On the Value of Quality Attributes for Refactoring Model Transformations Using a Multi-Objective Algorithm

Bader Alkhazi[a], Chaima Abid[a], Marouane Kessentini[a], Manuel Wimmer[b]

[a] *University of Michigan, USA*
[b] *Johannes Kepler University Linz, CDL-MINT, Austria*

Abstract

Context: Model transformations play a fundamental role in Model-Driven Engineering (MDE) as they are used to manipulate models and to transform them between source and target metamodels. However, model transformation programs lack significant support to maintain good quality which is in contrast to established programming paradigms such as object-oriented programming. In order to improve the quality of model transformations, the majority of existing studies suggest manual support for the developers to execute a number of refactoring types on model transformation programs. Other recent studies aimed to automate the refactoring of model transformation programs, mostly focusing on the ATLAS Transformation Language (ATL), by improving mainly few quality metrics using a number of refactoring types. **Objective:** In this paper, we propose a novel set of quality attributes to evaluate refactored ATL programs based on the hierarchical quality model QMOOD. **Method:** We used the proposed quality attributes to guide the selection of the best refactorings to improve ATL programs using multi-objective search. **Results:** We validate our approach on a comprehensive dataset of model transformations. The statistical analysis of

Email addresses: balkhazi@umich.edu (Bader Alkhazi), cabid@umich.edu (Chaima Abid), marouane@umich.edu (Marouane Kessentini), manuel.wimmer@jku.at (Manuel Wimmer)

our experiments on 30 runs shows that our automated approach recommended useful refactorings based on a benchmark of ATL transformations and compared to random search, mono-objective search formulation, a previous work based on a different formulation of multi-objective search with few quality metrics, and a semi-automated refactoring approach not based on heuristic search. **Conclusion:** All these existing studies did not use our QMOOD adaptation for ATL which confirms the relevance of our quality attributes to guide the search for good refactoring suggestions.

Keywords: Search based software engineering, model transformations, quality attributes, refactoring.

---

1. Introduction

Model-driven engineering (MDE) is a methodology using models as the primary development artifacts [1]. MDE is becoming recently more popular in industry within diverse domains [2, 3]. This approach helps to create high-level abstractions in which they later can be executed or transformed using model transformations [4, 1]. Due to the evolution of languages and metamodels, model transformations–like any regular software–continuously adapt to changes. Therefore, evolving model transformation programs become more complex, less readable, less comprehensible, and less maintainable, leading to a possible increase in the maintenance activities both in time and cost [5]. In fact, most existing model transformation programs are still written in one module containing all the complex transformation rules despite their large number [6, 7].

One of the most popular model transformation languages is the ATLAS Transformation Language (ATL) which is broadly used in both academia and industry [8]. ATL is a hybrid language, extensively used to write model transformation programs. Yet, few studies have proposed refactoring techniques for ATL programs to improve the quality of model transformation. Most of these studies

are mainly proposing a manual selection process to apply refactoring types such as extract rule and merge rule to improve only a few metrics such as Fan-in and Fan-out [9, 10, 11, 12, 13, 14, 15]. However, manual refactoring is error-prone, time-consuming and not scalable [16] which may explain the current low quality of existing model transformations programs [6]. Moreover, the existing object-oriented (OO) refactoring catalogs such as proposed in [17] cannot be reused as they are for model transformation programs [12, 14, 18, 19]. Model transformation programs, such as ATL, are a combination of rules between source and target metamodels. While some of the OO code refactoring types can be reused to refactor models or metamodels [20, 21] such as UML diagrams (e.g., extract class, move method, etc.), the refactoring of model transformations programs require different types of refactorings to deal with the structure of transformation rules such as extract rules, extract helper, merge rules, etc. Moreover, the semantics of rule inheritance and other concepts require a specific treatment in the refactoring process as they may only partially overlap with what is known from OO programming languages [22].

Recently, there are some attempts to automate the refactoring of ATL programs [14, 23] including our MODELS 2016 paper "Automated refactoring of ATL model transformations: a search-based approach" [24]. We proposed an automated approach for refactoring ATL programs that finds a trade-off between four different objectives related to fan-in, fan-out, reducing the number of rules and suggested refactorings. Thus, the search is guided based on those metrics. While the results are promising on refactoring ATL programs, our previous work was still limited to few basic metrics and refactoring types to mainly improve the modularity of ATL programs similar to [23].

In this paper, we are extending our previous work [24] by (i) defining a new quality model for model transformation programs taking inspiration from the

hierarchical quality model QMOOD [25] to consider important quality attributes beyond just the use of coupling and cohesion. We first select the most affected quality attributes by the design of an ATL program before adapting the formula associated with each attribute, following the same model as detailed in [25]; (*ii*) we adapted our multi-objective formulation to consider the new ATL-based quality metrics and refactoring types as detailed in Section 3. To find the optimal trade-off between the various–and possibly conflicting–objectives and to deal with this large search space of possible refactoring solutions, we propose to use a multi-objective formulation based on NSGA-II [26]; (*iii*) we extended our validation with seven case studies from the ATL Zoo [27] to evaluate the performance of our approach. We compared our approach with our previous multi-objective formulation not based on QMOOD [24], and also an existing semi-automated refactoring approach not based on heuristic search [14].

Statistical analysis of our experiments showed that our proposal performed significantly better than random search, our previous multi-objective work not based on QMOOD [24], a mono-objective formulation and [14] with an average precision and recall of 89% and 95% respectively when compared to manual solutions provided by a set of developers. The software developers, who participated in our experiments, confirmed also the relevance of the suggested refactorings as an outcome of a survey study.

