table 5 : The significantly best algorithm among random search, NSGA-II and MOPSO (Not sign. diff. means that NSGA-II and MOPSO are significantly better than random search, but not statistically different).

System	IHV	IGD	IC
GMF Map	NSGA-II	NSGA-II	NSGA-II
GMF Graph	NSGA-II	MOPSO	MOPSO
GMF Gen	NSGA-II	MOPSO	MOPSO
GanttProject	NSGA-II	NSGA-II	NSGA-II
Xerces-J	NSGA-II	NSGA-II	NSGA-II
JHotDraw	Not sign. diff.	NSGA-II	NSGA-II
ApacheAnt	NSGA-II	Not sign. diff.	Not sign. diff.
MROI-Ford	NSGA-II	NSGA-II	NSGA-II

Table 5 : The significantly best algorithm among random search, NSGA-II and MOPSO (Not sign. diff. means that NSGA-II and MOPSO are significantly better than random search, but not statistically different).confirms that both multi-objective algorithms NSGA-II and MOPSO are better than random search

based on the three quality indicators IHV, IGD and IC on all the seven open source systems and the industrial project. The Wilcoxon rank sum test showed that in 51 runs both NSGA-II and MOPSO results were significantly better than random search. Table 5 also shows the overview of the results of the significance tests comparison between NSGA-II and MOPSO. NSGA-II outperforms MOPSO in most of the cases: 17 out of 24 experiments (71%). MOPSO outperforms the NSGA-II approach only in GMF Gen and GMF Graph, which are the smallest open source system considered in our experiments, having a low number of operations to detect. In particular, NSGA-II outperforms MOPSO in terms of IHV values in 7 out 8 experiments with one 'no significant difference' result.

To answer RQ2, we evaluated the average of NR, AC-P, AC-R and MC scores for non-dominated changes detection solutions proposed by NSGA-II.

In this section, we evaluate the performance of NSGA-II to find good trade-offs between the two objectives of minimizing the number of refactorings and maximizing the correctness of detected changes.

Table 6 : NSGA-II Detection Correctness. Median number of disabled refactorings on 51 independent runs. The results were statistically significant on 51 independent runs using the Wilcoxon rank sum test with a 95% confidence level (α < 5%).

Model	AC-Precision	AC-Recall	MC	NR
GMF Map	14/14= 100%	14/17= 82%	14/17= 82%	17
GMF Graph	31/36= 77%	31/39= 79%	33/39= 91%	39
GMF Gen	104/112= 92%	104/122= 85%	108/122= 88%	122
GanttProject	69/72= 96%	69/84 = 82%	69/84 = 87%	84
Xerces-J	81/86 = 96%	81/92 = 88%	86/94 = 91%	92
JHot-draw	77/81 = 95%	77/86 = 89%	82/93 = 88%	86
Apache Ant	242/258 = 92%	242/267 = 90%	254/267 = 95%	267
MROI-Ford	314/327 = 96%	314/331 = 95%	328/341 = 96%	330

It confirms that our NSGA-II adaptation was successful in detecting model changes and describing them with a minimum number of refactorings. Overall, our approach was able to detect 932 refactoring operations correctly among all the 1037 expected operations (i.e., around 90% of recall), whereas 986 refactorings have been detected, which leads to a precision of around 94%. It is worth noting that the evolution history of the different systems is very different. The Graphical Definition Metamodel (GMF Graph for short) was extensively modified within only one large revision comprising 36 refactoring operations, but most of them were detected using our technique with a good recall of 80%. The Generator Metamodel (GMF Gen for short) was subjected to 40 revisions. Thus, the evolution of this model is a very representative mixture of different scenarios for the detection of refactoring operations leading to a precision of 92% and a recall of 85%. The evolution of the third model under consideration, the Mappings Metamodel (GMF Map for short), contained four revisions and in each revision at maximum five refactoring operations have been applied that all of them were detected using our approach.

In this section, we compare our NSGA-II adaptation to the current, state-of-the-art model changes detection approaches. To answer RQ3.1, we compared NSGA-II to another widely used multi-objective algorithm, MOPSO, using the same adapted fitness function as described earlier. A more qualitative evaluation is presented in Figure 18 which illustrates the box plots obtained for the multi-objective metrics on the different projects. We see that for almost all problems the distributions of the metrics values for NSGA-II have smaller variability than for MOPSO. This fact confirms the effectiveness of NSGA-II over MOPSO in finding a well-converged and well-diversified set of Pareto-optimal model changes detection solutions. Similarly, our results show that NSGA-II based multi objective search technique outperforms both mono objective search technique as well as other existing change detection schemes present today. In conclusion, we answer RQ3, the results support the claim that our NSGA-II formulation provides a good trade-off between both objectives, and outperforms on average the state-of-the-art of model changes approaches, both search-based and non-search-based ones.

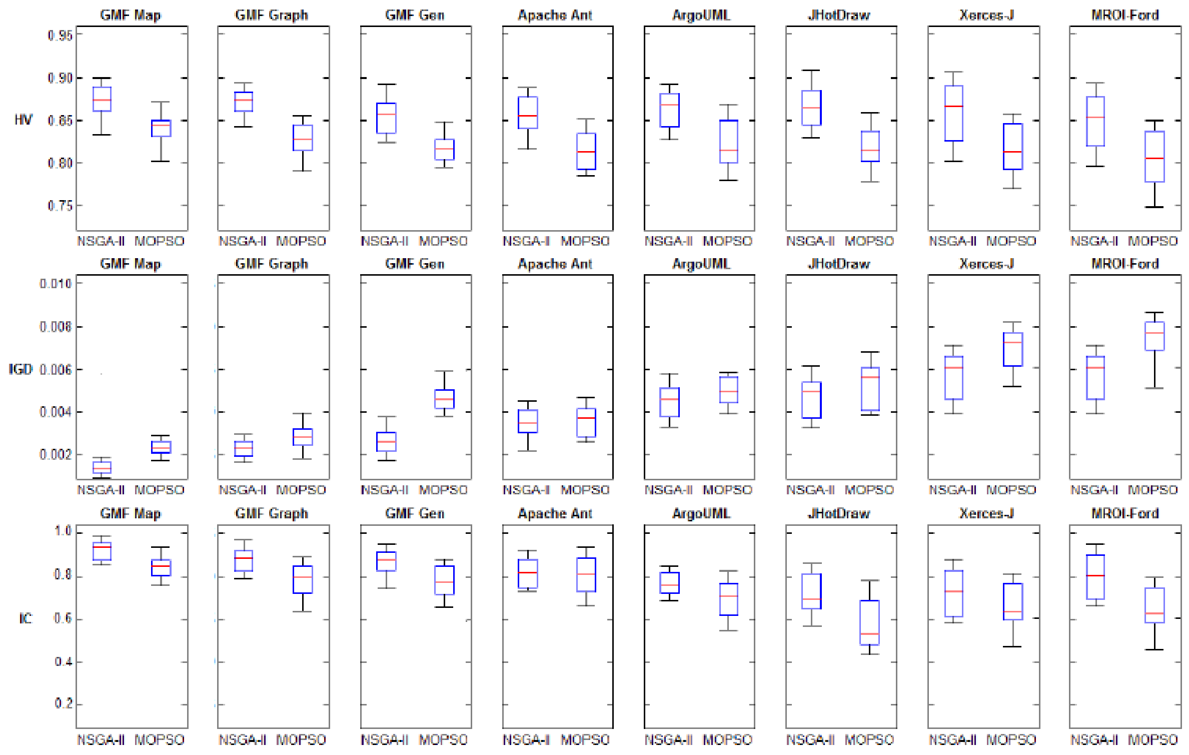


Figure 18 : Boxplots using the quality measures (a) HV, (b) IGD, and (c) IC applied to NSGA-II and MOPSO.

Figure 19 depicts the different Pareto surfaces obtained on two open source systems and one industrial project (GMFGraph, JHotDraw and MROI-Ford) using NSGA-II to optimize both objectives related to minimizing the number of refactorings and maximizing the correctness (similarity to the expected version). Due to space limitations, we show only some examples of the Pareto-optimal front approximations obtained which differ significantly in terms of size. Similar observations were obtained on the remaining systems. The 2-D projection of the Pareto front helps software engineers to select the best trade-off solution between the two objectives based on their own preferences. Based on the plots, the software developer could decrease the number of detected refactorings while controlling visually the correctness. In this way, the developer can select the preferred model changes detection solution to realize.

We found that the knee-point solution with lower number of refactorings did not include any redundancy and most of the refactorings are complex ones (not atomic). Hence we conclude that RQ4 is affirmed and that the multi-objective approach has value for software engineers in a real-world setting that can help programmers to understand detected changes while maximizing the correctness. In fact, all the interviewed developers mention that they could easily understand the solution with composite changes comparing atomic ones.

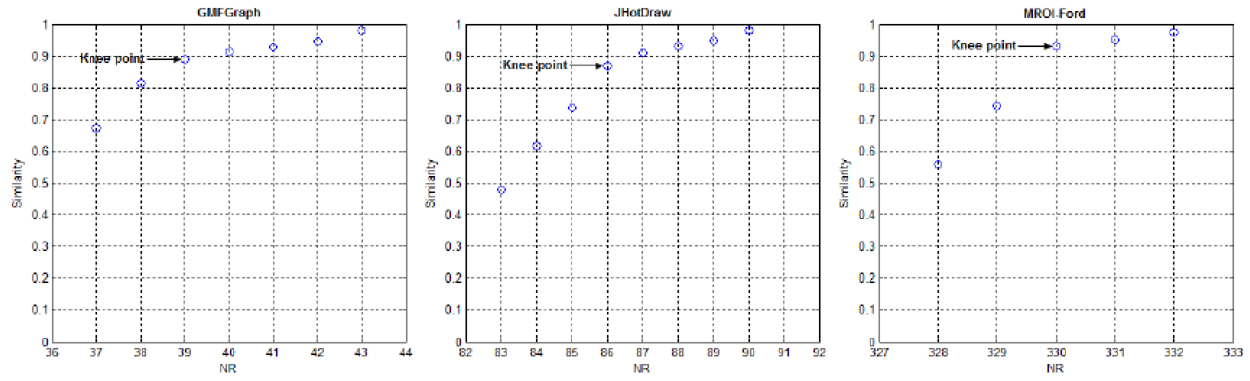


Figure 19 : Pareto fronts for NSGA-II obtained on GMF Graph (small), JHotDraw (medium), and MROI-Ford (large).

Chapter 5: Model Merging

5.1 Introduction

When models evolve or are changed in parallel, they have to be merged eventually to obtain a consolidated model. This may be needed to consolidate product lines, or design from different development teams focused on singular objective or merging development trunks with the goal of increasing model reuse and then eventually code reuse. Several approaches have been proposed for detecting the operations that have been applied in parallel by developers. Once the applied operations are available, conflict detection algorithms are used to identify pairs of operations that interfere with each other [98]. In this regard, a conflict denotes a pair of operations, where one operation masks the effect of another or one operation disables the applicability of another operation. An example for the former is a pair of parallel operations that update the same feature in a model with different values. Such scenarios occur frequently if composite operations (a sequence of cohesive atomic changes) are applied, because they may have potentially complex preconditions that may easily be invalidated by other parallel operations.

For resolving conflicts, empirical studies [99] showed that users prefer to work with a tentative merged model acting as a basis for resolving possible conflict resolutions, instead of working with the list of operations in terms of choosing to reject one or the other conflicting operation for creating a merged model. Few approaches respect this preference and produce a merged model by applying all non-conflicting operations; conflicting operations are omitted. However, especially in case of a large number of conflicts, many operations are not merged with this strategy, leading to a tentative merged model that does not reflect the maximal combined effect of the parallel operations. Furthermore, the majority of existing works [11] [12] [6] [100] [14] [10] treat the applied refactoring operations to be merged with equal importance. However, in a real world scenario these operations have different importance scores that can depend on the type of the refactoring and the context of the refactoring application. Thus, in existing work the developers cannot integrate their priority preferences concerning the importance of some refactorings that should be included in the merging process.

As described in Figure 20 the proposed multi-objective approach aims to find the best trade-off between minimizing the number of conflicts and maximizing the number of successfully applied *important* operations. A conflict denotes pairs of operations where one operation disables the applicability of the other. Whether one operation disables the other, however, often depends on the order in which they are applied. The scheme aims to find an operation sequence that minimizes the number of disabled operations among all parallel operations. The system takes as input the initial model and the revised ones (i.e., the different parallel versions), a list of the applied operations and an importance score for each composite operation, and generates as output a set of merging solutions that reflect the best trade-off between the two conflicting objectives. The importance score for each composite operation can be calculated manually or automatically. In following experiments the importance score is calculated using quality metrics that formalize the complexity of a class that will be modified and a classification of the types of composite operations based on their complexity. In general, larger composite operations (combinations of atomic and smaller composite operations) are more important than smaller ones since they introduce more significant changes to the design.

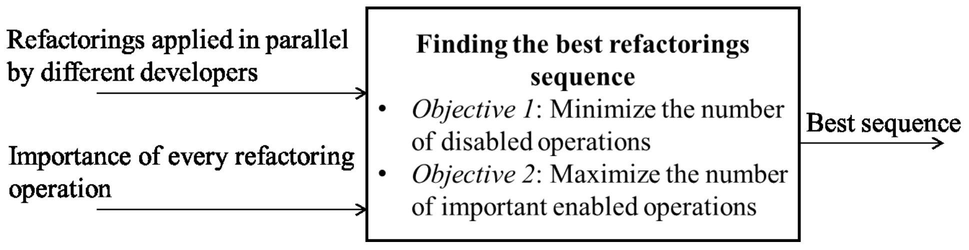

Refactorings applied in parallel by different developers
Importance of every refactoring operation
Finding the best refactorings sequence
• Objective 1: Minimize the number of disabled operations
• Objective 2: Maximize the number of important enabled operations
Best sequence

Figure 20 : Multi-objective model merging: overview

5.1.1 Model Merging: A Motivating Example

To present the challenges associated with the problem of model merging we present a following motivating example. The starting point is the UML class diagram shown in Figure 21. This version of a person management system has been subject to parallel evolution by two developers who concurrently applied a set of atomic and composite operations leading to the version shown in Figure 21 *b* and *c* respectively.

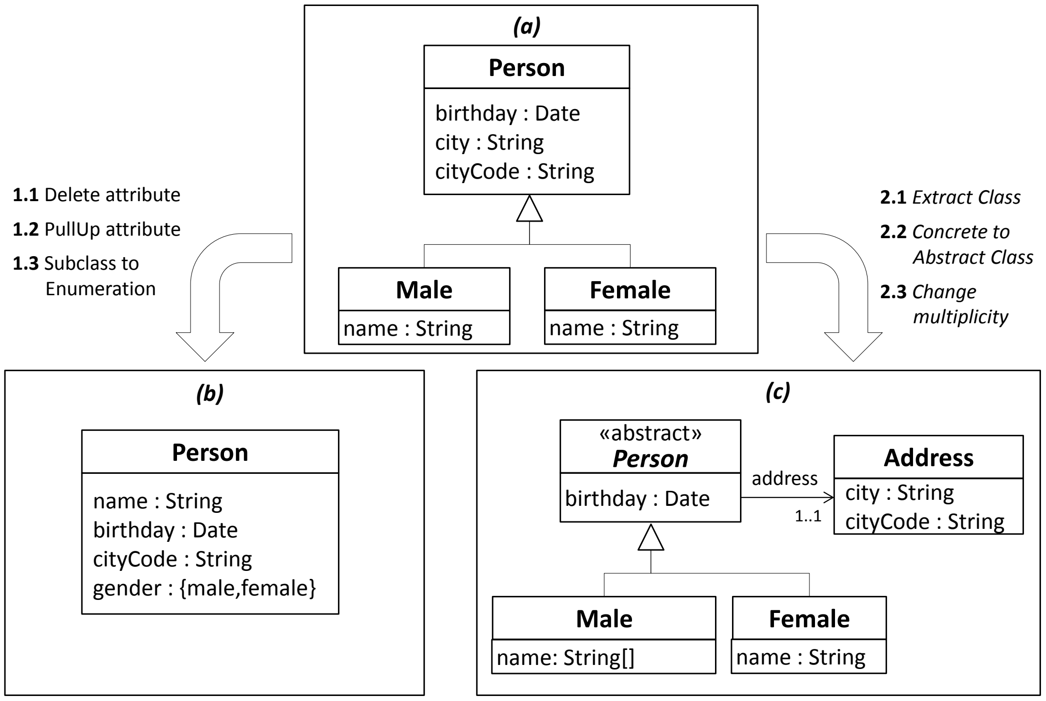


Figure 21: Parallel Evolution of a Class Diagram: (a) Initial Model v_0, (b) Revised Model v_{1a}, (c) Revised Model v_{1b}.

First, developer *A* deletes the *city* attribute of class *Person*, because she identifies that this attribute is redundant, as the information is already covered by *city Code* attribute. Furthermore, developer *A* is applying the *Pull Up Attribute* refactoring [101] in a second step: the *name* attribute of classes *Male* and *Female* are substituted by introducing this attribute to the class *Person*. This refactoring is represented by one composite operation which contains all atomic operations, i.e., adding the *name* attribute to the super class and deleting the *name* attributes in the subclasses. The precondition of this refactoring is clearly fulfilled: both subclasses of class *Person* have the *name* attribute with the same property values, i.e., same data type and multiplicities. Finally, developer *A* applies another refactoring: substituting the now empty subclasses by an enumeration-typed attribute. The precondition for applying this refactoring is that the superclass receiving the enumeration-typed attribute is a concrete class and not an abstract one.

Developer *B* applies the *Extract Class* refactoring to create an explicit class for the address information, and subsequently, the *Concrete to Abstract Class* refactoring. Again, these two refactorings are represented by composite operation consisting of several atomic operations. Please note that the *Concrete To Abstract Class* refactoring comprises a dedicated precondition that has to be fulfilled, namely the subject for this refactoring has to provide at least one concrete subclass. Furthermore, developer *B* changes the upper bound cardinality of the *name* attribute of class *Male* from one to many by marking this attribute as an array. The resulting model is depicted in Figure 21*c*.

Now a naive operation-based merge approach may apply the operations of developer *A* to the initial model, and subsequently, on this intermediate version, the changes of developer *B*. However, in this sequence, the *Extract Class* refactoring is not applicable anymore, because the *city* attribute is already missing. Furthermore, the *Concrete to Abstract Class* refactoring is disabled as well due to the unsatisfied precondition for the intermediate version that is produced by applying the operations of developer *A*. Starting with the operations of developer *B* and continuing with the operations of developer *A*, also leads to two disabled operations: the *Pull Up Attribute* refactoring cannot be executed, because the *name* attributes in both subclasses have different cardinalities after applying the operations by developer *B*. Thus, the precondition of the refactoring is not fulfilled, and consequently, the operation cannot be applied. The same is true for the *Subclass to Enumeration* refactoring.

Now we may consider whether we can find an operation sequence that contains a higher number of applicable operations by *intermingling* the operations of both developers with each other in order to maximize the combined effect of both developers. In this example, 720 (6!) different operation sequences would have to be considered since changing the order of applying the operations sequence will lead to different solutions. In fact, for this example, sequences exist that allow for applying *more operations successfully* by intermingling the operations of developer *A* and *B* than using a phasing approach. For instance, executing the *Extract Class* refactoring, deleting the *city* attribute, executing the *Pull Up Attribute* and *Subclass to Enumeration* refactorings, and finally, setting the upper bound cardinality represents one operation sequence that contains more enabled operations than the pure phasing approach. However, one operation is disabled, namely the *Concrete to Abstract Class* refactoring is in conflict with the *Subclass to Enumeration* refactoring irrespectively of the application sequence. Due to the fact that both operations are mutual exclusive, it is only possible to enable one of them. Selecting the operation that should be applied rather than disabled requires having a measure of importance of the operation. One measure is the magnitude of an operation, i.e., the number of atomic changes a composite change consists of. Consequently, the merge solution shown in Figure 21*b* is preferable over the merge in Figure 21*c*, because the number of disabled operations is equal in both b and c, but the overall magnitude of all enabled operations is higher in Figure 21*b*.

It is important to stress that for six operations, an exhaustive based approach is applicable, but when doubling the number of operations, we already have to explore over 470 million combinations because *n!* solutions exist where *n* is the number of applied operations. Thus, in the next section a scalable approach to solve this problem is presented that is able to consider the importance of operations, as well as to explore the sequences that maximize the application of operations.

5.1.2 Challenges of Model Merging Problem

The goal of our approach is to construct a tentative merged model that minimizes the number of disabled operations and maximizes the number of important enabled ones. Therefore, it is an inherently multi-objective optimization problem which needs an algorithm to compute an optimal sequence of merging operations in terms of finding trade-offs between minimizing the number of operations that are disabled by preceding operations and maximizing the number of important operations that are enabled.

The search-based process takes as inputs the sequences of operations that have been applied concurrently to a model by an arbitrary number of developers with an importance score for each composite operation. These sequences can be detected using operation detection algorithms presented in previous work [35]. Note that these sequences may be obtained alternatively by tools that record operations directly in the modeling editor. The sequences are composed of operation applications, thereby each entry in a sequence states the operation type as well as the elements on which it has been applied. Having these sequences at hand, we may now combine them into one common sequence of operations and compute the number of disabled operations. Therefore, we use composite operation specifications that contain explicitly specified preconditions in combination with a condition evaluation engine [100] to verify whether the preconditions of each operation in a sequence are fulfilled in a certain state of a model after the preceding operations in the sequence have been applied. If we determine an operation with invalid preconditions in a certain state of the model, we consider this operation to be disabled in the respective operation sequence.

The process of generating a solution can be viewed as the mechanism that finds the best order among all possible operation sequences that minimizes the number of disabled operations and maximizes the number of enabled important ones. The size of the search space is determined not only by the number of operations applied by the different developers on the same model, but also by the order in which they are applied. Due to the large number of possible refactoring sequences and the two conflicting objectives to optimize, we considered refactoring merging as a multi-objective optimization problem.

It is therefore evident that a model merging problem is intrinsically a challenging problem with multi dimensional dependencies for finding an optimal solution. The potentially huge size of search solution space is a primary obstacle. Unless explosively fast and resilient multi objective algorithms are employed, the probability of finding optimal solution shall be very low. Furthermore, the problem of ranking different operations is also subject to contention. To achieve relevant and applicable results developers need to define concise ranking for different operation sequences, along with pre and post conditions for each sequence. These pre and post conditions are tantamount in evaluating the potential conflict or disabling operations. The correctness of these conditions dictate the success and relevance of merged model.

5.2 Approach Overview

5.2.1 Detection Scheme

We propose a multi-objective formulation of the model merging problem that takes the importance of the refactoring operations to merge into account. Consequently, there are two objective functions in problem formulation: (1) minimize the number of disabled refactorings, and (2) maximize the number of important enabled refactorings. Analytically speaking, the formulation of the model merging problem can be stated as follows:

$$\begin{cases} Minimize\ f_1(S) = ndo \\ Maximize\ f_2(S) = \sum_{i=1}^{nb\_enabled\_ref} Importance(x_i) \end{cases}$$

where *ndo* is the number of disabled operations, x_i is the *i*-th enabled operation in the sequence *S*, *nb_enabled_ref* is the number of enabled refactorings and *Importance*(x_i) is the importance of the operation x_i in *S*. The importance of the operation is independent from its order in the vector.

The goal is to define an efficient and simple—in the sense that it is not computationally expensive—fitness function in order to reduce the computational complexity. This function should evaluate the number of disabled operations. Therefore, previously developed tools are being used to specify composite operations including their preconditions in combination with an engine for evaluating the conditions in a certain model state (see [35] for details). As evaluating conditions can be rather expensive, we only compute whether one operation in a sequence disables another for each possible pair-wise combination of operations in advance, instead of checking the preconditions of each operation with the combined effect of every operation that precedes the operation in the sequence. Please note that this approach might miss detecting some disabled operations in certain scenarios: the preconditions of an operation might be fulfilled with respect to each *single* preceding operation (checking operations pair-wise), but the preceding operations *in combination* might still invalidate the preconditions of a subsequent operation.

The information on which operation in a sequence disables the other is represented in terms of a matrix $n \times n$ where n is the number of operations applied originally by the different developers in total (after eliminating duplicates). Each item in this matrix represents a combination of two operations and holds a value of either 0 or 1: if an operation i disables the operation j then, the item (i,j) in the matrix takes the value 1, otherwise it takes 0. Based on this matrix, we may determine easily the number of disabled operations for a specific operation sequence by summing up all values in the matrix. The $n\ x\ n$ matrix is generated to each sequence (solution), and the matrix values consider the specific sequence of operations.

We define the refactoring importance as follows:

$$Importance(x_i) = type\_importance\,(\mathrm{x_i}) + \frac{\sum_{j=1}^{numberMetrics} M_j(c_i)}{numberMetrics}$$

such that M_j represents a software metric and c_i is the class where the refactoring x_i will be applied. Thus, we defined the class importance using a set of software metrics. In our experiments, we considered the following metrics:

Weighted Methods per Class (WMC), Response for a Class (RFC), Lack of Cohesion of Methods (LCOM), Cyclomatic Complexity (CC), Number of Attributes (NA), Attribute Hiding Factor (AH), Method Hiding Factor (MH), Number of Model Elements (NME), Coupling Between Object Classes (CBO), Number of Association (NAS), Number of Classes (NC), Depth of Inheritance Tree (DIT), Polymorphism Factor (PF), Attribute Inheritance Factor (AIF), Number of Children (NOC).

We have normalized each metric value into the interval [0, 1] using the following formula: M_norm(x) = (M(x) - M_min)/(M_max - M_min) where M_max and M_min correspond respectively to the maximal and minimal metric value taken from classes in the system. In this way, all values are between 0 and 1.

We use a model that considers two categories of operations: an operation can be a Low-Level Operation (LLO) or a High-Level Operations (HLO). HLO is composed by the combination of two or more operations and LLO is an elementary/atomic operation. We classify the following basic operations in the LLO category: *create_class*, *delete_method*, *add_field*, *move_method*, *rename_method*, *create_relationship*, etc. The HLO or complex refactoring are *extract_class*, *extract_subclass*, *pull_up_method*, *push_down_field*, etc. In fact, the application of these complex refactorings includes the execution of some LLO operations. For example, *extract_class* includes *create_new_class*, *move_method*, etc. In our experiments, we defined the *type_importance* measure as follows:

$$type\_importance(x_i) = \begin{cases} 1, \text{if } x_i \in \text{HLO} \\ 0.5, \text{if } x_i \in \text{LLO} \end{cases}$$

There are of course many ways in which the importance of classes and code changes could be measured, and one of the advantages of the search-based approach is that this definition could be easily replaced with a different one. In summary, the basic idea behind this work is to take into account the importance of the operations to merge while maximizing simultaneously the number of enabled refactorings. These two objectives are in conflict. Thus, the goal is to find a good compromise. In fact, once the bi-objective trade-off front is obtained, the user can navigate through this front in order to select his/her preferred merging solution. This is achieved through sacrificing some degree of number of enabled refactorings while gaining in terms of including important ones in the merging process.

5.2.2 Solution Coding

We use the well-known multi-objective evolutionary algorithm NSGA-II [82] to try to solve the model merging problem. As noted by Harman et al. [83] [84], a generic algorithm like NSGA-II cannot be used 'out of the box' – it is necessary to define problem-specific genetic operators to obtain the best performance. To adapt NSGA-II to our problem, the required steps are to create: (1) solution representation, (2) solution variation and (3) solution evaluation. We examine each of these in the following.

5.2.2.1 Solution Representation

To represent a candidate solution (individual), a vector is being used which contains all operations that have been applied by the developers in parallel, where each item in the vector represents a single operation (with links to the elements to which it is applied) and the order of operations in this vector represents the sequence in which the operations are applied. Please note that some operations can be eliminated in case they are equivalent; that is, two developers applied the same operation to the same model elements. Thus, we exclude duplicates. Consequently, all vectors, each representing one candidate solution, have the same number of dimensions that corresponds to the number of all parallel operations applied by all developers. Figure 22 depicts a possible population of operation sequences where *R*** refers to the label of the respective refactoring. For instance, the solutions represented are composed of five dimensions corresponding to five operations proposed by two different developers. All the solutions have the same length, but they constitute a different order.

The proposed algorithm first generates a population randomly from the list of all operations. Second, crossover and mutation operators are used to generate new populations in the next iterations as explained in the following.

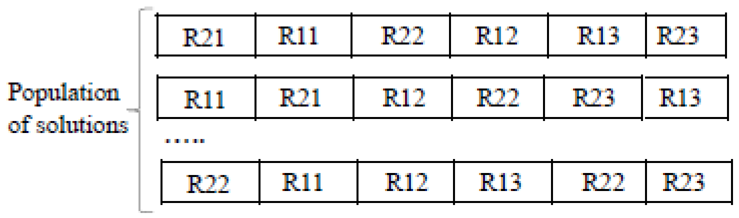


Figure 22 : Population of operation sequences

5.2.2.2 Solution Variation

In a search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards better solutions. We used the principle of the Roulette wheel [82] to select individuals for mutation and crossover. The probability to select an individual for crossover and mutation is directly proportional to its relative fitness in the population. In each iteration, we select *population_size / 2* individuals from the population pop_i to form population pop_{i+1}. These (*population_size / 2*) selected individuals will "give birth" to another (*population_size / 2*) new individuals using a crossover operator. Therefore, two parent individuals are selected, and a few dimensions picked on each one. When applying the crossover, we ensure that the length of the vector remains the same. The one point crossover operator allows creating two offspring *p'1* and *p'2* from the two selected parents *p1* and *p2*. It is defined as follows: a random position, *k*, is selected. The first *k* operations of *p1* become the first *k* elements of *p'1*. Similarly, the first *k* operations of *p2* become the first *k* operations of *p'2*.

The crossover operator could create a child that contains redundant operations. In order to resolve this problem, for each obtained child, we verify whether there are redundant operations or not. In case of redundancy, we replace the redundant operation by a randomly chosen one without causing another redundancy. We have used the one-point crossover since this operator has demonstrated its effectiveness in solving permutation based problems of similar degree e.g. the traveling salesman problem [102].

The mutation operator can be applied to pairs of dimensions of the vector selected randomly. Given a selected solution, the mutation operator first randomly selects one or many pairs of dimensions of the vector. Then, for each selected pair, the dimensions are swapped.

5.2.2.3 Solution Evaluation

The solution is evaluated based on the two conflicting objective functions. Maximizing the number of enabled refactorings maximizes the number of detected conflicts. Thus, the two objectives are conflicting since we want to maximize the number of important enabled refactorings and minimize the number of detected conflicts. In our case, we are using a multi-objective approach to maximize the number of enabled important refactorings and minimize the number of conflicts. To fix a number of conflicts then the number of omitted refactorings increases. For this reason, the number of omitted refactorings is conflicting with the number of enabled important refactorings. In addition, we found that important refactorings create, in general, conflicts. For this reason, a large increase of the number of enabled refactorings does not mean that the importance score is improved since most of the enabled ones are "simple" and not complex/important.

A widely used method in optimization to verify if two objectives are conflicting or not is the following [82]: If we optimize one of the objectives, the resulting solution may not be the optimum of the second objective and vice versa. Figure 23 illustrates the conflict between the two considered objectives. In fact, we recorded the values of each objective at the beginning of evolutionary process (generation 0) and at the end (generation 150) using NSGA-II. Figure 23 shows the intermediate values (between generation 0 and 150) resulting in the straight lines when trying to optimize the first objective (number of disabled refactorings). It is clear that when the number of disabled refactoring decreases, the importance of enabled refactoring decreases too since most of the enabled ones are atomic changes and not important - the atomic changes take place of the important complex refactorings.

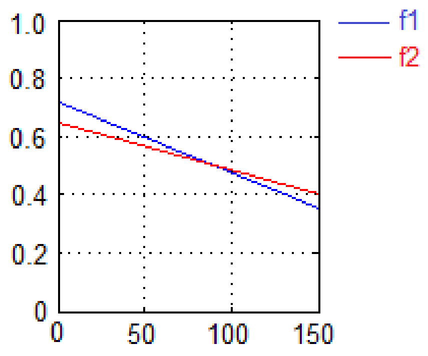


Figure 23 : Illustrating the conflict relation between the two objectives (f1: number of disabled refactorings, f2: importance of enabled refactoring).

5.3 Validation

5.3.1 Research Questions

We defined six research questions that address the applicability, performance in comparison to existing merging approaches, and the usefulness of our robust multi-objective merging approach. The six research questions are as follows:

RQ1: Search validation (sanity check). To validate the problem formulation of our approach, we compared our NSGA-II formulation with Random Search (RS). If RS outperforms an intelligent search method, we can conclude that there is no need to use a metaheuristic search.

RQ2. To what extent can the proposed approach reduce the number of disabled operations and maximize the number of enabled important operations? It is important to evaluate the performance of our model merging approach, based on NSGA-II, when applied to real-world scenarios.

The next four questions are related to the comparison between our proposal and the state-of-the-art model merging approaches.

RQ3.1: How does NSGA-II perform compared to another multi-objective algorithm? It is important to justify the use of NSGA-II for the problem of model merging. We compare NSGA-II with another widely used multi-objective algorithm, MOPSO (Multi-Objective Particle Swarm Optimization), [95] using the same fitness functions.

RQ3.2: How does our multi-objective model merging formulation perform compared to a mono-objective one? A multi-objective algorithm provides a trade-off between the two objectives where developers can select their desired merging solution from the Pareto-optimal front. A mono-objective approach uses a single fitness function that is formed as an aggregation of both objectives and generates as output only one merging solution. This comparison is required to ensure that the solutions provided by NSGA-II and MOPSO provide a better trade-off between the two objectives than a mono-objective approach. Otherwise, there is no benefit to our multi-objective adaptation.

RQ3.3: How does NSGA-II perform compared to existing search-based model merging approaches?

RQ3.4: How does NSGA-II perform compared to existing model merging approaches not based on the use of metaheuristic search? While it is very interesting to show that our proposal outperforms existing search-based merging approaches, developers will consider our approach useful, if it can outperform other existing model merging tools [10] that are not based on optimization techniques where the operations are applied as they arrive without trying the find the best sequence/order of applying them.

RQ4: Can our multi-objective approach for model merging be useful for software engineers in a real-world setting? In a real-world problem, it is important to show that it is useful to consider the importance/priority score related to each refactoring when merging models. Some scenarios are required to illustrate the importance of the use of a multi-objective approach for model merging in a real-world setting.

5.3.2 Experimental Setup

We chose to analyze the extensive evolution of three Ecore metamodels coming from the Graphical Modeling Framework (GMF) [90], an open source project for generating graphical modeling editors. We considered the evolution from GMF's release 1.0 over 2.0 to release 2.1 covering a period of two years. For achieving a broader data basis, we analyzed the revisions of three models, namely the Graphical Definition Metamodel (GMF Graph), the Generator Metamodel (GMF Gen), and the Mappings Metamodel (GMF Map). Therefore, the respective metamodel versions had to be extracted from GMF's version control system and, subsequently, manually analyzed to determine the actually applied operations between successive metamodel versions. In addition to GMF, we used UML class diagrams extracted from four open source projects: GanttProject (Gantt for short) [103], JHotDraw [104], ApacheAnt [91] and Xerces-J [91]. We considered the evolution across three versions of Gantt (v1.7, v1.8, and v1.9.10), three versions of JHotDraw (v5.1, v5.2, and v5.3), four versions of ApacheAnt (v1.6.1, v1.6.2, v1.6.3 and v1.6.4) and four versions of Xerxes-J (v1.4.4, v2.5.0, v2.6.0, and v2.6.1). Xerces-J is a family of software packages for parsing XML. GanttProject is a cross-platform tool for project scheduling. ApacheAnt is a build tool and library specifically conceived for Java applications. Finally, JHotDraw is a GUI framework for drawing editors. Table 7 summarizes for each model evolution scenario the number of applied refactorings, as well as the number of model elements for the smallest and largest model version.

Additionally, we had to specify all operation types (i.e., their comprised atomic operations and preconditions) that have been applied across all versions leading to 38 different types of operations. The evolution of the analyzed models provides a relatively large set of revisions containing overall 659 different applications of the operation types. Since we considered the evolution of several versions of the studied models, Table 7 describes also the minimum and maximum number of model elements between the different versions of a given model.

Due to the lack of existing parallel revision histories that we could have used for evaluating our approach, we emulate parallel evolution by dividing the applied operations from the single revisions into parallel sequences of operations manually and asked five graduate students to additionally modify different model fragments of these open source systems in order to cause disabled operations in the considered evolutions.

Table 7: Systems studied in Model Merging experiments.

Model	Number of refactorings	Number of elements (min, max)
GMF Map	14	367, 428
GMF Graph	36	277,310
GMF Gen	112	883,1295
GanttProject	72	451, 572
Xerces-J	86	1698,1732
JHotDraw	81	985, 1457
ApacheAnt	258	2166, 2489

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for a same problem instance from one run to another. For this reason, our experimental study is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 99% confidence level ($\alpha = 1\%$). The latter verifies the null hypothesis H_0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H_1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. In fact, for each problem instance, we compute the p-value of random search, MOPSO and mono-objective search results with NSGA-II ones. In this way, we could decide whether the superior performance of NSGA-II to one of each of the others (or the opposite) is statistically significant or just a random result.

5.3.3 Results and Discussion

To answer the first research question RQ1, an algorithm was implemented where merging solutions were randomly generated (random sequence) at each iteration. The obtained Pareto fronts were compared for statistically significant differences with NSGA-II using *IHV*, *IGD* and *IC*. We do not dwell long in answering the first research question (RQ1) that involves comparing our approach based on NSGA-II with random search. The remaining research questions will reveal more about the performance, insight, and usefulness of our approach.

Table 8 confirms that both multi-objective algorithms NSGA-II and MOPSO are better than random search based on the three quality indicators IHV, IGD and IC on all the seven open source systems. The Wilcoxon rank sum test showed that in 51 runs both NSGA-II and MOPSO results were significantly better than random search. The Wilcoxon rank sum test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference magnitude. The effect size could be computed by using the Cohen's d statistic [105]. The effect size is considered: (1) small if $0.2 \leq d < 0.5$, (2) medium if $0.5 \leq d < 0.8$, or (3) large if $d \geq 0.8$.

Table 8 shows the overview of the results of the significance tests comparison between NSGA-II and MOPSO. NSGA-II outperforms MOPSO in most of the cases: 17 out of 21 experiments (81%). MOPSO outperforms the NSGA-II approach only in GMF Gen, which is one of the smallest open source system considered in our experiments, having a low number of operations to merge. In particular, NSGA-II outperforms MOPSO in terms of IHV values in 6 out of 7 experiments with one ‘no significant difference’ result. Regarding IGD and IC, NSGA-II outperformed MOPSO in 5 out of 7 experiments, where only one case was not statistically significant, namely GMF Graph and one case where MOPSO provides better results namely GMF Gen.

Table 8 and Table 9, and Figure 24, Figure 25 confirm the superior performance of NSGA-II and MOPSO comparing to random search in terms of number of disabled refactoring, importance of enabled refactoring and automatic and manual correctness. In fact, random search is not efficient to generate good merging solutions using all the above metrics in all the experiments. Thus, an intelligent algorithm is required to find good trade-offs to propose efficient merging solutions. We conclude that there is empirical evidence that our multi-objective formulation surpasses the performance of random search thus our formulation is adequate and the use of metaheursitic search is justified (this answers RQ1).

Table 8 : The significantly best algorithm among random search, NSGA-II and MOPSO (Not sign. diff. means that NSGA-II and MOPSO are significantly better than random search, but not statistically different).

System	IHV	IGD	IC
GMF Map	NSGA-II	NSGA-II	NSGA-II
GMF Graph	NSGA-II	Not sign. diff.	Not sign. diff.
GMF Gen	Not sign. diff.	MOPSO	MOPSO
GanttProject	NSGA-II	NSGA-II	NSGA-II
Xerces-J	NSGA-II	NSGA-II	NSGA-II
JHotDraw	NSGA-II	NSGA-II	NSGA-II
ApacheAnt	NSGA-II	NSGA-II	NSGA-II

To answer RQ2, we evaluated the average of *NDR, IER, AC* and *MC* scores for non-dominated merging solutions proposed by NSGA-II. We evaluated the performance of NSGA-II to find good trade-offs between the two objectives of minimizing the number of disabled refactorings and maximizing the number of important enabled ones. Table 9 confirms that our NSGA-II adaptation was successful in generating merging sequences that minimize the number of disabled refactorings. The number of disabled refactorings seems reasonable if we consider the high number of refactorings (more than 250) to merge for ApacheAnt.

Table 9 : Median number of disabled refactorings on 51 independent runs. The results were statistically significant on 51 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

Systems	NDR-NSGAII	NDR-MOPSO	NDR-RS	NDR-AggGA	NDR-GA	NDR-Practical
GMF Map	3	4	7	6	3	8
GMF Graph	6	8	14	11	5	16
GMF Gen	16	18	43	23	16	49
GanttProject	21	23	31	34	19	33
Xerces-J	12	12	34	19	12	29
JHotDraw	11	14	29	18	11	37
ApacheAnt	26	29	68	34	23	71

Table 10 shows that NSGA-II provides merging solutions that not only minimize the number of disabled operations but also maximize the number of important enabled ones. This confirms that most of the disabled refactorings are not very important and it is interesting that the importance of the operations can help the algorithm which operation to disable when conflicts are detected. The highest importance score of enabled refactorings is 471.6 for ApacheAnt and the lowest importance score is 28.6 for GMF Map. An interesting observation that the highest and lowest number of disabled refactorings is also found in the same systems: ApacheAnt and GMF Map. Thus it is evident that NSGA-II finds the best trade-off between our two conflicting objectives. This can be explained by the fact that the algorithm tried to disable operations with the lowest importance when a conflict is detected to optimize simultaneously both objectives.

Table 10 : Median importance scores of enabled refactoring on 51 independent runs. The results were statistically significant on 51 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).

Systems	IER-NSGA-II	IER-MOPSO	IER-RS	IER-AggGA	IER-GA	IER-Practical
GMF Map	28.6	27.4	19.1	23.4	18.8	17.4
GMF Graph	39.4	41.1	22.5	29.8	19.6	18.1
GMF Gen	194.5	189.4	112.8	138.1	108.4	101.8
GanttProject	128.3	131.2	84.2	98.3	81.4	78.6
Xerces-J	139.1	134.4	88.1	96.1	86.3	81.3
JHotDraw	142.8	138.7	89.4	101.4	84.5	80.6
ApacheAnt	471.6	452.2	295.6	319.4	288.6	283.4

We considered two other metrics related to automatic and manual correctness. Figure 24 shows that the suggested merging solutions using NSGA-II are similar to the "reference" solution provided manually by developers with more 78% for all the 7 systems. However, it is a fastidious process for developers to manually merge parallel operations thus sometimes better solutions can be provided by our automated merging algorithm. A deviation with the set of reference solutions does not necessarily mean that there is an error/conflict with our multi-objective solutions but it could be another possible good merging solution different from the reference ones. To this end, we evaluated the suggested solutions by NSGA-II manually and we calculated a manual correctness score. Figure 25 confirms that most of the suggested merging solutions are good with an average of more than 85% in all the seven systems. This manual validation provides strong evidence that our merging solutions make sense semantically and could be applied to provide coherent model versions/releases. When comparing NSGA-II against MOPSO, we have found the following results: a) On small and medium scale software systems (Xerces-J, JHotDraw and ApacheAnt), NSGA-II is better that MOPSO on most systems with a medium effect size lower than 0.8; and b) on large scale software systems, NSGA-II is better than MOPSO on most systems with a small effect size lower than 0.5.

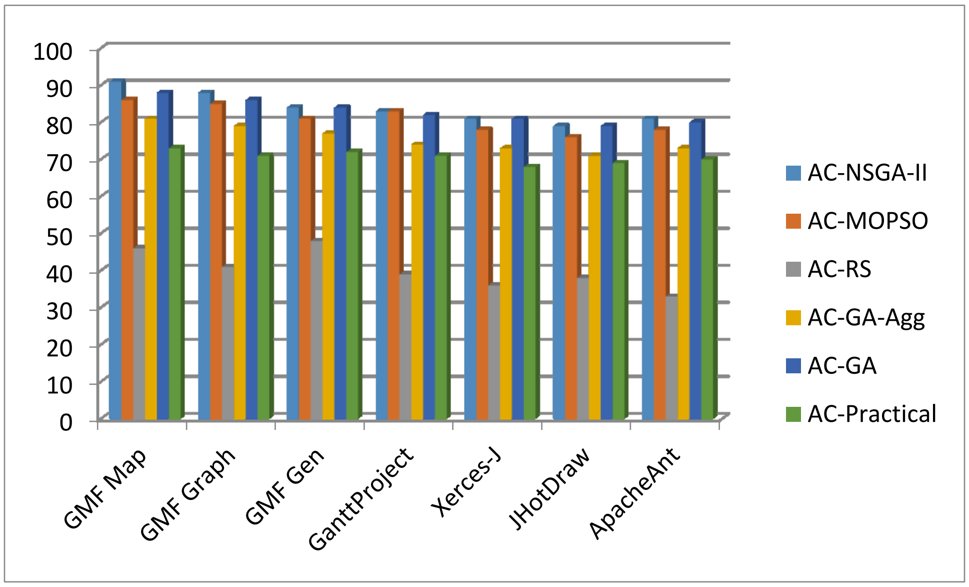


Figure 24 : Automatic correctness median values on 51 independent runs for the case studies

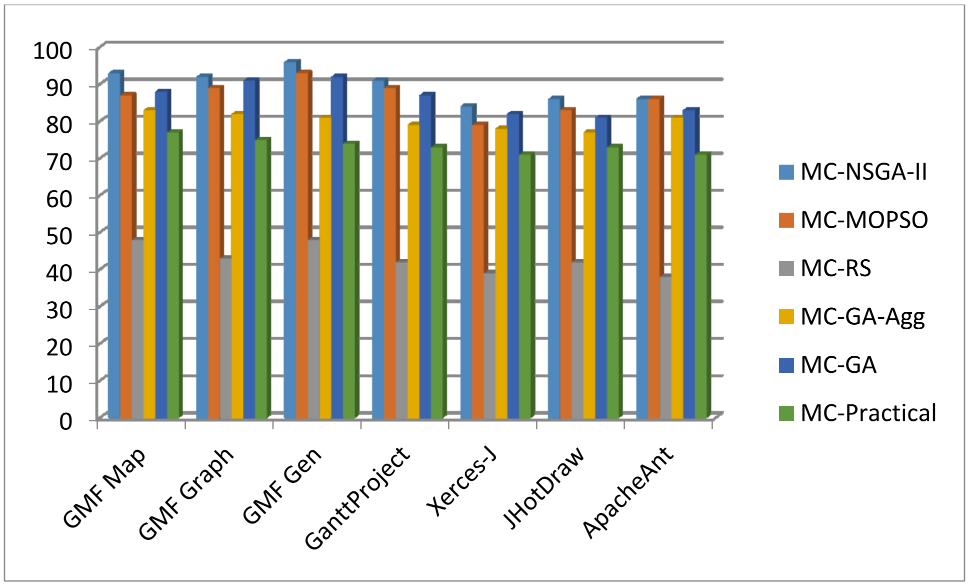


Figure 25 : Manual correctness median values on 51 independent runs for the case studies

To answer RQ3.1 we implemented a widely used multi-objective algorithm, namely multi-objective particle swarm optimization (MOPSO) and we compared NSGA-II and MOPSO using the same quality indicators used in RQ1. In addition, we used boxplots to analyze the distribution of the results and discover the knee point (best trade-off between the objectives).

To answer RQ3.2 we implemented a mono-objective Genetic Algorithm (Agg-GA) [106] where one fitness function is defined as an average of the two objectives. The multi-objective evaluation measures (*IHV*, *IGD* and *IC*) cannot be used in this comparison thus we considered the *NDR, IER, AC* and *MC* metrics. To answer RQ3.3 we compared NSGA-II with earlier mono-objective work for model merging [107] where the importance/priority of operations is not taken into account. We considered different metrics for the comparison such as *NDR, IER, AC* and *MC*. To answer RQ3.4 we used an existing model merging tool not based on optimization techniques where the operations are applied as they arrive without trying the find the best sequence/order of applying them. We compared the results of this tool with NSGA-II using *NDR, IER, AC* and *MC* metrics since only one solution can be proposed and not a set of “non-dominated” solutions.

A more qualitative evaluation is presented in Figure 26 illustrating the box plots obtained for the multi-objective metrics on the different projects. We see that for almost all problems the distributions of the metrics values for NSGA-II have smaller variability than for MOPSO. This fact confirms the effectiveness of NSGA-II over MOPSO in finding a well-converged and well-diversified set of Pareto-optimal merging solutions.

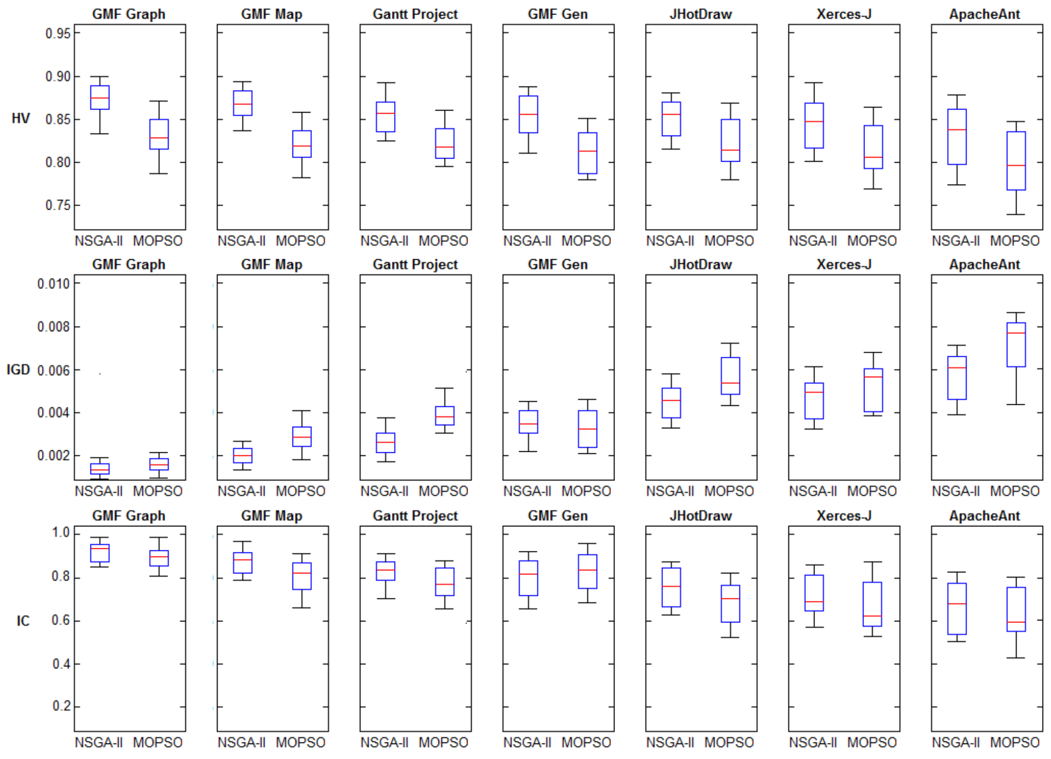


Figure 26 : Boxplots using the quality measures (a) IC, (b) IHV, and (c) IGD applied to NSGA-II and MOPSO.

Next, we use all five metrics NDR, IER, AC, MC and ICT to compare three robust refactoring algorithms: our NSGA-II adaptation, MOPSO, a mono-objective genetic algorithm (GA-Agg) that has a single fitness function aggregating the two objectives, mono-objective model merging and a manual merging tool [107]. In order to make meaningful comparisons, we select the best solution for NSGA-II and MOPSO using a knee point strategy.

For NDR, the number of disabled refactorings using NSGA-II is lower than MOPSO in all systems however the lowest number of disabled refactoring is provided by the mono-objective genetic algorithm approach where only one objective (maximizing the number of enabled refactorings) is considered. The number of disabled refactorings is the same between NSGA-II and GA in 50% of the cases thus NSGA-II can provide similar results to the mono-objective approach while taking into account the importance of enabled refactorings. The mono-objective GA approach that aggregates the two objectives in one provides highest number of disabled refactorings than NSGA-II, MOPSO and the non-search-based tool in 100% all the cases. This confirms that it is difficult to find trade-offs between conflicting objectives aggregated in only one fitness function. The random search and the non-search-based merging tool provides the highest number of disabled conflicts due the huge search space to explore to find the best sequence.

Table 9 describes the results of the comparison of our NSGA-II adaptation with the state-of-the-art merging approaches in terms of enabling the most of important refactorings for model merging (IER). NSGA-II suggests the best merging sequences that increase the number of enabled important refactorings for all systems except for Gantt. MOPSO outperforms NSGA-II in terms of IER only for Gantt and GMF Graph. The deviation between the performance of NSGA-II comparing to the performance of other approaches (except MOPSO) is high since all the mono-objective and manual approaches provides low importance score of the enabled refactoring on all the seven systems. This can be explained by the fact that multi-objective formulation selected the least important refactorings to disable when a conflict is detected. A more qualitative evaluation of NDR and IER is presented in Figure 27. It clearly shows the high deviation between the multi-objective and mono-objective formulation on 51 runs in terms of IRE and also it is important to note that the results of our multi-objective formulation is similar to the mono-objective ones in terms of NDR.

For AC and MC, Figure 25 and Figure 26 show that the solutions provided by NSGA-II have the highest manual and automatic correctness values. In fact, the average AC value for NSGA-II is 83% and it is lower than 80% for all the remaining algorithms for the seven systems. The same observation is valid for MC, NSGA-II has the highest MC average value with 86% while the remaining algorithms their MC average is lower than 82%. Figure 24 Figure 25 also reveal an interesting observation that there is no correlation between the number of operations to merge and the correctness values. More precisely, we sort AC and MC based on the number of refactorings for each open source system. From this data, we conclude that AC and MC are not necessarily affected negatively by a larger number of refactorings. For example, MC even increases from 82% to 88% when the number of refactorings increases from 86 to 112. Thus, we can conclude that our proposal shows a good scalability and is not affected negatively by the number of refactorings. However, when the number of operations increases, it does not necessarily mean that the number of disabled operations does.

Regarding IC, the execution time of NSGA-II is invariably lower than that of MOPSO with the same number of iterations (100,000), however the execution time required by Mono-EA (1h34 in average) is lower than both NSGA-II and MOPSO (1h58 in average for NSGA-II and 2h09 for MOPSO). It is well known that a mono-objective algorithm requires less execution time for convergence since only one objective is handled. However, the execution times of NSGA-II and MOPSO are not so far from those of mono-objective algorithms. For this reason, we can say that the proposed scheme has an accepted efficiency since it generates high quality solutions with a CPU time that is not significantly larger than the time needed by mono-objective algorithms.

In conclusion, we answer RQ3.1-3.4, the results support the claim that our NSGA-II formulation provides a good trade-off between both objectives, and outperforms on average the state-of-the-art of model merging approaches, both search-based and non-search-based ones.

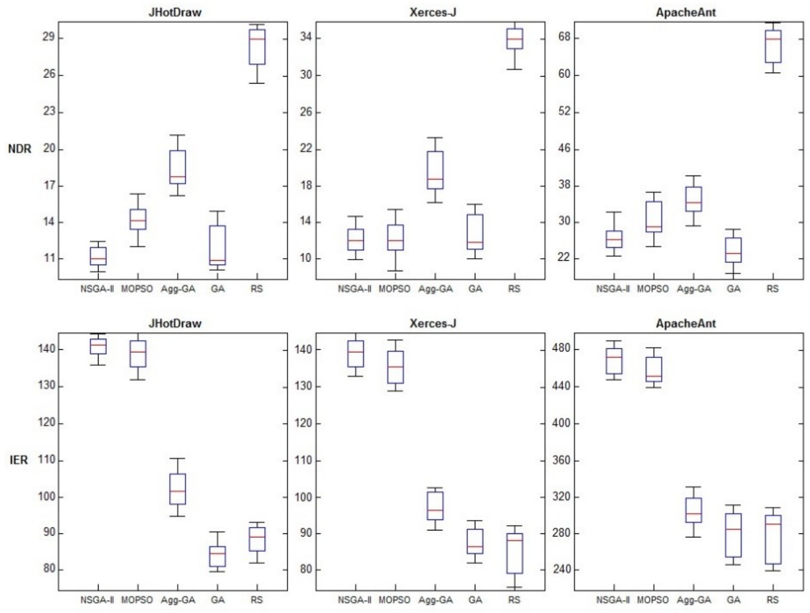


Figure 27: Boxplots using the measures *NDR* and *IER* applied to NSGA-II, MOPSO, Agg-GA, and Random Search (RS) on the different systems for 51 independent runs: GMF Graph, GMF Map, GMF Gen, Gantt, JHotDraw, Xerces-J and ApacheAnt.

To answer the last question (RQ4) we discussed on how the shape of the Pareto front can help developers to select the best merging solution based on their preferences.

Figure 28 depicts the different Pareto surfaces obtained on three open source systems (Apache Ant, JHotDraw and GMF Graph) using NSGA-II to optimize both objectives related to minimizing the number of disabled refactorings and maximizing the number of important enabled ones. Due to space limitations, we show only some examples of the Pareto-optimal front approximations obtained which differ significantly in terms of size. Similar observations were obtained on the remaining systems.

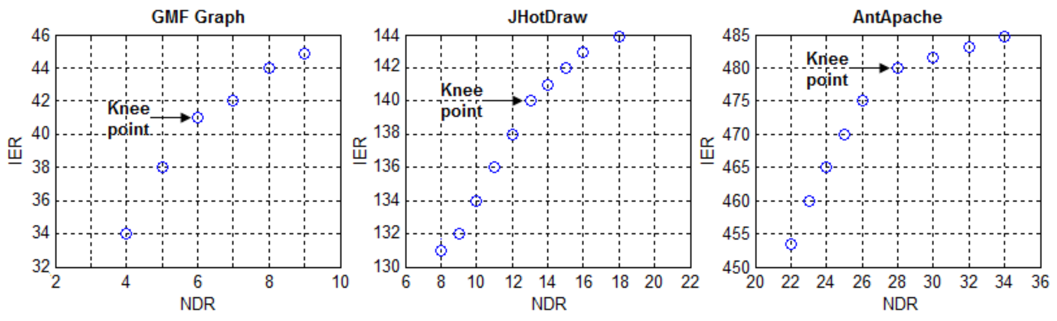


Figure 28: Pareto fronts for NSGA-II obtained on three open source systems:GMF Graph (small), JHotDraw (medium), and ApacheAnt (large).

One striking feature about all the three plots is that starting from the lowest number of disabled operations solution the trade-off between both objectives is not in favor of including important operations in the merging process, meaning that the number of enabled operations degrades slowly with a fast increase in the overall importance score of considered operations in the merging up to the knee point, marked in each figure. Thereafter, there is a sharp drop in the number of enabled operations with only a small increase in the importance score of considered operations in the merging solutions. It is very interesting to note that this property of the Pareto-optimal front is apparent in all the problems considered in this study. It is likely that a software engineer would be drawn to this knee point as the probable best trade-off between our both objectives. Without any consideration of the importance of operations in the search process, one would obtain the highest number of enabled operations in solution all the time, but Figure 28 shows how a better merging solution that include important operations can be obtained by sacrificing just a little in the number of total enabled refactorings.

We also compared the merging solution at the knee-point for JHotDraw with the best mering solution that maximizes only the number of enabled operations to understand why the former solution finds a good trade-off. We found that the knee-point solution disabled refactorings that were not important and not applied to important classes. Hence we conclude that RQ4 is affirmed and that the multi-objective approach has value for software engineers in a real-world setting.

The experiment results indicate clearly that the number of disabled operations is reduced significantly in comparison to the number of disabled operations without taking into consideration the different possible operation orders. Furthermore, the results provide strong evidence to support the claim that our proposal enables the generation of efficient model merging solutions to be comparable in terms of minimizing the number of conflicts to those suggested by existing approaches and to carry a high importance score of merged operations.

Although our approach has been evaluated with real-world models with a reasonable number of applied operations, we are working now on larger models and with larger lists of operations applied in parallel. This is necessary to investigate more deeply the applicability of the approach in practice, but also to study the performance of our approach when dealing with very large models.

Chapter 6: Defects Detection at the Model Level

6.1 Introduction

Code-smells classify shortcomings in software that can decrease software maintainability. They are also defined as structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and trigger refactoring of code [108] [109]. Code-smells are not limited to design flaws since most of them occur in code and are not related to the original design. In fact, most of code-smells represent patterns or aspects of software design that may cause problems in the further development and maintenance of the system [110]. As stated by Fowler in [88], code-smells are unlikely to cause failures directly, but may do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. Different types of code-smells, presenting a variety of symptoms, have been studied in the intent of facilitating their detection and suggesting improvement solutions.

In our approach, we focus on the following *five* code-smell types:

Blob: It is found in designs where one large class monopolizes the behavior of a system (or part of it), and the other classes primarily encapsulate data.

Feature Envy (FE): It occurs when a method is more interested in the features of other classes than its own. In general, it is a method that invokes several times accessor methods of another class.

Data Class (DC): It is a class with all data and no behavior. It is a class that passively store data.

Spaghetti Code (SC): It is a code with a complex and tangled control structure.

Functional Decomposition (FD): It occurs when a class is designed with the intent of performing a single function. This is found in a code produced by non-experienced object-oriented developers.

We choose these code-smell types in our experiments because they are the most frequent to detect and fix based on recent empirical studies [111] [112] [113] [114]. Of course, the proposed approach in this paper can be extended to other types of code smell.

Software metrics can be used to capture the structural and semantic attributes of the software, and can be a reliable indicator of the quality of design. These quality indicators can then be used to quantitatively estimate and reflect the design signatures of software architecture in terms of many metrics including coupling, cohesion, cyclic complexity, etc. The code smell detection process usually involves finding the fragments of code which violate these software metrics. Several studies [48] [53] [51] classified these software metrics for object oriented architectures. We selected and used in our experiments the software metrics described in Table 11.

Table 11: List of Software Metrics.

Metrics	Description
Weighted Methods per Class (WMC)	WMC represents the sum of the complexities of its methods.
Response for a Class (RFC)	RFC is the number of different methods that can be executed when an object of that class receives a message.
Lack of Cohesion of Methods (LCOM)	Chidamber and Kemerer define Lack of Cohesion in Methods as the number of pairs of methods in a class that does not have at least one field in common minus the number of pairs of methods in the class that does share at least one field. When this value is negative, the metric value is set to 0.
Number of Attributes (NA)	Number of attributes in aclass.
Attribute Hiding Factor (AH)	AH measures the invisibilities of attributes in classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is not visible.

Method Hiding Factor (MH)	MH measures the invisibilities of methods in classes. The invisibility of a method is the percentage of the total classes from which the method is not visible.
Number of Lines of Code (NLC)	NLC counts the lines but excludes empty lines and comments.
Coupling Between Object classes (CBO)	CBO measures the number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.
Number of Classes (NC)	Number of classes in the considered package.
Depth of Inheritance Tree (DIT)	DIT is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. In cases involving multiple inheritances, the DIT is the maximum length from the node to the root of the tree.
Polymorphism Factor (PF)	PF measures the degree of method overriding in the class inheritance tree. It equals the number of actual method overrides divided by the maximum number of possible method overrides.
Attribute Inheritance Factor (AIF)	AIF is the fraction of class attributes that are inherited.
Number of Children (NOC)	NOC measures the number of immediate descendants of the class.

The manual definition of rules to identify can be time-consuming for some types of code smell such as the Functional Decomposition defect. One of the main issues is related to the definition of thresholds when dealing with quantitative information. For example, the 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. Thus, the manual definition of detection rules sometimes requires a high calibration effort. Furthermore, the manual selection of the best combination of metrics that formalize some symptoms of code-smells is challenging. In addition, the translation of code-smell definitions into metrics is not straightforward. Some definitions of code-smells are confusing and it is difficult to determine which metrics to use to identify such design problems. To address these challenges, we describe in the next section our approach based on the use of multi-objective genetic programming to generated code-smells detection rules using not only bad design practice examples but also good ones.

6.2 Approach Overview

6.2.1 Detection Scheme

We propose in this work to consider the problem of code-smells detection as a multi-objective problem as shown in Figure 29 where examples of code-smells and well-designed models are used to generate detection rules. To this end, we use multi-objective genetic programming (MOGP) to find the best combination of metrics that maximizes the detection of design defect examples and minimizes the detection of well-designed relevant software model.

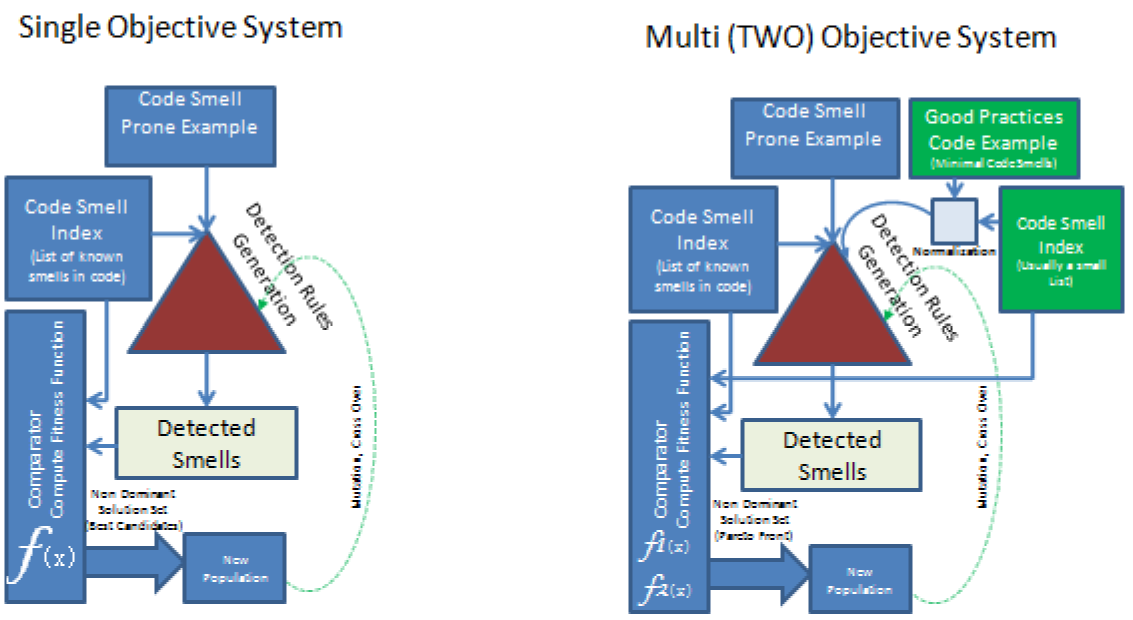

Single Objective System
Code Smell Prone Example
Code Smell Index (List of known smells in code)
Detection Rules Generation
Comparator
Compute Fitness Function
$f(x)$
Detected Smells
Non Dominant Solution Set (Best Candidates)
New Population
Mutation, Cross Over
Multi (TWO) Objective System
Code Smell Prone Example
Good Practices Code Example (Minimal CodeSmells)
Code Smell Index (List of known smells in code)
Normalization
Code Smell Index (Usually a small List)
Detection Rules Generation
Comparator
Compute Fitness Function
$f_1(x)$
$f_2(x)$
Detected Smells
Non Dominant Solution Set (Pareto Front)
New Population
Mutation, Cross Over

Figure 29: Single Objective system compared with multi objective system. The multi objective system uses Good-Example to improve the quality of detection rules and help minimize the false positives.

The code-smells detection problem involves searching for the best metric combinations among the set of candidate ones, which constitutes a huge search space. A solution of our code-smells detection problem is a set of rules (metric combination with their thresholds values) where the goal of applying these rules is to detect code smells in a system. We propose a multi-objective formulation of the code-smells rules generation problem. Consequently, we have two objective functions to be optimized: (1) maximizing the coverage of code-smell examples, and (2) minimizing the detection of good design-practice examples as false positives. The collected examples of well-design code and code-smells on different systems are taken as an input for our approach.

6.2.2 Solution Coding

Analytically speaking, the formulation of the multi-objective problem can be stated as follows:

$$\begin{cases} \max f_1(x) = \dfrac{\dfrac{|DCS(x)| \cap |ECS|}{|ECS|} + \dfrac{|DCS(x)| \cap |ECS|}{|DCS(x)|}}{2} \\ \min f_2(x) = \dfrac{\dfrac{|DCS(x)| \cap |EGE|}{EGE} + \dfrac{|DCS(x)| \cap |EGE|}{|DCS(x)|}}{2} \end{cases}$$

where $|DCS(x)|$ is the cardinality of the set of Detected Code-Smells by the metric combination x, $|ECS|$ is the cardinality of the set of Existing Code-Smells, and |GDE| is the cardinality of the set of Existing Good Examples.

Once the bi-objective trade-off front is obtained, the developer can navigate through this front in order to select his/her preferred solution (metric combination).

As noted by Harman et al. [84], a multi-objective algorithm cannot be used “out of the box” – it is necessary to define problem-specific genetic operators to obtain the best performance.

To adapt MOGP to our problem, the required steps are to create: (1) solution representation, (2) solution variation, and (3) solution evaluation.

6.2.2.1 Solution Representation

In this MOGP based detection approach, a solution representation is composed of terminals and functions. Therefore, when applying MOGP to solve a specific problem, they should be carefully selected and designed to satisfy the requirements of the current problem. After evaluating many parameters related to the code-smells detection problem, the terminal set and the function set are decided as follows. The terminals correspond to different quality metrics with their threshold values. The functions that can be used between these metrics are Union (*OR*) and Intersection (*AND*). More formally, each candidate solution x in this problem is a set of detection rules where each rule is represented by a binary tree such that:

(1) each leaf-node (Terminal) L belongs to the set of metrics (such as number of methods, number of attributes, etc.) discussed in Section 2 and their corresponding thresholds generated randomly.

(2) each internal-node (Functions) N belongs to the Connective (logic operators) set $C = \{AND, OR\}$.

The set of candidates solutions (rules) corresponds to a logic program that is represented as a forest of *AND*-*OR* trees. Each sub-tree corresponds to a rule for the detection of specific code-smell (e.g. blob, functional decomposition, etc.). Figure 30 illustrates an example of a solution according to our formulation including two rules. It is an example of rules generated randomly by the genetic programming and not the best set of rules found at the end of the execution of the algorithm. The first rule is to detect Blob using the two metrics NLC and NA and the second rule detects Spaghetti Code (SC) using one metric NOC. The thresholds values are selected randomly along with the comparison and logic operators.

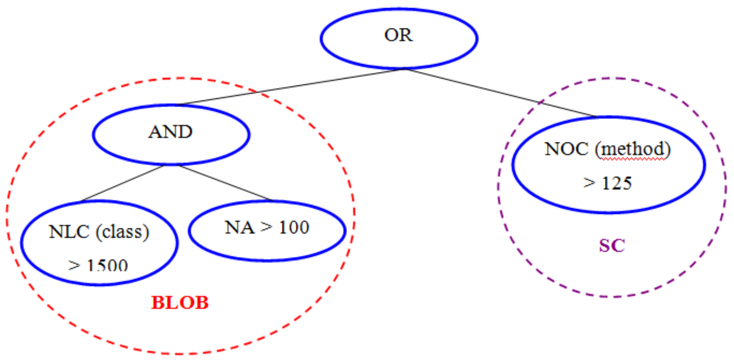


Figure 30: An example of a detection rule according to our formulation.

6.2.2.2 Solution Variation

Solution variation in this detection scheme utilizes a mutation operator that can be applied to a function node or a terminal node. It starts by randomly selecting a node in the tree. Then, if the selected node is a terminal (quality metric), it is replaced by another terminal (metric or another threshold value). If it is a function (*AND-OR*), it is replaced by a new function. If a tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree. Figure 31 illustrates an example of a mutation operation. One node is deleted from the tree representation to generate a new other possible solution. For the crossover, two parent individuals are selected and a sub-tree is picked on each one. Then crossover swaps the nodes and their relative sub-trees from one parent to the other. This operator must ensure the respect of the depth limits. The crossover operator can be applied with only parents having the same rules category (code-smell type to detect). Each child thus obtains information from both parents. Figure 32 illustrates a set of detection rules before crossover and Figure 33 shows the rules after the application of cross over operation. Two sub-trees are exchanges between two solutions to generate two new ones.

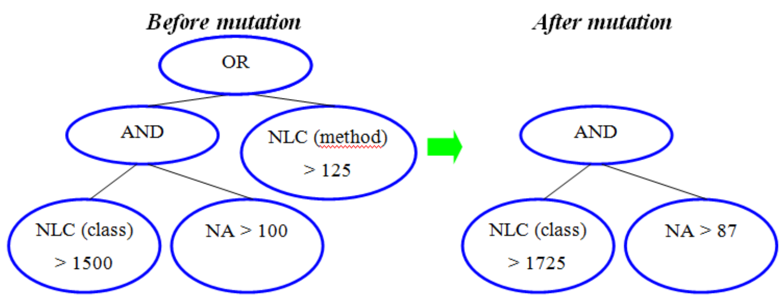


Figure 31: An exemplified mutation operation.

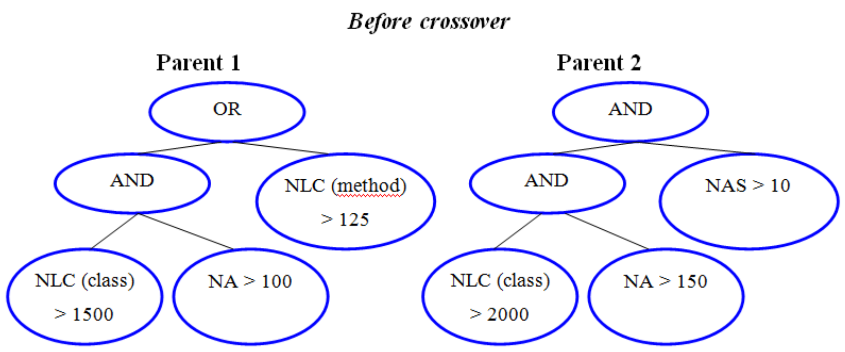


Figure 32: Before Crossover operation is performed

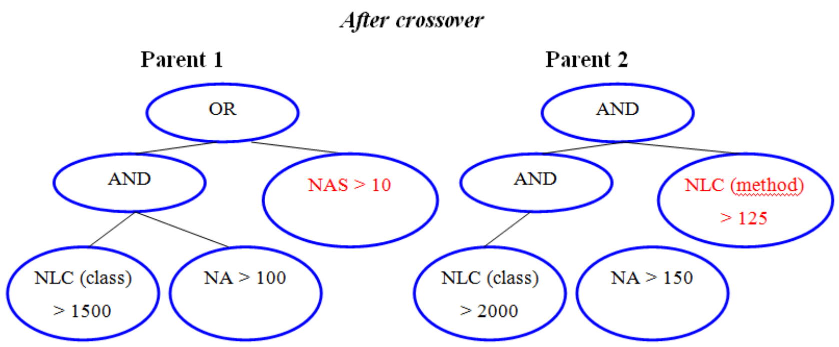


Figure 33: After Crossover operations is performed

6.2.2.3 Solution Evaluation

The solution is evaluated based on the two objective functions: maximize detection of anti patterns while minimizing the detection of false positives in reference design. Since we are considering a bi-objective formulation, we use the concept of Pareto optimality to find a set of compromise (Pareto-optimal) solutions. The fitness of a particular solution in MOGP corresponds to a couple (*Pareto Rank*, *Crowding distance*). In fact, MOGP classifies the population individuals (of parents and children) into different layers, called non-dominated fronts. Non-dominated solutions are assigned a rank of 1 and then are discarded temporary from the population. Non-dominated solutions from the truncated population are assigned a rank of 2 and then are discarded temporarily. This process is repeated until the entire population is classified with the domination metric. After that, a diversity measure, called *crowding distance*, is assigned front-wise to each individual. The crowding distance is the average side length of the cuboid formed by the nearest neighbors of the considered solution. Once each solution is assigned its Pareto rank, mating selection and environmental selection are performed. This is based on the crowded comparison operator ($\prec n$) that favors solutions having better Pareto ranks and, in case of equal ranks, it favors the solution having larger crowding distance. In this way, convergence towards the Pareto optimal bi-objective front and diversity along this front are emphasized simultaneously. The output of MOGP is the last obtained parent population containing the best of the non-dominated solutions found. When plotted in the objective space, they form the Pareto front from which the user will select his/her preferred code-smells detection rules solution.

6.3 Validation

6.3.1 Research Questions

We defined five research questions that address the applicability, performance in comparison to existing refactoring approaches, and the usefulness of our multi-objective code-smells detection approach. The five research questions are as follows:

RQ1: How does MOGP perform against Random Search (this serves as sanity check)? If random search outperforms an intelligent search method, then we can conclude that our problem formulation is not adequate.

RQ2: How does MOGP perform against start of the art code-smell detectors?

RQ 2.1: How does MOGP perform compared to MOAIS? It is important to justify the use of MOGP for the problem of multi-objective code-smells detection. It is almost impossible to formally justify the use of a specific metaheuristic search against another. The only proof in general is the experiments' result. To this end, we compared MOGP with another multi-objective algorithm, MOAIS [115] using the same adaptations. MOAIS is a widely used multi objective non dominated neighbour based selection algorithm which is used as a benchmark for evaluation of multi object search algorithms.

RQ 2.2: How does MOGP perform compared to mono-objective genetic programming (GP) with aggregation of both objectives? This research question is essential to validate the choice of using multi objective algorithm as opposed to using mono objective algorithm.

RQ 2.3: How does MOGP perform compared to some existing code-smells detection approaches? It is important to determine if the proposed detection approach which employs good and bad design examples performs better than existing approaches including non-search techniques.

RQ3: To what extent can our approach generate rules that detect different code-smell types? It is important to discuss the ability of our approach to detect different types of code-smells to evaluate the quality of each detection rule separately.

6.3.2 Experimental Setup

Our study considers the extensive evolution of different open source Java systems analyzed in the literature [116] [117]. As described in Table 2, the corpus used includes releases of Apache Ant, ArgoUML, Gantt, Azureus and Xerces-J. Apache Ant is a build tool and library specifically conceived for Java applications. ArgoUML is an open-source UML modeling tool. Xerces is a family of software packages that implement a number of standard APIs for XML parsing. GanttProject is a tool for creating project schedules in the form of Gantt charts and resource-load charts. Azureus is a peer-to-peer file-sharing tool. Table 12 reports the size in terms of classes of the analyzed systems. The table also reports the number of code smells identified manually in the different systems by a team of software engineers and researchers, more than 800 in total for the five types of code smell considered in our experiments. For replication of experimental results, the team provided a corpus describing instances of different code smells including blob, spaghetti code, and functional decomposition. Subjects included 8 master students in Software Engineering, 5 Ph.D. students in Software Engineering and 2 faculty members in Software Engineering. All the 15 subjects were familiar with Java development, software maintenance activities including code smells detection and refactoring. The experience of these subjects on Java programming ranged from 1 to 17 years. In our study, we verified the capacity of our approach to detect classes that correspond to instances of these code smells. We choose the above-mentioned open source systems because they were analyzed in related work and for comparison purposes. JHotDraw was chosen as an example of reference code (training examples for our MOGP of good design practices) because it contains very few known code-smells.

Table 12: Software projects features.

Systems	**Number of classes**	**Number of code smells**				
		Blob	FD	SC	DC	FE
ArgoUML v0.26	1358	22	52	64	21	14
ArgoUML v0.3	1409	10	58	61	16	22
Xerces v2.7	991	14	32	36	19	24
Ant-Apache v1.5	1024	22	47	34	23	21
Ant-Apache v1.7.0	1839	25	51	48	18	26
Gantt v1.10.2	245	4	21	16	14	11
Azureus v2.3.0.6	1449	19	44	52	29	34

We use the two following performance indicators (which are among the most used in multi-objective optimization) when comparing MOGP and MOAIS:

Hypervolume (*IHV*) [118]: It corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. Larger values for this metric mean better performance.

Inverted Generational Distance (*IGD*) [118]: It is a convergence measure that corresponds to the average Euclidean distance between the Pareto front Approximation *PA* provided by the algorithm and the Reference Front *RF* (RF is the set of non-dominated solutions obtained over all runs). The distance between *PA* and *RF* in an *M*-objective space is calculated as the average *M*-dimensional Euclidean distance between each solution in *PA* and its nearest neighbour in *RF*. Lower values for this indicator mean better performance (convergence).

To assess the accuracy of our approach, we compute two measures: (1) precision and (2) recall, originally stemming from the area of information retrieval:

Precision (PR): It denotes the fraction of correctly detected code-smells among the set of all detected code-smells. It could be seen as the probability that a detected code-smell is correct.

$$precision = \frac{\{(releant\ code\ smells) \cap (detected\ code\ smells)\}}{(detected\ code\ smells)}$$

Recall (RE): It corresponds to the fraction of correctly detected code-smells among the set of all manually identified code-smells (i.e., how many code-smells have not been missed). It could be seen as the probability that an expected code-smell is detected.

$$recall = \frac{\{(releant\ code\ smells) \cap (detected\ code\ smells)\}}{(relevant\ code\ smells)}$$

Since metaheuristic algorithms are stochastic optimizers, they usually provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$). This statistical test verifies the null hypothesis H_0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H_1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than α (> 0.05) means the opposite.

Parameter setting has a significant influence on the performance of a search algorithm on a particular problem instance. For this reason, for each multi-objective algorithm and for each system (cf. Table 3), we perform a set of experiments using several population sizes: 50, 100, 200, 500 and 1000. The stopping criterion was set to 250,000 fitness evaluations for all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.20 where the probability of gene modification is 0.3; stopping criterion = 250,000 fitness evaluations. For MOAIS, the maximum size of the dominant population, and the clone population size are set to 100 and 20 respectively.

6.3.3 Result and Discussion

We do not dwell long in answering the first research question (RQ1) that involves comparing our approach based on NSGA-II with random search. The remaining research questions will reveal more about the performance, insight, and usefulness of our approach. Table 13 confirms that MOGP and MOAIS are better than random search based on the two quality indicators IHV and IGD on all seven open source systems. The Wilcoxon rank sum test showed that in 51 runs both MOGP and MOAIS results were significantly better than random search. We conclude that there is empirical evidence that our multi-objective formulation surpasses the performance of random search thus our formulation is adequate (this answers RQ1).

Table 13: The significantly best algorithm among random search, MOGP and MOAIS (No stat. diff. means that MOGP and MOAIS are significantly better than random, but not statistically different).

Project	IHV	IGD
ArgoUML v0.26	MOGP	MOGP
ArgoUML v0.3	MOGP	MOGP
Xerces v2.7	No stat. diff.	MOAIS
Ant-Apache v1.5	MOGP	MOGP
Ant-Apache v1.7.0	MOGP	MOGP
Gantt v1.10.2	MOGP	No stat. diff.
Azureus v2.3.0.6	MOGP	MOGP

In section, we compare our MOGP adaptation to the current, state-of-the-art code-smells detection approaches. To answer the second research question, RQ2.1, we compared MOGP to another multi-objective algorithm, MOAIS, using the same adaptations. Table 14 shows the overview of the results of the significance tests comparison between MOGP and MOAIS. MOGP outperforms MOAIS in most of the cases: 11 out of 14 experiments (78%).

A more qualitative evaluation is presented in Figure 34Figure 35 illustrating the box plots obtained for the multi-objective metrics on the different projects. We see that for almost all problems the distributions of the metrics values for MOGP have smaller variability than for MOAIS. This fact confirms the effectiveness of MOGP over MOAIS in finding a well-converged and well-diversified set of Pareto-optimal detection rules solutions (RQ2.1).

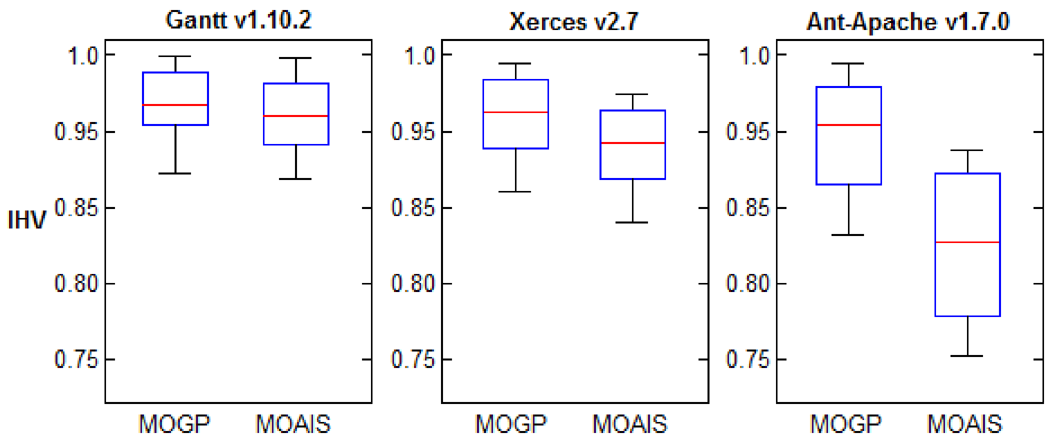


Figure 34: IHV boxplots on 3 projects having different sizes (Gantt v1.10.2: small, Xerces v2.7: medium, Ant-Apache v1.7.0: large).

Next, we use precision and recall measures to compare the efficiency of our MOGP approach compared to mono-objective GP (aggregating both objectives) and two existing code-smells detection studies [116] [117]. We first note that the mono-objective approaches provide only one detection solution (set of detection rules), while MOGP generate a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for MOGP using a *knee point* strategy as. The knee point corresponds to the solution with the maximal trade-off between maximizing the coverage of code-smells and minimizing the number of detected well-designed code. Thus, for MOGP, we select the knee point from the Pareto approximation having the median IHV value. We aim by this strategy to ensure fairness when making comparisons against the mono-objective EA. For the latter, we use the best solution corresponding to the median observation on 51 runs.

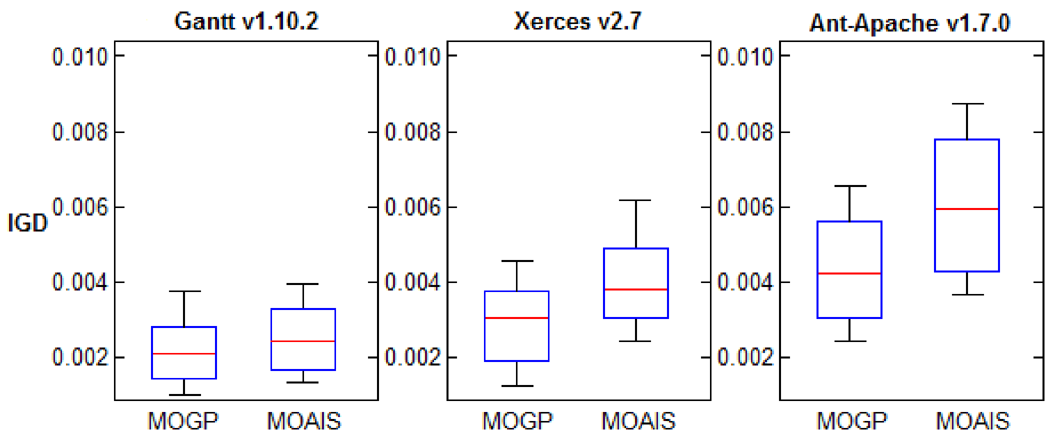


Figure 35: IGD boxplots on 3 projects having different sizes (Gantt v1.10.2: small, Xerces v2.7: medium, Ant-Apache v1.7.0: large).

The results from 51 runs are depicted in Table 14. It can be seen that MOGP provides better precision and recall scores for the detection of code-smells. For recall (RE), MOGP is better than GP in 100% of the cases. We have the same observation for the precision also where MOGP outperforms GP in all cases with an average of more than 90%. Thus, it is clear that a multi-objective formulation of our problem outperforms the aggregation-based approach. In conclusion, we answer RQ2.2 by concluding that the results obtained in Table 14 confirm that both multi-objective formulations are adequate and outperform the mono-objective algorithm based on an aggregation of two objectives related to use of good and bad software design examples. Table 14 also shows the results of comparing our multi-objective approach based on MOGP with two mono-objective refactoring approaches [116] [117]. In [116], the authors used search-based techniques to detect code-smells from only code-smell examples. In [117], an artificial immune system approach is proposed to detect code-smells by deviation with well-designed code examples. It is apparent from Table 14 that MOGP outperforms both mono-objective approaches in terms of precision and recall in most of the cases. This is can be explained by the fact that our proposal takes into account both positive and negative examples when generating the detection rules. If only code-smell examples are used then it is difficult to ensure the coverage of all possible bad design behaviors. The same observation is still valid for the use only of well-designed code examples. The use of both types of examples represents a complementary way to formulate the problem of code-smells detection using a multi-objective approach. To answer RQ2.3, the results of Table 14 support the claim that our MOGP formulation outperforms, on average, the two code-smells existing approaches.

Table 14: Recall and precision median values of MOGP, MOAIS, GP, [28] and [29] over 51 independent simulation runs.

System	RE-MOGP	RE-MOAIS	RE-GP	RE-[28]	RE-[29]	PR-MOGP	PR-MOAIS	PR-GP	PR-[28]	PR-[29]
ArgoUML v0.26	92% (158/173)	90% (155/173)	85% (147/173)	81% (141/173)	83% (143/173)	87% (158/176)	85% (155/178)	76% (147/188)	74% (141/194)	72% (141/198)
ArgoUML v0.3	90% (149/167)	88% (146/167)	81% (136/167)	74% (123/167)	76% (126/167)	86% (149/174)	86% (146/174)	73% (136/179)	64% (123/194)	63% (123/196)
Xerces v2.7	90% (113/125)	90% (113/125)	83% (103/125)	75% (93/125)	76% (95/125)	87% (113/131)	85% (113/136)	71% (103/143)	69% (93/132)	67% (93/136)
Ant-Apache v1.5	89% (131/147)	87% (127/147)	79% (116/147)	85% (124/147)	82% (120/147)	90% (131/145)	90% (127/145)	76% (116/152)	75% (124/160)	72% (124/167)
Ant-Apache v1.7.0	93% (157/168)	93% (157/168)	77% (129/168)	71% (119/168)	69% (115/168)	95% (157/163)	92% (157/169)	72% (129/176)	67% (119/176)	65% (119/181)
Gantt v1.10.2	90% (57/66)	86% (55/66)	83% (56/66)	77% (51/66)	72% (48/66)	79% (57/73)	76% (55/76)	63% (56/88)	58% (51/89)	63% (51/82)
Azureus v2.3.0.6	94% (169/178)	92% (163/178)	79% (140/178)	69% (122/178)	72% (128/178)	86% (169/187)	86% (163/187)	76% (140/184)	67% (122/188)	69% (122/182)

We noticed that our technique does not have a bias towards the detection of specific code-smells types. In all systems, as shown in Figure 36, we had an almost equal distribution of each code-smell types (SCs, Blobs, FEs, DCs and FDs). Overall, all the five code smell types are detected with good precision and recall scores in the different systems (more than 85%). This ability to identify different types of code-smells underlines a key strength to our approach. Most other existing tools and techniques rely heavily on the notion of size to detect code-smells. This is reasonable considering that some code-smells like the Blob are associated with a notion of size. For code-smells like FDs, however, the notion of size is less important and this makes this type of anomaly hard to detect using structural information. This difficulty limits the performance of GP in well detecting this type of code-smells. Thus, we can conclude that our MOGP approach detects well all the five types of considered code-smells (RQ3).

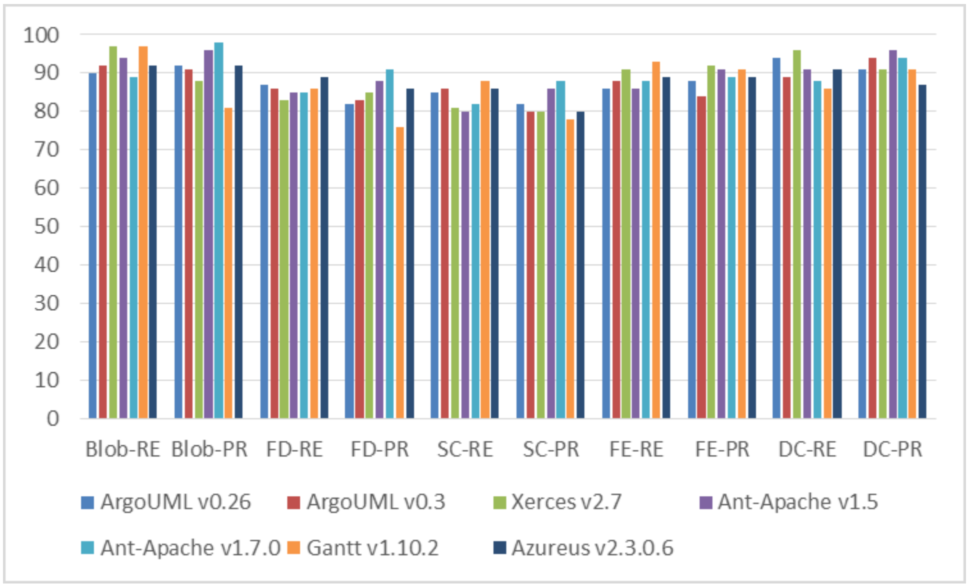


Figure 36: The median values of precision and recall on 51 runs for the five types of code-smell.

Chapter 7: Refactoring Recommendation at the Model Level

7.1 Introduction

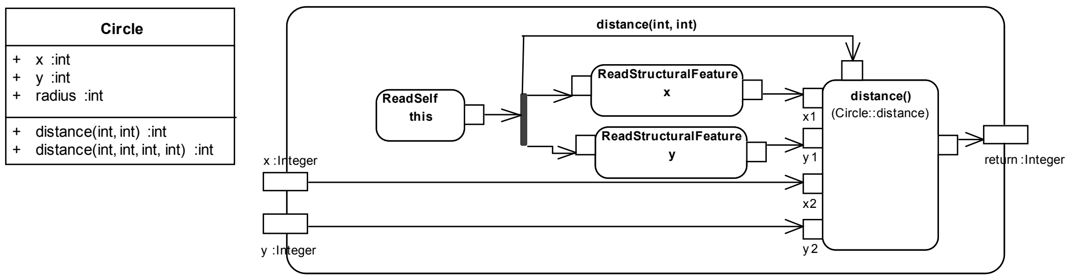
One of the most widely used techniques for fixing design defects and improving structural integrity of evolving software systems is refactoring, which improves design structure while preserving external behavior [5]. Various tools [5] [119] [120] [121] [122] supporting refactoring have been proposed in the literature. The vast majority of these tools provide different environments to apply manually or automatically refactorings to avoid and fix bad-design practices [123].

The majority of existing refactoring works focus mainly on the source code level. The suggestion of refactorings at the model level is more challenging due to the difficulty to evaluate: a) the impact of the suggested refactorings applied to a diagram on other related diagrams to improve the overall system quality, b) their feasibility, and c) inter-diagram consistency. In the source code level, traditional code quality metrics are used to evaluate the quality of a system after applying a sequence of refactorings. However, applying refactoring on a specific model such as class diagrams has an impact on related other diagrams such as activity diagrams, sequence diagrams, etc. Sometimes, an improvement of class diagram quality metrics may decrease the quality of an activity diagram. Thus, it is important to evaluate the impact of suggested refactorings not only on one diagram, but also other related diagrams to estimate the overall quality. Second, some refactorings suggested at the model level cannot be applied to the source code level. Third, it is difficult to check if a refactoring applied to a class diagram preserves the behavior or not without the use of some related behavioral diagrams such as an activity diagram.

We propose, in this chapter, a novel framework that enables software designers to apply refactoring at the model level. To this end, we used a multi-objective evolutionary algorithm to find a trade-off between improving the quality of different diagrams at the same time such as class diagrams and activity diagrams. The proposed multi-objective approach provides a multi-view for software designers to evaluate the impact of suggested refactorings applied to class diagrams or related activity diagrams in order to evaluate the overall quality, and check their feasibility and behavior preservation. The statistical evaluation performed on models extracted from four open source systems confirms the efficiency of our approach.

7.1.1 Multi View Refactoring : A Motivating Example

To showcase the challenges of finding an optimal sequence of refactorings to improve the quality of the class diagram and at the same time the quality of the activity diagram, consider the example illustrated in Figure 37 (initial version before refactoring) and Figure 38 (intermediate version after refactoring of class diagram), and Figure 39 (final version after refactorings of class and activity diagrams).



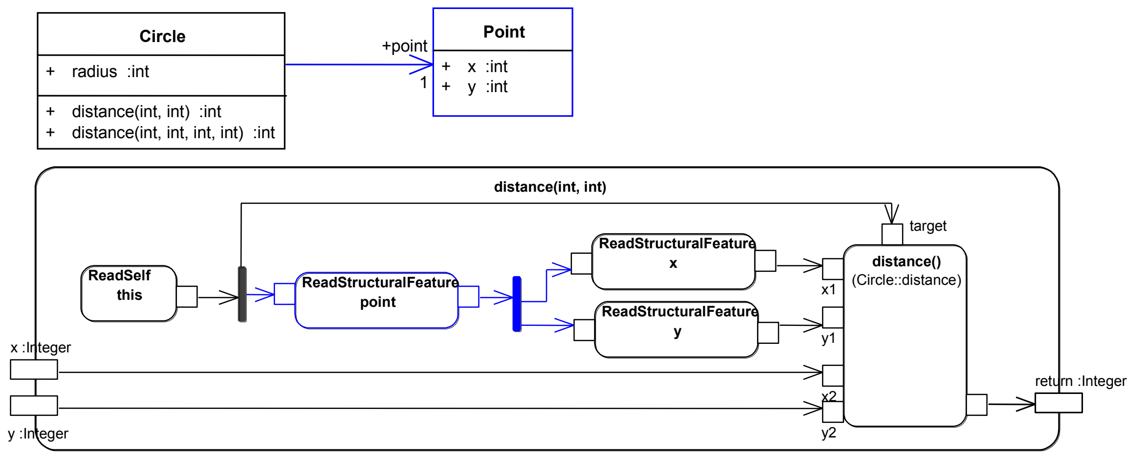
Classes	
# classes	= 1
PPC	= 3
OPC	= 2

Activity	
# edges	= 9
# nodes	= 5
# actions	= 4
CFC	= 3
LOC	= 1

Legend
PPC … # properties per class
OPC … # operations per class,
CFC … control-flow complexity = #forks + # decisions
LOC … locality = # reads in referenced class / all reads

Figure 37 : Initial Version for Motivating Example – Initial Version before Refactoring

The class *Circle* contains two properties *x* and *y*, which specify the coordinates of its center point, we may apply the refactoring “Extract Class” to encapsulate these two parameters. Of course, we also have to co-refactor the activity diagram accordingly. The class and activity diagram after the refactoring and the co-refactoring is depicted in Figure 38. Thus, a new class *Point* has been introduced, which now contains the two properties *x* and *y*. Besides, a new association is created to link the point from the class *Circle.* Alongside the class diagram, we had to apply co-refactorings in the operation *distance(int, int).* In particular, a new *ReadStructuralFeatureValueAction* ("point") has been added to obtain the values *x* and *y*. We may observe that, although the quality of the class diagram might have been improved (e.g., there is a better distribution of properties per class), the length, the number of edges, the control-flow complexity (CFC), as well as the locality (LOC) indicate a worse design with respect to the activity diagram. The reason for this is that the activity specifying the behavior of the operation *distance(int, int)* contains an additional read-action to obtain values from a referenced object.



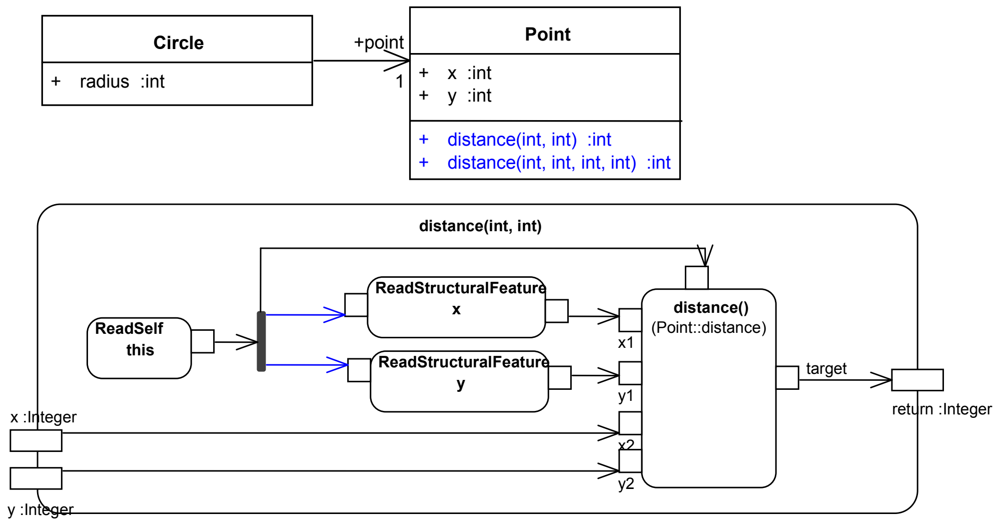
Classes	
# classes	= 2
PPC	= 1,5
OPC	= 1

Activity	
# edges	= 11
# nodes	= 7
# actions	= 4
CFC	= 4
LOC	= 0,4

Legend
PPC … # properties per class
OPC … # operations per class,
CFC … control-flow complexity = #forks + # decisions
LOC … locality = # reads in referenced class / all reads

Figure 38 : Intermediate Version for Motivating Example for Model Refactoring – After Extract Class Point Refactoring

To improve this situation, we have to apply another refactoring, namely “Move Operation”, in order to move the operations distance(int, int) into the newly created class *Point* as shown in Figure 39. Then, however, we break the conformance rules of class diagram and the activity diagram, because in *distance(int, int)*, the operation *distance(int, int, int, int)* is called, which is not possible in the scope of *Point*, since *Point* has no access to the instance of *Circle.* Nevertheless, when we accept the temporary inconsistency and also move the operation *distance(int, int, int, int)* into the class *Point*, we obtain a new result, depicted in Figure 3, which not only validates all conformance rules, but also improves the metrics of the activity diagram significantly; the number of edges has been reduced and the control-flow complexity, as well as the locality, has been improved.



Classes	
# classes	= 2
PPC	= 1,5
OPC	= 1

Activity	
# edges	= 9
# nodes	= 5
# actions	= 4
CFC	= 3
LOC	= 1

Legend
PPC … # properties per class
OPC … # operations per class,
CFC … control-flow complexity = #forks + # decisions
LOC … locality = # reads in referenced class / all reads

Figure 39 : Final Version for Motivating Example for Model Refactoring – After Move Methods *distance* Operation

In conclusion, applying refactorings on the class diagram may have a strong impact on the quality of the activity diagrams that specify the behavior of the classes' operations. Even worse, in several scenarios, the class refactorings will break their consistency. Finding a good sequence of refactorings to obtain a consistent and improved class and activity diagram is a major challenge. First, we have to deal with a multi-dimensional optimization problem, and second, we may have to accept temporarily inconsistencies to ultimately reach even better solutions.

7.1.2 Challenges in Refactoring at Model Level

Finding an optimal sequence of refactorings on class diagrams and the corresponding co-refactorings on activity diagrams in order to accomplish a high quality of both views on a software system is a challenging task, because the effects of refactorings may improve the quality of one view, while they decrease the quality of the other. In this section, we introduce some well-known quality metrics that we use to evaluate the overall quality of the design and discuss the refactorings of class diagrams and the corresponding co-refactorings of activity diagrams that can be applied to improve the quality of both views. Based on these quality metrics and refactorings, we showcase the challenge of finding an optimal sequence based on a small example. However, at the same time we like to stress that our approach is not limited to these specific multi-view refactoring problem, but maybe it can be used as a general approach to tackle also another multi-view refactoring scenarios.

7.1.2.1 Quality Metrics

It is not a trivial task to select appropriate quality metrics to determine quality of class and activity diagram. Among the several metrics have been proposed to evaluate the structural quality of software artifacts (e.g., [124]), many have been successfully adopted for evaluating the structural design quality of UML (meta-)models, e.g., by Ma *et al*. [125]. Based on those works, we selected several metrics for class diagrams and activity diagrams (for activity diagrams we mostly based our metrics on existing work in the field of business processes, e.g., [126]) covering their design size and complexity (e.g., number of attributes and methods per class, number of parameters of methods, etc.), their coupling and encapsulation (e.g., number of associations, number data accesses over associations), as well as their abstraction (e.g., inheritance depth, number of polymorphic methods).

7.1.2.2 Refactorings

The refactoring of object-oriented programs is a well-researched domain [88] and many of the identified refactorings for object-oriented programs have been adopted for the refactoring of design models [127]. In this work, we consider refactorings that are applicable on class diagrams. However, the problem becomes challenging when we also have to identify the necessary co-refactorings for activity diagrams. The co-refactoring of activities is necessary after applying a refactoring to the class diagram in order to maintain the validity of consistency rules among classes and activities. A complete list of the considered class diagram refactorings and the corresponding co-refactorings of activities is available in [128]. This makes the problem of refactoring at the model level inherently more challenging, but at the same time gives the developer a better overview of different system states.

When using class diagrams and activity diagrams to represent the structure and behavior of programs, the consistency rules between class diagrams and activity diagrams largely correspond to the static semantics rules between classes and statements of an object-oriented, statically-typed programming language. To avoid ambiguities regarding the semantics of classes, activities, and actions, we adopt the semantics of the Foundational Subset For Executable UML (fUML) to define consistency rules and to derive necessary co-refactorings. As an example for such consistency rules, we may consider a *ReadStructuralFeatureValueAction* in an activity, which obtains the value of a specific feature from an object. The consistency rule of this action with respect to the class diagram is that the feature to be read must be a direct or inherited feature of the object's class. Moreover, the feature must be visible in the current context. The complete list of refactorings and co-refactoring is shown in Table 15.

We consider 15 well-known refactorings of class diagrams [127] [58] ranging from moving features, such as properties and operations, through extracting classes or superclasses from other classes, as well as pushing down and pulling up features along inheritance relationships, through to replacing inheritance with delegation and vice versa. For each of those refactorings of class diagrams, we identified the necessary co-refactorings for activity diagrams to maintain the validity of consistency rules between classes and activities. For instance, if a new class is extracted from one class and, thereby, a new association is added from the original class to the extracted class and one or more features (properties and operations) of the original class are moved to the new extracted class, all *StructuralFeatureValueActions* that access the moved features have to be prepended with a *ReadStructuralFeatureValueAction* that first reads the introduced association to navigate from the original class to the extracted class; otherwise, moved features would not be accessible in the object that is of the type of the original class. Note that in certain scenarios, it might not be possible to re-establish the validity of all conformance relationships with a co-refactoring of activities. For instance, when a private property of a class is pulled-up to its superclass and there are activities in the subclass reading this private property, we would have to pull-up this activity and the corresponding operation too. However, if this activity also reads other private properties that were not pulled up into the superclass, we cannot pull-up the activity and the operation; thus, it is not possible to establish valid conformance rules.

Table 15: The list of refactorings and corresponding co-refactorings

Name of refactoring:	Description	Co-evolution process
Rename Class	Changes the name of a class with a new name, and updates its references.	No co-refactoring needed, because in the abstract syntax the element references do not break with renames
Replace Inheritance with Delegation	Replaces a direct inheritance relationship with a delegation relationship.	We assume that a delegation relationship is a 1..1 association For each operation of the original super class, introduce a new operation in the original subclass and activities that navigate through the delegation association (ReadStructuralFeatureValueAction) and call the respective operation (CallOperationAction). Existing CallOperationActions calling the original superclass's operation have to be changed in order to call the new introduced operation in original subclass instead. ReadIsClassifiedObjectActions (instance of) have to be adapted, because the original subclass is not a subclass of the original superclass anymore.
Replace Delegation with Inheritance	Replaces a delegation relationship with a direct inheritance relationship.	Delete operations and activities from new subclass. Existing CallOperationActions must be changed to call the operation of the new superclass instead.

Extract Subclass	Adds a new subclass to class C and moves the relevant features to it.	The CreateObjectAction that created the object of class C must be changed to the create an object of the new subclass instead, if features that are pushed down in this refactoring are used on the created object. Types of operation parameters have to be changed to the new subclass if the respective operation accesses features which are now only available in the subclass. For methods being pushed down to the new subclass, see Push Down Method / Field.
Extract Superclass	Adds a new super class to class C and moves the relevant features to it.	For the features being pulled up to the new superclass, see Pull Up Method / Field. Co-evolution would be beneficial for code quality; the superclass should be used instead of the subclass wherever it is possible; that is, where no features are used that remain in the respective subclass.
Collapse Hierarchy	Removes a class from an inheritance hierarchy.	Adaptation necessary; see Extract Superclass and Extract Subclass
Inline Class	Moves all features of a class into another class and deletes it.	We assume that a 1..1 association to the deleted class exists Access to attributes and call of operations have to be adapted (no navigation through association using a ReadStructuralFeatureValueAction is needed anymore). Adapt all references to deleted class and use class containing all its features instead in CreateObjectAction, ReadIsClassifiedObjectAction, parameter types, etc.

Extract Class	Creates a new class and moves the relevant features from the old class into the new one.	We assume that a 1..1 association to the new class is introduced Delegating operations have to be added for operations moved to extracted class. Read/Add/ClearStructuralFeatureValueActions and CallOperationValueActions have to be adapted. Before these can be executed, the object of the extracted class has to be obtained first through a ReadStructuralFeatureValueAction on the association pointing to the extracted class. Usages of non-encapsulated and non-private attributes outside of the class from which the features were extracted having to be adapted (navigation to extract object has to be added). CreateObjectAction and respective linking for the extracted class has to be added wherever the existing class was instantiated (also DestroyObjectAction for new class has to be added).
Push Down Method	Moves a method from a class to those subclasses that require it.	If the pushed-down operations were pushed-down into multiple subclasses, these operations are moved *only from to one subclass* and *copied* from the other subclasses; thus, for the references to those operations must be adapted in all CallOperationActions depending on the type of the object on which the operation is called. It must be ensured that the moved operations still have access to the used features (i.e., private attributes and operations in the superclass C must not be used in Read/Add/ClearStructuralFeatureValueActions, CallOperationAction, etc. in the moved operations). Pushed-down operations must be non-private, if they are calling somewhere in the superclass or on the level of the superclass type, because otherwise the pushed-down operations would not be accessible anymore.

		If clients of the superclass call the operation, they must use the/a subclass instead (thus CreateObjectActions or parameter types must be adopted).
Pull Up Method	Moves a method of some class (es) to the immediate superclass.	If the pulled-up operations were pulled-up from multiple subclasses, these operations are moved *only from one class* and *removed* from the other subclasses; thus, for all CallOperationActions that point to the removed operation, the corresponding moved operation in the new superclass has to be used instead of the deleted ones. Pulled up operations must be non-private, if they are used somewhere in the subclass, because otherwise they would not be accessible anymore from the subclasses.
Rename Method	Changes the name of a method to a new one, and updates its references.	No co-refactoring needed
Push Down Field	Moves a field from a class to those subclasses that require it.	If the pushed-down fields were pushed-down to multiple subclasses, these fields are moved *only from to one subclass* and *copied* from the other subclasses; thus, for the references to those fields must be adapted in all StructuralFeatureValueActions depending on the type of the object on which the field is accessed. Pushed-down fields must be non-private, if they are accessed somewhere in the superclass or on the level of the superclass type, because otherwise the pushed-down fields would not be accessible anymore. If clients of the superclass access the pushed-down fields, they must use the subclass instead (thus CreateObjectActions or parameter types must be adopted).

Pull Up Field	Moves a field from some class(es) to the immediate superclass.	If the pulled-up fields were pulled-up from multiple subclasses, these features are moved *only from one class* and *removed* from the other subclasses; thus, for all actions that access those removed features, the respective corresponding moved field in the new superclass has to be used instead of the deleted ones in StructuralFeatureValueActions. Pulled up fields must be non-private, if they are used somewhere in the subclass, because otherwise they would not be accessible anymore from the subclasses.
Rename Field	Changes the name of a field to a new name, and updates its references.	No co-refactoring needed
Encapsulate Field	Creates getter and setter methods for the field and uses only those to access the field	Getter and setter activity have to be created (ReadSelf & Read/AddStructuralFeatureValueAction). Replace StructuralFeatureValueActions to that field with CallOperationActions to the getter and setter respectively.

7.2 Approach Overview

7.2.1 Detection Scheme

The goal of our approach is to generate the best refactoring sequence that improve the quality of different diagrams at the same time and also preserve some behavior preservation constraints. Therefore, we use a multi-objective optimization algorithm to compute an optimal sequence of refactorings in terms of finding trade-offs between maximizing the quality of class diagrams and activity diagrams, and minimizing the number of violated behavioral preservation constraints. In fact, the evaluation of refactorings applied on a class diagram depends on their impact on the related diagrams such as activity diagrams. In addition, activity diagrams should be used to verify if the behavior is changed after applying the refactorings on the class diagram.

The general structure of our approach is sketched in Figure 40. The search-based process takes as inputs the list of 15 possible types of refactoring that can be applied to a class diagram, the list behavioral preservation constraints, the co-evolution rules to generate the equivalent activity diagram from a refactored class diagram, a list of metrics to evaluate the quality of class diagrams and activity diagrams, and the system design to refactor. The process of generating a solution can be viewed as the mechanism that finds the best refactorings sequence among all possible solutions that minimizes the number of violated behavioral constraints, maximizes the quality of the class diagram and maximizes also the quality of the related activity diagram. The size of the search space is determined not only by the number of refactorings but also by the order in which they are applied. Due to the large number of possible refactoring combinations and the three objectives to optimize, we considered model refactoring as a multi-objective optimization problem. In the next subsection, we describe the adaptation of NSGA-II proposed by Deb et al. [82] to our problem domain.

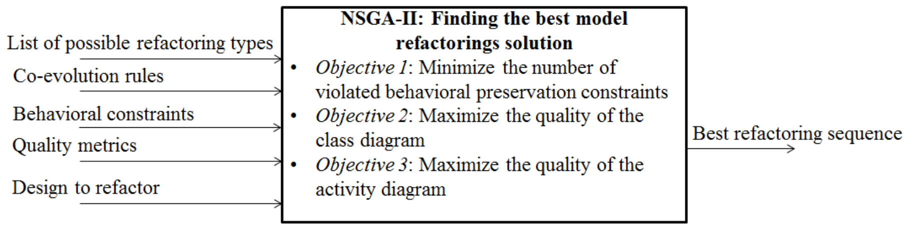

List of possible refactoring types
Co-evolution rules
Behavioral constraints
Quality metrics
Design to refactor
NSGA-II: Finding the best model refactorings solution
• Objective 1: Minimize the number of violated behavioral preservation constraints
• Objective 2: Maximize the quality of the class diagram
• Objective 3: Maximize the quality of the activity diagram
Best refactoring sequence

Figure 40: Multi-objective model refactoring: overview

The search based NSGA-II drives the population of candidate solutions to evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of near-optimal solutions, called non-dominated solutions or the Pareto front. A non-dominated solution is one that provides a suitable compromise between all objectives without degrading any of them. As described in Algorithm below (, the first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation (line 2). Both populations are merged into a new population R_0 of size N (line 5). Fast-non-dominated-sort is the algorithm used by NSGA-II to classify individual solutions into different dominance levels. Indeed, the concept of Pareto dominance consists of comparing each solution x with every other solution in the population until it is dominated by one of them. If no solution dominates it, the solution x will be considered non-dominated and will be selected by the NSGA-II to be a member of the Pareto front. If we consider a set of objectives f_i , $i,j \in 1 \ldots n$, to maximize, a solution x dominates x'

$$iff \ \forall i, f_i(x') \leq f_i(x) \text{ and } \exists j \mid f_j(x') < f_j(x).$$

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0 Then, after taking these solutions out, fast-non-dominated-sort calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance to make the selection (line 9). This parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line

13), and the first (N-$|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to the stop criteria (line 4). It is interesting to mention that NSGA-II is an elitist algorithm that does not use any explicit archive of elite individuals. In fact, elitism is ensured by the crowded comparison operator that prefers solutions having better Pareto ranks. In this way, NSGA-II preserves elite solutions by keeping best non-dominated fronts in the race, and when considering the last non-dominated front, only least crowded solutions are selected from this latter to build the next population of N individuals.

```
Create an initial population $P_0$
Generate an offspring population $Q_0$
$t$=0;
**while** *stopping criteria not reached* **do**
$R_t = P_t \cup Q_t$;
$F$ = fast-non-dominated-sort ($R_t$);
$P_{t+1} = \emptyset$ and $i$=1;
**while** $| P_{t+1}| + |F_i| \leq N$ **do**
Apply crowding-distance assignment($F_i$);
$P_{t+1} = P_{t+1} \cup F_i$ ;
$i = i+1$;
**end**
$Sort(F_i, \prec n)$;
$P_{t+1} = P_{t+1} \cup F_i[1 : (N-| P_{t+1} |)]$;
$Q_{t+1}$ = create-new-pop($P_{t+1}$);
$t = t+1$;
**end**
```

Figure 41: High-level pseudo-code of NSGA-II

7.2.2 Solution Coding

This section describes how NSGA-II can be used to find design refactoring solutions with multiple conflicting objectives. To apply NSGA-II to a specific problem, the following elements have to be defined:

- Representation of the individuals;
- Evaluation of individuals using a fitness function for each objective to optimize to determine a quantitative measure of their ability to solve the problem under consideration;
- Selection of the individuals to transmit from one generation to another;
- Creation of new individuals using genetic operators (crossover and mutation) to explore the search space.

Next, we describe the adaptation of the design of these elements for the generation of model refactoring solutions using NSGA-II.

7.2.2.1 Solution Representation

To represent a candidate solution (individual), we used a vector representation. Each vector's dimension represents a class diagram refactoring operation. Thus, a solution is defined as a long sequence of refactorings applied to different parts of the design. The size of the solution represents the number of refactoring (dimensions) in the vector. When created, the order of applying these refactorings corresponds to their positions in the vector. In addition, for each refactoring, a set of controlling parameters (stored in the vector), e.g., actors and roles are randomly picked from the class diagram to be refactored and stored in the same vector. For example, the controlling parameters of a move method refactoring are the source and target classes and the method to move from the source class as described in the example of Figure 37. Thus, the refactorings and its parameters are encoded as logic predicates (Strings). Each dimension of the vector is a logic predicate describing the refactoring type and its parameters.

An example of a solution is given in Figure 42 on the class diagram of the motivating example discussed earlier. This solution contains 3 dimensions that correspond to three refactorings applied to different parts of the source class diagram. For instance, the predicate *move method* (*Circle, Point, distance(int, int)*) means that the method *distance(int, int)* is moved from class *Circle* (source class) to class *Point* (target class).

move method (*Circle, Point, distance(int, int)*)
move field (*Point, Circle, x*)
extract class(*Circle, Proprieties, radius, distance (int, int, int)*)

Figure 42: Representation of an NSGA-II individual

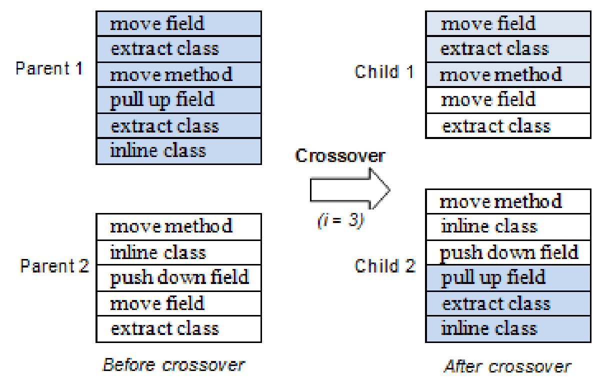
After the generation of the refactoring for the class diagram, we automatically generate the equivalent activity diagram refactorings using the co-evolution rules described in previous section. These activity diagram refactorings are also represented in a vector similar to those applied to the class diagram. Moreover, when creating a sequence of refactorings (individuals), it is important to guarantee that they are feasible and that they can be legally applied. For example, to apply the refactoring operation *move method(Circle, Point, distance())*, a number of necessary preconditions should be satisfied, e.g., *Circle* and *Point* should exist and should be classes; *distance()* should exist and should be a method; the classes *Circle* and *Point* should not be in the same inheritance hierarchy; the method *distance()* should be implemented in *Circle*; the method signature of *distance()* should not be present in class *Point*. As postconditions, *Circle*, *Point*, and *distance()* should exist; *distance()* declaration should be in the class *Point*; and *distance ()* declaration should not exist in the class *Circle*.

7.2.2.2 Solution Variation

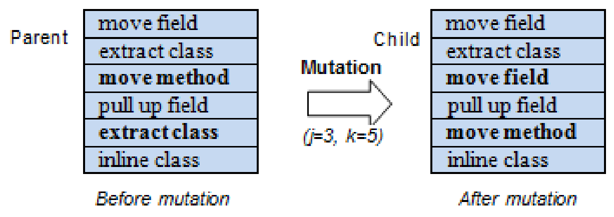
To guide the selection process, NSGA-II uses a binary tournament selection based on dominance and crowding distance. NSGA-II sorts the population using the dominance principle which classifies individual solutions into different dominance levels. Then, to construct a new offspring population Q_{t+1}, NSGA-II uses a comparison operator based on a calculation of the crowding distance to select potential individuals having the same dominance level.

To better explore the search space, the crossover and mutation operators are defined. For crossover, we use a single, random, cut-point crossover. It starts by selecting and splitting at random two parent solutions. Then, crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and, for the second child, the first part of the second parent with the second part of the first parent. Each solution has a length limit in terms of number of refactorings. When applying the crossover operator, the new solution may have higher number of refactorings than the length limit (input of the algorithm). Thus, the algorithm randomly eliminates some of the dimensions of the vector (refactorings) to respect the size constraint. As illustrated in Figure 7.a, each child combines some of the refactoring operations of the first parent with some ones of the second parent. In any given generation, each solution will be the parent in at most one crossover operation.

The mutation operator picks randomly one or more operations from a sequence and replaces them with other ones from the initial list of possible refactorings. An example is shown in Figure 43. After applying genetic operators (mutation and crossover), we verify the feasibility of the generated sequence of refactoring by checking the pre and post conditions. Each refactoring operation that is not feasible due to unsatisfied preconditions will be removed from the generated refactoring sequence. The new sequence after applying the change operators is considered valid in our NSGA-II adaptation if the number of rejected refactorings is less than 5% of the total sequence size.



(a) Cross-over operator



(b) Mutation operator

Figure 43: Changes operators in Model Refactoring.

Overall, the adaptation of NSGA-II to our model refactoring problem is generic thus it can be easily extended to include other modelling languages by adding a new fitness function (to evaluate the quality of the new type of models). The solution representation and change operators will remain the same. Of course, the input should be also extended to integrate new quality metrics related to the new considered modelling language that will be used by the new fitness function as a new objective to optimize

7.2.2.3 Solution Evaluation

After creating a solution, it should be evaluated using fitness functions. Since we have three objectives to optimize, we are using three different fitness functions to include in our NSGA-II adaptation. We used the following fitness functions:

Quality of the class diagram fitness function is calculated using a set of 11 quality metrics used by the QMOOD model as shown in Table 16. The table shows cases the QMOOD metrics being used along with computation of quality attributes. All the 11 metrics are aggregated in one fitness function with equal importance and normalized between 0 and 1.

Table 16: QMOOD metrics for design properties

Design Property	**Metr**	**Description**
Design size	DSC	Design size in classes
Complexity	NOM	Number of methods
Coupling	DCC	Direct class coupling
Polymorphism	NOP	Number of polymorphic methods
Hierarchies	NOH	Number of hierarchies
Cohesion	CAM	Cohesion among method in class
Abstraction	ANA	Average number of ancestors
Encapsulation	DAM	Data access metric
Composition	MOA	Measure of aggregation
Inheritance	MFA	Measure of functional abstraction
Messaging	CIS	Class interface size

Table 17: Computation of Quality Attribute using QMOOD Metrics

Quality attribute	**Definition and Computation**
Reusability	A design with low coupling and high cohesion is easily reused by other designs. 0.25*Coupling+0.25*Cohesion+0.5*Messaging+0.5*Design Size
Flexibility	The degree of allowance of changes in the design. 0.25*Encapsulation-0.25*Coupling+0.5*Composition+0.5*Polymorphism

Understandability	The degree of understanding and the easiness of learning the design implementation details. 0.33*Abstraction+0.33*Encapsulation-0.33*Coupling+0.33*Cohesion-0.33*Polymorphism-0.33*Complexity-0.33*Design Size
Functionality	Classes with given functions that are publically stated in interfaces to be used by others. 0.12*Cohesion+0.22*Polymorphism+0.22*Messaging+0.22*DesignSize+0.22*Hierarchies
Extendibility	Measurement of design's allowance to incorporate new functional requirements. 0.5*Abstraction-0.5*Coupling+0.5*Inheritance+0.5*Polymorphism
Effectiveness	Design efficiency in fulfilling the required functionality. 0.2*Abstarction+0.2*Encapsulation+0.2*Composition+0.2*Inheritance+0.2*Polymorphism

- *Quality of the activity diagram fitness function* represents an aggregation (sum) of 12 metrics described in Table 18. All these metrics are normalized between 0 and 1.

Table 18: Activity diagrams metrics

Metric	Description
NP	Number of parameters
NNO	Number of nodes
NED	Number of edges
NAC	Number of actions
CFC	Control-Flow Complexity
ICOM	Inteface Comlexity
HAC	Halstead-based Activity Complexity
CNC	Coefficient of Network Complexity
FIFO	Fan-in/Fan-out Metrics for Activities
TD	Tree depth metric

TW	Tree width metric
LO	Locality

- *Number of violated behavioral constraints fitness function* checks how many behavioral constraints are violated by the generated refactoring solutions when applied to an activity diagram.

7.3 Validation

7.3.1 Research Questions

In our study, we assess the performance of our model refactoring approach of finding out whether it could generate meaningful sequences of refactorings that improve the structure of class diagrams and activity diagrams while preserving the behavior. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. To this end, we defined the following research questions:

RQ1: To what extent can the proposed approach improve the quality of class diagrams and activity diagrams?

RQ2: To what extent the proposed approach preserves the behavior while improving the quality?

RQ3: How does the proposed multi-objective approach based on NSGA-II perform compared to a mono-objective approach where only one objective is considered to improve the quality of class diagrams?

RQ4: How does the proposed multi-objective design refactoring approach performs compared to an existing model refactoring approach not based on heuristic search?

RQ5: Insight. How our multi-objective model refactoring approach can be useful for software engineers in real-world setting?

To answer **RQ1**, we validate the proposed design refactoring solutions to improve the quality of the system by evaluating their ability to fix some design defects that can be detected on class diagrams extracted from four open-source systems. We adapted earlier work [107] based on quality metrics rules to detect three types of design defects: Blob (it is found in designs where one large class tends to centralize the functionalities of a system, and the other related classes primarily exposing data.), Long Parameter List (methods with numerous parameters are a challenge to maintain, especially if most of them share the same data-type) and Data Clumps (interrelated data items which often occur as clump in the model. The same data items are often together in different places such as attributes in classes or parameters in method signatures). We defined a measure NFD, Number of Fixed Defects, which corresponds to the ratio of the number of corrected design defects over the initial number of detected defects on a class diagram before applying the suggested refactoring solution.

$$NFD = \frac{\#\,\text{fixed design defects on a class diagram}}{\#\,\text{defects before applyingrefactorings}}$$

It is also important to assess the refactoring impact on the design quality and not only on a class diagram. The expected benefit from refactoring is to enhance the overall software design quality as well as fixing design defects. The quality metrics considered by our approach can improve different aspects of the design quality related to reusability, flexibility, understandability, functionality, extendibility, and effectiveness. The improvement in quality can be assessed by comparing the quality before and after refactoring independently to the number of fixed design defects. Hence, the total gain in quality *G* before and after refactoring can be easily estimated as: $G-ClassDiagram = \frac{\sum_{i=1}^{i=12} |q'_i - q_i|}{12}$ and $G-ActivityDiagram = \frac{\sum_{i=1}^{i=11} |q'_i - q_i|}{11}$,

where *q'* q'_i and q_i represents the value of the quality attribute *i* respectively after and before refactoring. As described in the previous section, we considered a total of 12 metrics related to class diagrams and 11 metrics for activity diagrams.

To answer **RQ2**, we asked groups of potential users of our refactoring tool to evaluate, manually, whether the suggested refactorings are feasible and preserve the behavior or not. The users evaluated the entire best solutions proposed by our approach. We define the metric "refactoring precision" (RP) which corresponds to the number of meaningful refactorings over the total number of suggested refactoring operations: $RP = \frac{\#\text{feasible refactorings}}{\#\text{proposedrefactorings}}$

To answer **RQ3**, we compare our approach to a mono-objective formulation using a genetic algorithm (GA) that considers the refactoring suggestion task only from the class diagram quality improvement perspective (single objective). The use of a single-objective algorithm is to show that the two objectives of our multi-objective formulation are conflicting. If the two objectives were not conflicting then the results of NSGA-II will be similar to GA. Thus, in that case we will not need to propose a multi-view approach.

To answer **RQ4**, we compared our design refactoring results with a recent tool, called DesignImpl proposed recently by Iman and Mel [129]that does not use heuristic search techniques. The current version of DesignImpl is implemented as an Eclipse plug-in that proposes a list of class diagram and code refactorings based on an interaction with the designer who specify the desired design based on an evaluation of the class diagram.

To answer **RQ5**, we asked a group of eight software engineers to refactor manually some of the detected design defects on the class diagrams, and then compare the results with those proposed by our tool. To this end, we define the following precision metric *MP* (Manual precision): $MP = \frac{|R| \cap |R_m|}{|R_m|}$, where R is the set of refactorings suggested by our tool, and R_m is the set of refactorings suggested manually by software engineers. In fact, several equivalent refactoring solutions can be proposed to improve the quality. Thus, the tool can propose some refactorings that are different than those proposed by the designers, but improve the overall quality of the design. Thus, MP corresponds to the portion of the correct refactorings after manually evaluating them by the developers (that can be dissimilar from their suggestions).

7.3.2 Experimental Setup

Our study considers 27 model fragments extracted from four open source projects using the IBM Rational Rose tool [130]: Xerces-J, GanttProject (Gantt for short), JFreeChart, and Rhino. Xerces-J is a family of software packages for parsing XML. GanttProject is a cross-platform tool for project scheduling. JFreeChart is a powerful and flexible Java library for generating charts. Finally, Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser. Table 19 summarizes for each model the number of detected design defects, as well as the number of model elements. A model fragment is a set of model elements extracted from the open source system. In fact, we extracted these model fragments from the different open source systems (we did not consider the open source system as one model but we extracted several fragments from these systems). The number of elements in Table 19 is not the number of model fragments, but the number of elements in all the model fragments per open source system.

We selected these systems for our validation because they range from medium to large-sized open source projects that have been actively developed over the past 10 years, and include a large number of design defects. Our study involved 6 subjects from the University of Michigan and some of them are working in automotive industry companies. Subjects include 4 master students in Software Engineering and 2 PhD students in Software Engineering. 4 of them are working in industry as senior software engineers. All the subjects are volunteers and familiar with Java development. The experience of these subjects on Java programming ranged from 6 to 16 years. The subjects manually evaluated the best refactoring solutions proposed by the different techniques. In addition, they manually refactor some of the detected design defects on the class diagrams. This outcome was compared with the solutions proposed by our techniques.

Parameter setting has a significant influence on the performance of a search algorithm on a particular problem instance. For this reason, for each algorithm and for each system, we perform a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 100000 evaluations of all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.8; mutation probability = 0.5 where the probability of gene modification is 0.3; stopping criterion = 100000 evaluations. The elitism can cause premature convergence since population members would be converging towards the same region of the search space. Based on our experimentations, we concluded that for our problem, it is effective and efficient to use an elitist schema while using a high mutation rate (0.8). The latter allows the continued diversification of the population, which discourage premature convergence to occur.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 99% confidence level ($\alpha = 1\%$). The latter verifies the null hypothesis H_0 that the obtained results of the two algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H_1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.01) means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than α (> 0.01) means the opposite. In this way, we could decide whether the outperformance of NSGA-II over one of each of the others (or the opposite) is statistically significant or just a random result.

Table 19: The systems studied for Model Refactoring

Systems	Release	#Elements	#Smells
JFreeChart	v1.0.9	81	22
GanttProject	v1.10.2	114	28
Xerces-J	V2.7.0	96	31
Rhino	v1.7R1	88	26

The Wilcoxon signed-rank test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. The effect size could be computed by using the Cohen's *d* statistic [131]. The effect size is considered: (1) small if $0.2 \leq d < 0.5$; (2) medium if $0.5 \leq d < 0{:}8$, or (3) large if $d > 0.8$. Table 20 gives the effect sizes in addition to the p-values of the Wilcoxon test when comparing the results of NSGA-II to the GA.

Table 20: Statistical test results when comparing NSGA-II to the mono-objective approach

Scenario	JFreeChart	GanttProject	Xerces-J	Rhino
p-value	6.12E-06	3.51E-09	8.27E-04	2.32E-04
Effect size	0.13	0.69	0.52	0.82

We note that the mono-objective algorithm provides only one refactorings solution, while NSGA-II generates a set of non-dominated solutions. In order to make meaningful comparisons, we select the best solution for NSGA-II using a knee point strategy [132]. The knee point corresponds to the solution with the maximal trade-off between the three objectives. Hence moving from the knee point in either direction is usually not interesting for the user.

7.3.3 Result and Discussion

As described in Figure 44, after applying the proposed refactoring operations by our approach (NSGA-II), we found that, on average, more than 85% of the detected design defects (model smells) were fixed (*NFD*) for all the class diagrams extracted from the four studied systems.

This high score is considered significant to improve the quality of the refactored diagrams by fixing the majority of defects that were from different types. We found that the majority of non-fixed defects are related to the *blob* type. The similar observation is also valid for the other techniques used in our experiments. This type of defect usually requires a large number of refactoring operations and is known to be very difficult to fix [133].

Another observation is that our technique may introduce some new defects after refactoring. These new defects can be fixed by our approach since the fitness function counts the number of remaining defects in the system (to minimize) after executing the refactorings sequence.

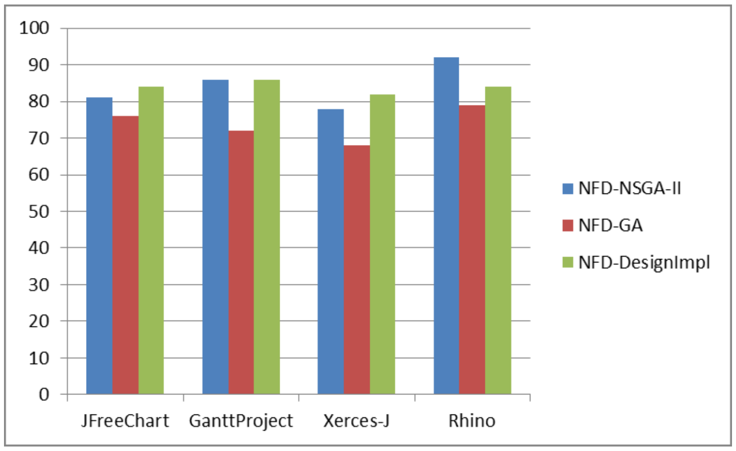


Figure 44: NFD median values of NSGA-II, GA and DesignImpl over 51 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

In Figure 45 and Figure 46, we show the obtained gain values that we calculated for all the metrics considered for both class diagrams and activity diagrams before and after refactoring for each studied system. We found that the diagrams quality increases across all the quality factors. As a consequence, we noticed that the quality of both class diagrams and activity diagrams is improved. The highest quality improvement scores of all systems are mainly observed on class diagrams.

To sum up, we can conclude that our approach succeeded in improving the design quality not only by fixing the majority of detected model smells but also by improving the user understandability, the reusability, the flexibility, as well as the effectiveness of the refactored design. Figure 46 shows that all the quality metrics were improved on all the systems except the functionality attribute for JFreeChart and Xerces-J. We looked to experiments data to understand the reason of the loss on Functionality of JFreeChart and Xerces-J. In fact, the functionality measure is calculated as the following:

$$Measure = 0.12*Cohesion+0.22+Polymorphism+0.22*Messaging+0.22*DesignSize+0.22*Hierarchies.$$

We found that the design size of some JFreeChart and Xerces-J models after refactoring was lower than the design size before refactoring. Several move methods/fields were applied, leading to some empty classes after refactoring (thus not considered in the design size anymore). Furthermore, we found that the best refactoring solution included few extract class refactorings thus the design size was not increased with new classes (model elements). This can be explained by the fact that JFreeChart and Xerces-J were the only systems that did not include several large classes and most of the classes have a small or medium size in terms of number of methods. Of course, the overall functionalities of the system were the same before refactoring as demonstrated later in RQ2 by the manual refactoring evaluation (RP).

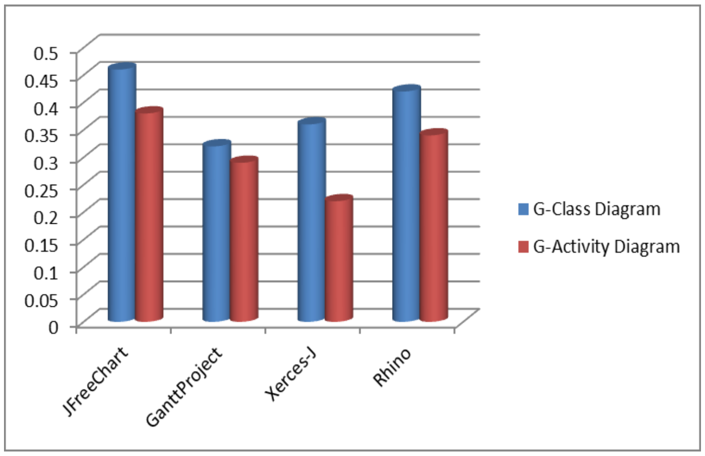


Figure 45 : Design quality improvements median values for Class diagrams and Activity diagrams of NSGA-II, GA and DesignImpl over 51 independent simulation runs.

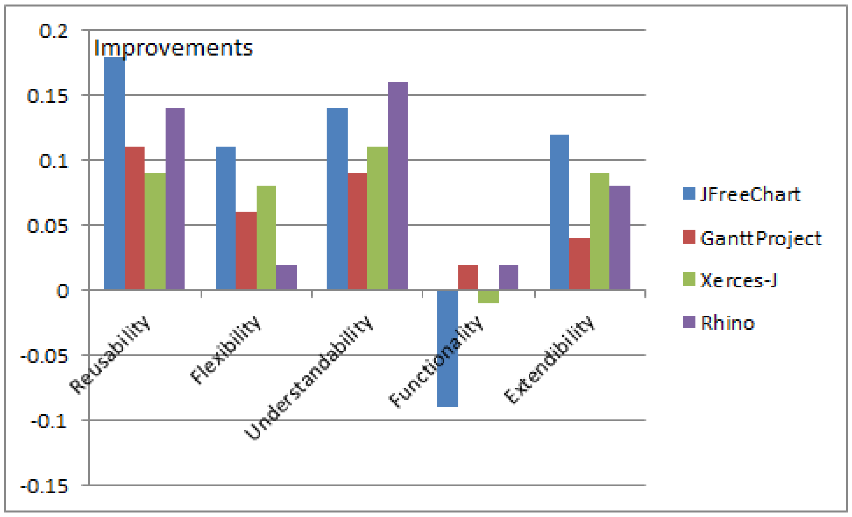


Figure 46 : Quality factors median values of NSGA-II, over 51 independent simulation runs.

Lets now discuss the results for RQ2. As described in Figure Figure 47, we found that an average of more than 80% of proposed refactoring operations are considered as feasible and do not generate behavior incoherence. A slight loss in the *NFD* and *G* is largely compensated by the significant improvement in terms of refactorings feasibility and behavior preservation.

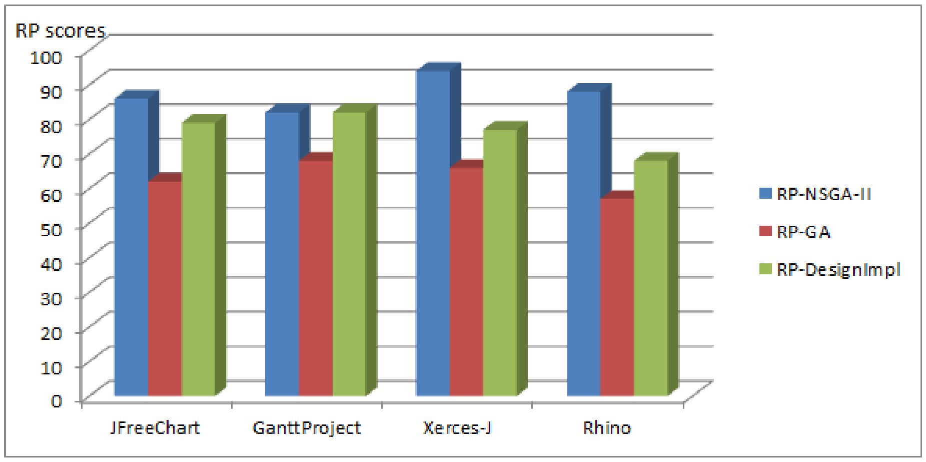


Figure 47: The refactoring precision (RP) median values of NSGA-II, GA and DesignImpl over 51 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

To answer results for RQ3 we refer to Figure 45,Figure 46Figure 47, it is clear that our proposal outperforms both the mono-objective GA and DesignImpl. Figure 44Figure 45 show that our approach improves the quality of the design with a comparable values to both GA and DesignImpl. However, in terms of behavior preservation it is clear that our approach provides much more feasible refactorings than GA and DesignImpl for all the systems considered in our experiments. This can be explained by the fact that our proposal checks the behavior preservation using the activity diagrams, however existing approaches did not consider it.

For RQ4 we need to evaluate the relevance of our suggested design refactorings for real software engineers, we compared the refactoring strategies proposed by our technique and those proposed manually by the subjects (software engineers) to fix several model smells on the diagrams considered in our experiments. Figure 48 shows that most of the suggested refactorings by NSGA-II are similar to those applied by developers with an average of more than 70%. In fact, we calculated the intersection between the recommended refactorings by NSGA-II and the manually suggested refactorings by the subjects over the total number of recommend refactorigs. Some defects can be fixed by different refactoring strategies and also the same solution can be expressed in different ways. Thus, we consider that the average of precision of more than 70% confirms the efficiency of our tool for real developers to automate the refactoring process. It is clear that our proposal outperforms GA and DesignImpl on all the diagrams, this can be explained by the fact that both of these techniques do not consider the behavioral constraints defined on the activity diagrams.

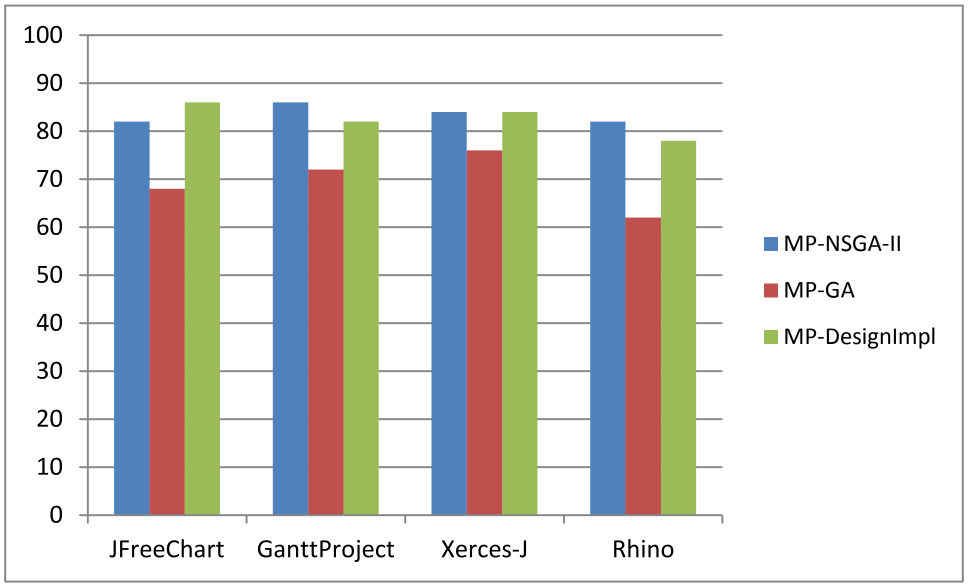


Figure 48: The MP median values of NSGA-II, GA and DesignImpl over 51 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).

Another advantage related to the use of our multi-objective approach is the diversity of non-dominated solutions as described in Figure 49. Figure 49 depicts the Pareto front obtained on Xerces using NSGA-II to optimize the three considered objectives. Similar fronts were obtained on the remaining systems. The 2-D projection of the Pareto front helps software engineers select the best tradeoff solution between the three objectives based on their own preferences.

The selection of the "best" solution is based on the preferences of the developer. In fact, the developer may select a solution providing a high quality of class and activity diagrams but violating several constraints since he has enough time to fix them before the next release for example. In another situation, the developer may select a solution that do not violate constraints (or only violating few of them) and slightly increase the quality of the models because he do not have enough time to fix the errors created by the constraints or if he want to minimize the risk related to the new refactorings. In case that the developer do not have any specific preferences and he wants to optimize all the three objectives at the same time then he can select the solution at the knee point or the closest solution to the ideal point (the ideal point is (0,0,0) if all the objectives are to minimize and normalized between zero and one).

Based on the plots of Figure 49, the engineer could degrade quality in favor of behavior preservation. In this way, the user can select the preferred refactoring solution to realize. This is a very interesting feature, since recent studies showed that software developers still select refactoring solutions that could change the behavior with a high quality improvement because they believe that it is easy to fix the behavior violation.

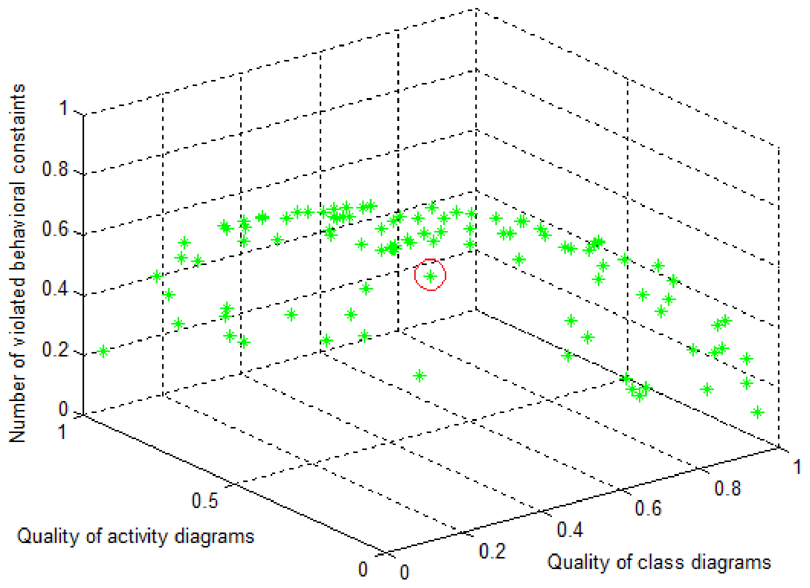


Figure 49: Pareto front for NSGA-II obtained on Xerces-J.

Chapter 8: Conclusion

Model-Driven Engineering is applied increasingly to cope with the complexity of software systems by raising the level of abstraction. This work utilizes the power of refactoring in multi objective optimization domain to solve emerging problems in the field of model driven engineering. Today model driven engineering is being used to modify or merge existing models to accomodate changes in customer requirements. However, to accomplish this reliably with quick turn around time software developers need access to powerful tools in change detection, model merging, defect detection and correction in model merging. Consequently, techniques to support building models collaboratively are a necessity. This research tends to address this as a multi objective optimization problem. The four major objectives studies in this research are listed below:

- Design, implement and evaluate a multi-objective approach to detect model changes as a sequence of refactoring applications.
- Propose a multi-objective formulation of the problem of model merging that aims to find the best trade-off between minimizing the number of omitted operations and maximizing the number of successfully applied important operations.
- The use of genetic programming to generate detection rules of design defects using quality and changes history metrics.
- Design, implement and evaluate a multi-objective framework for the refactoring of software models to find a trade-off between improving the quality of different diagrams at the same time such as class diagrams and activity diagrams.

We addressed each objective as a separate search based optimization problem with a set of objectives and constraints. We have evaluated each scheme on multiple open source systems and validated the applicability of NSGA-II based search scheme in solving the software development problems in model driven engineering.

8.1 Discussions on Change Detection in Model

Software models, defined as code abstractions, are iteratively refined, restructured and evolved due to many reasons such as reflecting changes in requirements or modifying a design to enhance existing features. For understanding the evolution of a model a-posteriori, change detection approaches have been proposed for models. The majority of existing approaches are successful to detect atomic changes. However, composite changes, such as refactorings, are difficult to detect due to several possible combinations of atomic changes or eventually hidden changes in intermediate model versions that may be no longer available. Moreover, a multitude of refactoring sequences maybe used to describe the same model evolution. Chapter 4 focused on a multi-objective approach to detect model changes as a sequence of refactorings. The approach takes as input an exhaustive list of possible types of model refactoring operations, the initial model, and the revised model, and generates as output a list of refactoring applications representing a good compromise between the following two objectives (i) maximize the similarity between the expected revised model and the generated model after applying the refactoring sequence on the initial model, and (ii) minimize the number of atomic changes used to describe the evolution. Due to the huge number of possible refactoring sequences, a metaheuristic search method is used to explore the space of possible solutions. To this end, we use the non-dominated sorting genetic algorithm (NSGA-II) to find the best trade-off between our two objectives. The algorithm starts by randomly generating a set of refactoring combinations, executing them on the initial model to generate a revised model, and then evaluates the quality of the proposed solution (refactorings) by comparing the generated revised model and the expected one. The goal is to maximize their similarity and at the same time to minimize the number of generated refactorings.

The chapter also exhibited the results of an empirical study of our multi-objective model changes detection technique as applied on various versions of real-world models taken from open source projects and one industrial project. The approach is compared with random search, multi-objective particle swarm optimization (MOPSO), an existing mono-objective changes detection approach and two model changes detection tools not based on computational search. The statistical test results provide evidence to support the claim that the scheme enables the generation of changes detection solutions with correctness higher than 85%, in average, using a variety of real-world scenarios.

Although the approach has been evaluated with real-world models with a reasonable number of changes to detect, future work should focus on larger models and with larger lists of refactoring types. This is not only necessary to investigate deeper the applicability of the approach in practice, but to also study the performance of the approach when dealing with very large models.

8.2 Discussions on Model Merging

Chapter 5 described in detail a novel multi-objective approach for merging parallel versions of models by finding the best operation sequence that takes into account the importance of the operations to merge. Having such a sequence is very useful in model versioning to find a tentative merge, as a basis for subsequently resolving the remaining conflicts manually. Therefore, a merged model is essential that maximizes the combined effect of all operations that have been applied by multiple developers in parallel to the same model. This is achieved by finding an optimal (potentially interleaved) order of operations that minimizes the number of disabled operations. Furthermore, it is useful for team managers to select merging solutions that include the most important operations while minimizing the number of disabled ones. As the search space in terms of all possible sequences of operations is potentially huge and we have two conflicting objectives to optimize, we considered in this scheme the merging process as a multi-objective optimization problem.

We evaluated our scheme with seven real-world model evolutions extracted from different open source systems. The experiment results indicate clearly that the number of disabled operations is reduced significantly in comparison to the number of disabled operations without taking into consideration the different possible operation orders. Furthermore, the results provide strong evidence to support the claim that our proposal enables the generation of efficient model merging solutions to be comparable in terms of minimizing the number of conflicts to those suggested by existing approaches and to carry a high importance score of merged operations.

Although this scheme has been evaluated with real-world models with a reasonable number of applied operations, future work in this area needs to focus on larger models and with larger lists of operations applied in parallel. This is necessary to investigate not only the applicability of the approach in practice, but also to study the performance of our approach when dealing with very large models.

8.3 Discussions on Defect Detection

In chapter 6 we introduced a novel multi-objective approach to generate rules for the detection of code-smells. To this end, we used MOGP to find the best trade-offs between maximizing the detection of examples of code-smells and minimizing the detection of well-designed code examples. We evaluated our approach on seven large open source systems. All results were found to be statistically significant over 51 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$). Our MOGP results outperform existing studies by detecting most of the expected code-smells with an average of 87% of precision and 92% of recall.

This approach is inspired by contributions in the domain of Search-Based Software Engineering (SBSE) [134]. 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. Based on recent SBSE surveys [134], the use of multi-objective optimization is still limited in software engineering especially in the area of anti pattern detection.

8.4 Discussions on Defect Correction

Chapter 7 presented a multi-view refactoring approach taking into consideration multiple criteria to suggest good and feasible design refactoring solutions to improve the design quality. The suggested refactorings preserve the behavior of the design to restructure and consider the impact of refactoring a class diagram on related diagrams such as activity diagrams. Our search-based approach succeeded to find the best trade-off between these multiple criteria. Thus, our proposal produces more meaningful refactorings in comparison to some of those discussed in the literature. Moreover, the proposed approach was empirically evaluated on several diagrams extracted from four open-source systems, and compared successfully to an existing approach not based on heuristic search.

In our experiments, construct validity threats are related to the absence of similar work that uses multi-objective techniques for automated multi-view model refactoring. For that reason, we compare our proposal with GA-based approach and an existing semi-automated design refactoring technique. Another threat to construct validity arises because, although we considered 3 types of model smells, we did not evaluate the performance of our proposal with other model smell types. In future work, we plan to use additional model smell types and evaluate the results. For the selection threat, the subject diversity in terms of profile and experience could affect our study. First, all subjects were volunteers. We also mitigated the selection threat by giving written guidelines and examples of refactorings already evaluated with arguments and justification. Additionally, each group of subjects evaluated different operations from different systems using different techniques/algorithms.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 51 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 99% confidence level ($\alpha = 1\%$). However, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work by additional experiments. The parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. In fact, parameter tuning of search algorithms is still an open research challenge till today. We have used the trial-and-error method which one of the most used ones. However, the use of ANOVA-based technique could be another interesting direction from the viewpoint of the sensitivity to the parameter values.

External validity refers to the generalization of our findings. In this study, we performed our experiments on diagrams extracted from different widely-used open-source systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, and to other practitioners. Future replications of this study are necessary to confirm the general aspect of our findings and evaluate the scalability of our approach with larger models.

The proposed multi-objective approach provides a multi-view for software designers to evaluate the impact of suggested refactorings applied to class diagrams on related activity diagrams in order to evaluate the overall quality, and check their feasibility and behavior preservation. The statistical evaluation performed on models extracted from four open source systems confirms the efficiency of our approach.

8.5 Future Work

In this section we outline some of the directions for possible future work in this field. These future areas of research revolve around application of search based optimization techniques in solving problems in software engineering.

This work investigates fixing of design defects by analyzing class and activity diagrams. The work can be extended by adding addition views when suggesting refactorings such as sequence diagrams as well as object diagrams. Also the work can be extended by adding an additional set of refactoring and co refactoring operations to fix different types of model-smells.

The type of design defects being detected in this work involves structural metrics. However, history information based design defects e.g. shotgun surgery can also be detected by enhancing the detection rules, optimization objectives along with corresponding constraints. By adding the additional dimension of history based defect detection, the overall solution can be made more relevant to rapidly evolving large software systems which have development history spanning multiple years.

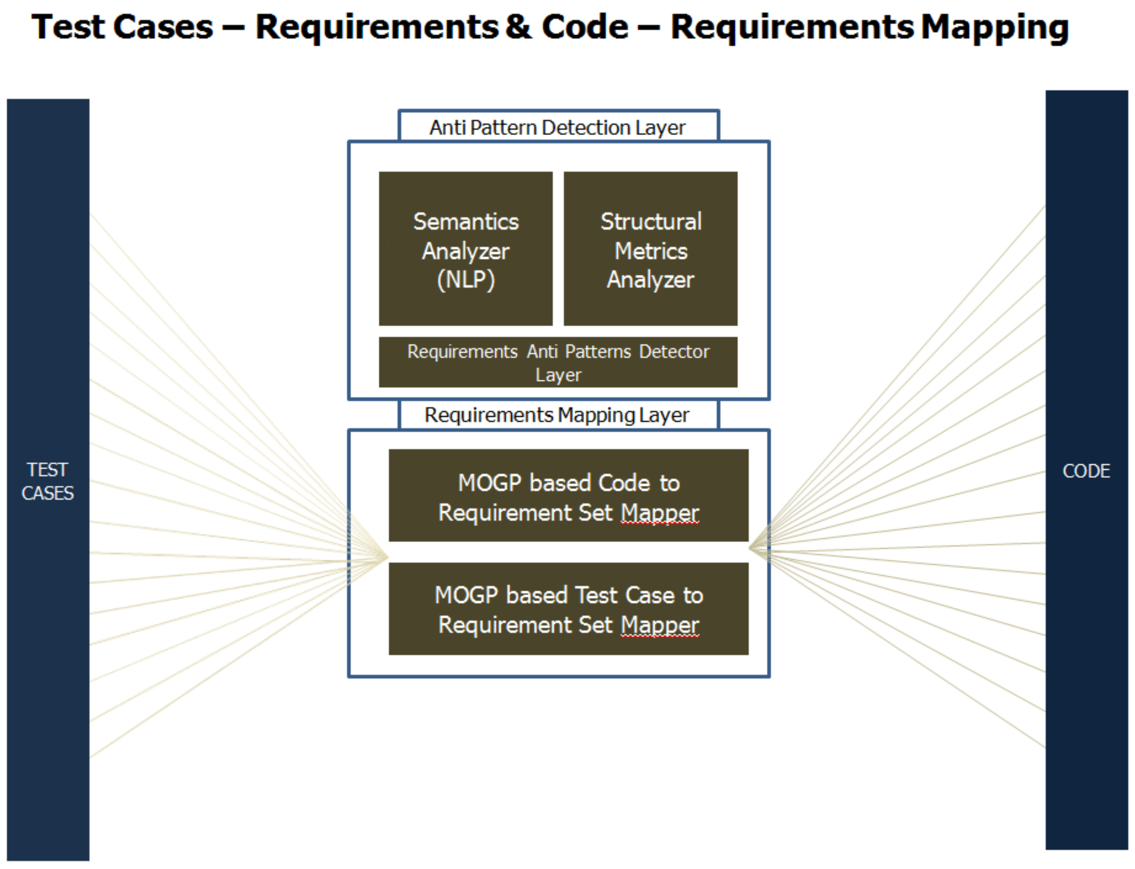

Test Cases – Requirements & Code – Requirements Mapping
TEST CASES
Anti Pattern Detection Layer
Semantics Analyzer (NLP)
Structural Metrics Analyzer
Requirements Anti Patterns Detector Layer
Requirements Mapping Layer
MOGP based Code to Requirement Set Mapper
MOGP based Test Case to Requirement Set Mapper
CODE

Figure 50: Mapping of Test Cases to Software Code using Search Based techniques.

Search based schemes have many potential applications with limitless horizons. Another interesting direction of future work which has potential impact in improving software quality involves software testing. One of the proposed applications include mapping test cases to software code by using semantic and structural analysis techniques employing search based algorithms. With the ability to map validation test cases to code or model level, the software team will be better able to analyze code coverage, defect analysis as well as have the ability to proactively update test cases when a segment of code or a model changes as described in Figure 50.

References

[1] J. Bézivin. On the Unification Power of Models. Software and Systems Modeling, 4(2):171-188, (2005).

[2] M. Brambilla, J. Cabot, M. Wimmer: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, (2012).

[3] R. France, B. Rumpe: Model-driven development of complex software: a research roadmap; Engineering, In: Proceedings of the International Conference on Software Engineering (ICSE'07): Future of Software; IEEE Computer Soceity Press, (2007).

[4] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Statecharts Specifications. In: Proceedings of the International Conference on Software Engineering (ICSE 2007), pp. 54–64. IEEE (2007).

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: Refactoring – Improving the Design of Existing Code; 1st ed. Addison-Wesley, June (1999).

[6] M. Koegel, M. Herrmannsdoerfer, Y. Li, J. Helming, D. Joern: Comparing State- and Operation-based Change Tracking on Models; in: Proceedings of the IEEE International EDOC Conference, (2010).

[7] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer: An Introduction to Model Versioning. In Proceedings of SFM, (2012).

[8] G. Sunyé, D. Pollet, Y. Le Traon, J. M. Jézéquel. Refactoring UML Models. In Proceedings of UML, (2001).

[9] C. Ignat and M. C. Norrie. Operation-based versus State-based Merging in Asynchronous Graphical Collaborative Editing. In Workshop on Collaborative Editing, (2004).

[10] K. Wieland, P. Langer, M. Seidl, M. Wimmer, G. Kappel. Turning Conflicts into Collaboration - Concurrent Modeling in the Early Phases of Software Development. Computer Supported Cooperative Work, 22(2-3):181-240, (2013).

[11] D. Dig, K. Manzoor, R. E. Johnson, T. N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. IEEE Transactions on Software Engineering, 34(3):321-335, (2008).

[12] T. Ekman, U. Asklund. Refactoring-aware Versioning in Eclipse. Electronic Notes in Theoretical Computer Science 107:57-69, (2004).

[13] P. Langer. Adaptable Model Versioning based on Model Transformation By Demonstration. PhD Thesis, Vienna University of Technology, (2011).

[14] W. K. Edwards: Flexible Conflict Detection and Management in Collaborative Applications. In Proceedings of Symposium on User Interface Software and Technology, pages 139-148, (1997).

[15] E. Lippe, N. van Oosterom. Operation-based merging. In Proceedings of SDE, pages 78-87, (1992).

[16] Tom Mens and Tom Tourwé. A Survey of Software Refactoring. IEEE Trans. Softw. Eng. 30, 2, 126-139, (2004).

[17] Mens, T. On the use of graph transformations for model refactoring. In: Generative and Transformational Techniques in Software Engineering pp. 219-257, LNCS 4143, Springer, (2006).

[18] Mens, T., Taentzer, G., & Runge, O. Analyzing Refactoring Dependencies Using Graph Transformation. Journal on Software and Systems Modeling. Springer, Heidelberg, (2007).

[19] D. Arcelli, V. Cortellessa, C. Trubiani: Antipattern-based model refactoring for software performance improvement. QoSA 2012: 33-42, (2012).

[20] D. Dig, C. Comertoglu, D. Marinov, R. Johnson: Automated Detection of Refactorings in Evolving Components; in: Proceedings of the European Conference on Object-Oriented Programming ECOOP'06, Vol. 4067 of LNCS, Springer, pp. 404-428, (2006).

[21] P. Weissgerber, S. Diehl: Identifying Refactorings from Source-Code Changes; in: Proceedings of the International Conference on Automated Software Engineering ASE'06, IEEE, pp. 231-240, (2006).

[22] M. Kim, D. Notkin, D. Grossman, G. Jr. Wilson: Identifying and Summarizing Systematic Code Changes via Rule Inference; IEEE Transactions on Software Engineering, early access article, (2012).

[23] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, W. Schwinger, An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example; in: Proceedings of MoDELS'09, Springer, pp. 271-285, (2009).

[24] S. Vermolen, G. Wachsmuth, E. Visser: Reconstructing complex metamodel evolution; Tech. Rep. TUD-SERG-2011-026, Delft University of Technology (2011).

[25] J. M. Küster, C. Gerth, A. Förster, G. Engels: Detecting and Resolving Process Modeling Differences in the Absence of a Change Log; in: Proceedings of the International Conference on Business Process Management BPM'08, LNCS, Springer, pp. 244, (2008).

[26] T. Kehrer, U. Kelter, G. Taentzer: A rule-based approach to the semantic lifting of model differences in the context of model versioning; in: Proceedings of the International Conference on Automated Software Engineering ASE'11, IEEE, pp. 163, (2011).

[27] Z. Xing, E. Stroulia: Refactoring Detection based on UMLDiff Change-Facts Queries; in: Proceedings of the 13th Working Conference on Reverse Engineering WCRE'06, IEEE, pp. 263-274, (2006).

[28] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige: Different models for model matching: An analysis of approaches to support model differencing. In Proceedings of the Workshop on Comparison and Versioning of Software Models CVSM'09 @ ICSE, (2009).

[29] Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-oriented Design Differencing. In: Proceedings of the International Conference on Automated Software Engineering ASE 2005, pp. 54–65, (2005).

[30] http://pi.informatik.uni-siegen.de/Projekte/sidiff.

[31] Alanen, M., Porres, I.: Difference and Union of Models. In: Proceedings of the International Conference on the Unified Modeling Language (UML'03). LNCS, vol. 2863, pp. 2-17. Springer, (2003).

[32] Rivera, J., Vallecillo, A.: Representing and Operating With Model Differences. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 141–160. Springer (2008).

[33] Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-specific Models. European Journal of Information Systems 16(4), 349–361 (2007).

[34] Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional 9(2), 29–34 (2008).

[35] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdoerfer, M. Seidl, K. Wieland, G. Kappel: A Posteriori Operation Detection in Evolving Software Models; Journal of Systems and Software, (2012).

[36] S. Vermolen, G. Wachsmuth, E. Visser: Reconstructing complex metamodel evolution; Tech. Rep. TUD-SERG-2011-026, Delft University of Technology (2011).

[37] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdoerfer, M. Seidl, K. Wieland, G. Kappel: A Posteriori Operation Detection in Evolving Software Models; Journal of Systems and Software, (2012).

[38] J. Estublier, D. B. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. F. Tichy, D. Wiborg Weber: Impact of software engineering research on the practice of software configuration management. ACM Trans. Softw. Eng. Methodol. 14(4): 383-430 (2005).

[39] R. Conradi, B. Westfechtel: Version Models for Software Configuration Management. ACM Comput. Surv. 30(2): 232-282 (1998).

[40] T. Mens. A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering, 28(5):449-462, (2002).

[41] C. Gerth, J. Malte Küster, M. Luckey, G. Engels: Detection and resolution of conflicting change operations in version management of process models. Software and System Modeling 12(3):517-535 (2013).

[42] S. Barrett, P. Chalin, G. Butler. Table-Driven Detection and Resolution of Operation-Based Merge Conflicts with Mirador. In Proceedings of ECMFA, (2011).

[43] C. Schneider, A. Zündorf, J. Niere. CoObRA - a small step for development tools to collaborative environments. In Workshop on Directions in Software Engineering Environments, (2004).

[44] A. Mougenot, X. Blanc, M. Gervais. D-Praxis: A Peer-to-Peer Collaborative Model Editing Framework. In Proceedings of DAIS, (2009).

[45] M. Alanen, I. Porres: Difference and Union of Models. In: Proceedings of UML 2003: 2-17, (2003).

[46] M. Schmidt, S. Wenzel, T. Kehrer, U. Kelter. History-based Merging of Models. In Workshop on Comparison and Versioning of Software Models, (2009).

[47] Antonio Cicchetti, Davide Di Ruscio, Alfonso Pierantonio: Managing Model Conflicts in Distributed Development. MoDELS 2008: 311-325, (2008).

[48] Chidamber, S.R., Kemerer, C.F. A metrics suite for object-oriented design. IEEE Trans. Software. Eng.20(6), 293–318, (1994).

[49] Chidamber, S.R., Kemerer, C.F. A metrics suite for object-oriented design. IEEE Trans. Software. Eng.20(6), 293–318, (1994).

[50] http://staff.unak.is/andy/StaticAnalysis0809/metrics/overview.html.

[51] Fenton N, Pfleeger SL. Software Metrics: A Rigorous and Practical Approach (2nd edition). London, UK, International Thomson Computer Press, (1998).

[52] Engineering:, McCabe. A Complexity Measure. IEEE Transactions on Software, (1976).

[53] F. Abreu, M. GoulÃ£o and R. Esteves "Toward the Design Quality Evaluation of Object-Oriented Software Systems", Proceedings of 5th ICSQ, (1995).

[54] Da-wei, E. Analysis and Implementation of Software Metric for Object-Oriented. Computational Intelligence and Software Engineering, International Conference on , vol., no., pp.1,4, 11-13, (2009).

[55] M. Mohamed, M. Romdhani, K. Ghédira: Classification of Model Refactoring Approaches. JOT 8(6): 143-158 (2009).

[56] T. Mens, G. Taentzer, D. Müller: Challenges in Model Refactoring. In: Proc. 1st Workshop on Refactoring Tools, University of Berlin (2007).

[57] Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.M.: Refactoring UML Models. In: UML'01. LNCS, vol. 2185, pp. 134–148. Springer (2001).

[58] Boger, M., Sturm, T., Fragemann, P.: Refactoring Browser for UML. In: NetObjectDays'02. LNCS, vol. 2591, pp. 366–377. Springer (2002).

[59] P. Weissgerber, S. Diehl: Identifying Refactorings from Source-Code Changes; in: Proceedings of the International Conference on Automated Software Engineering ASE'06, IEEE, pp. 231-240, (2006).

[60] Zhang, J., Lin, Y., Gray, J.: Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In: Model-driven Software Development—Research and Practice in Software Engineering. pp. 199–217. Springer (2005).

[61] Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update Transformations in the Small with the Epsilon Wizard Language. JOT 6(9), 53–69 (2007).

[62] Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G.,Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: MoDELS'06. LNCS, vol. 4199, pp. 425–439. Springer (2006).

[63] Mens, T.: On the Use of Graph Transformations for Model Refactoring. In: GTTSE'05. LNCS, vol. 4143, pp. 219–257. Springer (2006).

[64] R. B. France, S. Ghosh, E. Song, D.-K. Kim: A Metamodeling Approach to Pattern-Based Model Refactoring. IEEE Software 20(5): 52-58 (2003).

[65] Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. SoSym 7(1), 25–47 (2008).

[66] A. Correa, C. Werner. Applying Refactoring Techniques to UML/OCL Models. Proc. Int'l Conf. UML 2004, LNCS 3273, pp. 173-187, Springer-Verlag, (2004).

[67] W. Sun, R. B. France, I. Ray: Analyzing Behavioral Refactoring of Class Models. ME@MoDELS 2013: 70-79, (2013).

[68] Steimann, F: Constraint-Based Model Refactoring. MoDELS (2011).

[69] J. Reimann, M. Seifert and U. Aßmann Role-based Generic Model Refactoring. In Proc. of MODELS, (2010).

[70] M. Wimmer, N. Moreno, A. Vallecillo: Viewpoint Co-evolution through Coarse-Grained Changes and Coupled Transformations. TOOLS 50: 336-352, (2012).

[71] T. Arendt, G. Taentzer: A tool environment for quality assurance based on the Eclipse Modeling Framework. Automated Software Engineering, Volume 20, Issue 2, page 141-184, (2013).

[72] Eramo, R., Pierantonio, A., Romero, J.R., Vallecillo, A.: Change management in multiviewpoint systems using ASP. In: WODPEC'08. IEEE (2008).

[73] Van Der Straeten, R., Jonckers, V., & Mens, T. Supporting Model Refactorings through Behaviour Inheritance Consistencies, In: UML, pp. 305-319, LNCS 3273, Springer, (2004).

[74] Van Gorp, P., Stenten, H., Mens, T., & Demeyer, S. Towards automating source-consistent UML refactorings, In: UML, pp. 144-158. LNCS 2863, Springer, Heidelberg, (2003).

[75] N. Moha, V. Mahé, O. Barais, J.-M. Jézéquel: Generic Model Refactorings. MoDELS: 628-643, (2009).

[76] Cicchetti, A., Ruscio, D.D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: ICMT'09. LNCS, vol. 5563, pp. 35–51. Springer (2009).

[77] Grundy, J., Hosking, J., Mugridge,W.B.: Inconsistency Management for Multiple-view Software Development Environments. IEEE Trans. Softw. Eng. 24(11), 960–981 (1998).

[78] A. Ghannem, G. El-Boussaidi, M. Kessentini: Model Refactoring Using Interactive Genetic Algorithm. SSBSE (2013).

[79] P. Bottoni, F. Parisi-Presicce, G. Taentzer: Specifying Integrated Refactoring with Distributed Graph Transformations, 220-235, AGTIVE (2003).

[80] J. von Pilgrim, B. Ulke, A. Thies, F. Steimann: Model/code co-refactoring: An MDE approach. ASE: 682-687, (2013).

[81] Saxena, D.K., Duro, J.A., Tiwari, A., Deb, K. and Zhang, Q. Objective Reduction in Many-objective Optimization: Linear and Nonlinear Algorithms. IEEE Transactions on Evolutionary Computation. vol. 17, no. 1. 77–99, (2013).

[82] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput., vol. 6, pp. 182–197, Apr. (2002).

[83] M. Harman. The current state and future of search based software engineering. In Proceedings of ICSE, (2007).

[84] M. Harman, S. A. Mansouri, Y. Zhang. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys 45(1):Article11, (2012).

[85] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M. and da Fonseca, V. G.: Performance assessment of multiobjective optimizers: an analysis and review. IEEE Transaction on Evolutionary Computation. vol. 7, no. 2, 117-132, (2003).

[86] Bechikh, S., Ben Said, L. and Ghédira, K.: Estimating Nadir Point in Multi-objective Optimization using Mobile Reference Points. In Proceedings of the IEEE Congress on Evolutionary Computation CEC'10. 2129–2137, (2010).

[87] John R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA, (1992).

[88] Fowler, Martin, et al.: Refactoring: Improving the design of existing programs. (1999).

[89] Fonseca, C. M. and Fleming, P. J. Genetic algorithms for multi-objective optimization: Formulation, discussion and generalization. In Forrest, S., editor, Proceedings of the Fifth International Conference on Genetic Algorithms, pages 416–423, Morga, (1993).

[90] http://www.eclipse.org/gmf/.

[91] http://ant.apache.org/.

[92] http://argouml.tigris.org.

[93] http://www.jhotdraw.org.

[94] http://xerces.apache.org.

[95] Li, X. A non-dominated sorting particle swarm optimizer for multiobjective optimization. In Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation GECCO'03. 37-48, (2003).

[96] Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-oriented Design Differencing. In: Proceedings of the International Conference on Automated Software Engineering ASE 2005, pp. 54–65. ACM (2005).

[97] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, O. Strollo. When does a Refactoring Induce Bugs? An Empirical Study. In Proceedings of the 12th IEEE WCRE, IEEE Press, (2012).

[98] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer: An Introduction to Model Versioning. In Proceedings of SFM, (2012).

[99] K. Wieland, P. Langer, M. Seidl, M. Wimmer, G. Kappel. Turning Conflicts into Collaboration - Concurrent Modeling in the Early Phases of Software Development. Computer Supported Cooperative Work, 22(2-3):181-240, (2013).

[100] P. Langer. Adaptable Model Versioning based on Model Transformation By Demonstration. PhD Thesis, Vienna University of Technology, (2011).

[101] M. Herrmannsdoerfer, S. Vermolen, G. Wachsmuth: An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In: Proceedings of SLE'10, pp. 163-182, (2010).

[102] S. Elaoud, J. Teghem, T. Loukil: Multiple crossover genetic algorithm for the multiobjective traveling salesman problem. Electronic Notes in Discrete Mathematics 36: 939-946, (2010).

[103] http://www.ganttproject.biz.

[104] http://www.jhotdraw.org.

[105] J. Cohen, “Statistical power analysis for the behavioral sciences,” Routledge, (1988).

[106] C. 1. J. R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press.

[107] M. Kessentini, W. Werda, P. Langer, M. Wimmer: Search-based model merging. In: Proceedings of GECCO 2013: 1453-1460, (2013).

[108] Aiko Fallas Yamashita, Leon Moonen: Do code smells reflect important maintainability aspects? ICSM 2012: 306-315, (2012).

[109] Brown WJ, Malveau RC, Brown WH, Mowbray TJ. Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, (1998).

[110] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, Denys Poshyvanyk, When and Why Your Code Starts to Smell Bad, ICSE (2015).

[111] Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y. G., & Aimeur, E. Smurf: A svm-based incremental anti-pattern detection approach. In Reverse Engineering WCRE, 19th Working Conference on pp. 466-475. IEEE, (2012).

[112] Al Dallal, Jehad. "Identifying refactoring opportunities in object-oriented code: A systematic literature review." Information and Software Technology 58: 231-249, (2015).

[113] Palomba, Fabio, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells." In Software Maintenance and Evolution ICSME, IEEE International Confe, (2014).

[114] Sahin, Dilan, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. "Code-smell detection as a bilevel problem." ACM Transactions on Software Engineering and Methodology TOSEM 24, no. 1: 6, (2014).

[115] Gong M, Jiao L, Du H, Bo L. Multiobjective Immune Algorithm with Nondominated Neighbor-Based Selection. Evolutionary Computation, vol. 6, no. 2, pp. 225-255, (2008).

[116] Kessentini M, Kessentini W, Sahraoui H, Boukadoum M, Ouni A. Design Defects Detection and Correction by Example. In Proceedings of the 19th IEEE International Conference on Program Comprehension ICPC’11, pp. 81-90, (2011).

[117] Kessentini M, Vaucher S, Sahraoui H. Deviance from Perfection is a Better Criterion than Closeness to Evil when Identifying Risky Code. In proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering ASE, pp. 141-151.

[118] Arcuri A, Briand LC. A practical Guide for using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In Proceedings of the 33rd International Conference on Software Engineering ICSE, pp. 1-10, (2011).

[119] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. L. Meur: DECOR: A method for the specification and detection of code and design smells; IEEE Transactions on Software Engineering TSE, (2009).

[120] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao: Facilitating software refactoring with appropriate resolution order of bad smells; in Proceedings of the ESEC/FSE’09, pp. 265–268, (2009).

[121] M. Harman, L. Tratt: Pareto optimal search based refactoring at the design level; in: Proceedings of the Genetic and Evolutionary Computation Conference GECCO'07, pp. 1106-1113, (2007).

[122] Workbook, W. C. Wake: Refactoring; Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (2003).

[123] B. Du Bois, S. Demeyer, J. Verelst: Refactoring—Improving Coupling and Cohesion of Existing Code; in: Proceedings of the 11th Working Conference on Reverse Engineering, WCRE'04, pp. 144-151, (2004).

[124] N. Fenton and S.L. Pfleeger: Software Metrics: A Rigorous and Practical Approach, International Thomson Computer Press, (1997).

[125] H. Ma et al.: Applying OO metrics to assess UML meta-models. In: Proc. of UML, (2004).

[126] J. Cardoso, J. Mendling, G. Neumann, H.A. Reijers: A Discourse on Complexity of Process Models. BPM Workshops, (2006).

[127] G. Sunye et al.: Refactoring UML models. In Proc. of UML, (2001).

[128] www.sbse.us/SQJ/.

[129] Iman Hemati Moghadam, Mel Ó Cinnéide: Automated Refactoring Using Design Differencing. CSMR: 43-52, (2012).

[130] http://www-03.ibm.com/software/products/en/ratirosefami.

[131] Cohen J. Statistical power analysis for the behavioral sciences. Lawrence Erlbaum Associates, Mahwah, NJ, USA, (1988).

[132] Lily Rachmawati, Dipti Srinivasan: Multiobjective Evolutionary Algorithm With Controllable Focus on the Knees of the Pareto Front. IEEE Trans. Evolutionary Computation 13(4): 810-824 (2009).

[133] Dag I. K. Sjøberg, Aiko Fallas Yamashita, Bente Cecilie Dahlum Anda, Audris Mockus, Tore Dybå: Quantifying the Effect of Code Smells on Maintenance Effort. IEEE Trans. Software Eng. 39(8): 1144-1156 (2013).

[134] Harman M, Mansouri SA, Zhang Y. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys, 45, 61 pages, (2012)..

[135] A. Abran and H. Nguyenkim, "Measurement of the Maintenance Process from a Demand- Based Perspective," Journal of Software Maintenance: Research and Practice, pp. 63-90, (1993).

[136] M. Kessentini, W. Werda, P. Langer, M. Wimmer: Search-based model merging. In: Proceedings of GECCO 2013: 1453-1460, (2013).

[137] Beck, Kent, Martin Fowler, and Grandma Beck. "Bad smells in code."Refactoring: Improving the design of existing code: 75-88, (1999).

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

[139] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Engg., 28(1): 4–17, (2002).

[140] 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,” IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20–36, (2010).

[141] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” IEEE Transactions on Software Engineering, vol. 35, no. 3, pp. 347–367, (2009).

[142] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in 20th International Conference on Software Maintenance ICSM 2004, 11-17 September 2004, Chicago, IL, USA. IEEE Computer Society, pp. 350–359, (2004).

[143] D. Arcelli, V. Cortellessa, C. Trubiani: Antipattern-based model refactoring for software performance improvement. QoSA 2012: 33-42, (2012).

[144] Li, X. A non-dominated sorting particle swarm optimizer for multiobjective optimization. In Proceedings of the 2003 International Conference on Genetic and Evolutionary Computation (GECCO'03). 37-48, (2003).