The remainder of this paper is structured as follows. Section 2 provides the background and challenges addressed in this paper. Section 3 describes our approach to automated ATL refactoring while the results obtained from our experiments are presented and discussed in Section 4. Threats to validity are discussed in Section 5. Section 6 provides the related work. In Section 7, we summarize and present some ideas for future work.

2. Background, Motivating Example, and Challenges

In this section, we introduce the background for our work, present a motivating example, and discuss the challenges of refactoring ATL transformations.

2.1. Background

ATL transformations are rule-based programs which are executed on fixed input models to produce output models from scratch. For this process, matches in the input model are computed based on the *input patterns* of the transformation rules which trigger the creation of output elements based on the *output patterns* of the transformation rules. In addition, Object Constraint Language (OCL) expressions may be employed for *filter conditions* to restrict the matches in the input model as well as for computing values with so-called *bindings* for setting features of the produced output elements. In essence, two kinds of rules are provided by ATL. First, *matched rules* are rules which are automatically executed by the transformation engine. Second, *lazy rules* and *called rules* have to be explicitly triggered from matched rules similar to calling operations or methods in programming languages. Besides rules, ATL transformations may contain *helpers* which allow reusing OCL expressions to compute values by simply calling the helper definitions. Furthermore, ATL transformations are typed by the input and output metamodels, i.e., the input and output pattern elements have to refer to existing elements in the involved metamodels. An additional feature of ATL is the *module* concept which acts as a container for transformation rules. Thus, each ATL transformation corresponds to a main module which may also compose other modules based on superimposition [28].

2.2. Motivating Example

To further introduce ATL as well as to motivate the need of automatically refactoring ATL transformations, an excerpt of an ATL model transformation

example is shown in Listing 1. The transformation has been extracted from the ATL transformation zoo [27] which is a public repository for collecting ATL transformations frequently used for research purposes [6, 29, 7]. The transformation is, in essence, a simple copy transformation that converts MOF-based metamodels into KM3-based metamodels. The input metamodel excerpt for this transformation excerpt is shown in Figure 1 and the output metamodel excerpt is illustrated in Figure 2. As can be seen in the figures, the input metamodel and the output metamodel have the same class structure and inheritance hierarchy with slight name differences as can be also observed in the ATL transformation shown in Listing 1. The reader can find the details about the ATL language syntax and descriptions in the following references [30, 8]. As can be further seen in the transformation presented in Listing 1, several duplicated bindings for the rules transforming attributes and references are used. The reason for this is simple. The two concepts share many common features which are defined by common superclasses.

Similar to using inheritance between classes in metamodels, ATL also allows to use rule inheritance to introduce abstract rules for defining, for instance, the bindings for setting the features of the *TypedElement* class, namely for setting the *type*, *lower bound*, *upper bound*, and *ordered* features. Furthermore, it can be also observed that the *name* binding is occurring for all three rules in the transformation excerpt which could be also defined for the *ModelElement* class by introducing a top rule for the transformation definition from which all other rules directly or indirectly inherit. Rule inheritance is then used to build a hierarchy of transformation rules whereas the subrules inherit the *input pattern elements* including the *filter conditions* as well as the *output pattern elements* including the *bindings* of the superrules. Listing 2 gives an idea on how rule inheritance may be introduced for the *Attribute* and *Reference* transformation

rules. In particular, the refactoring operations as shown in Table 1 are applied to produce the new transformation design. The refactoring operations are reused from previous work and the full refactoring catalog for ATL can be found in [14] and additional refactorings concerning the module concept of ATL are presented in [23]. The refactorings presented in [14] are classified into renaming, restructuring, inheritance-related, and OCL-related. In the motivating example, we focus on two inheritance-related refactoring operations-–see Table 1.

Table 1: List of considered refactorings for our motivating example based on [14]

Refactoring	**Description**
Extract Superrule	Rules may have several commonalities which should be extracted in one unique definition. The precondition for extracting a superrule is to have common supertypes for the input and output pattern elements of the selected rules. The postcondition is to have a new rule which is becoming the superule for the selected set of rules sharing the commonalities.
Pull Up Binding	A binding which is duplicated in all subrules of a superrule can be pulled up to the superrule in order to eliminate duplicates. The precondition is to have the feature which is computed as well as the features used in the value computation defined as features of the types used in the superrule. The postcondition is to have the binding presented in the superrule and the binding deleted in all subrules.

By using rule inheritance, the binding duplicates can be removed. On the one hand, this has a positive impact on certain design metrics which have been discussed for ATL in [14]. The average number of bindings per rule is reduced as several feature bindings are pushed to the superrules. On the other hand, it has also a negative impact on other design metrics. For instance, the number of rules is increased which may lead to a higher complexity with respect to

Listing 1: Excerpt of the initial Ecore 2 KM3 transformation

```
1  module Ecore2KM3;
2  create OUT : KM3 from IN : MOF;
3
4  rule Class {
5      from i : MOF!EClass
6      to o : KM3!Class (
7          name <- i.name,
8          structuralFeatures <-i.eStructuralFeatures,
9          supertypes <- i.eSuperTypes,
10         isAbstract <- i."abstract"
11     )
12 }
13
14 rule Attribute {
15     from i : MOF!EAttribute
16     to o : KM3!Attribute (
17         name <- i.name,
18         type <- i.eType,
19         lower <- i.lowerBound,
20         upper <- i.upperBound,
21         isOrdered <- i.ordered
22     )
23 }
24
25 rule Reference {
26     from i : MOF!EReference
27     to o : KM3!Reference (
28         name <- i.name,
29         type <- i.eType,
30         lower <- i.lowerBound,
31         upper <- i.upperBound,
32         isOrdered <- i.ordered,
33         opposite <- i.eOpposite,
34         isContainer <- i.containment
35     )
36 }
```

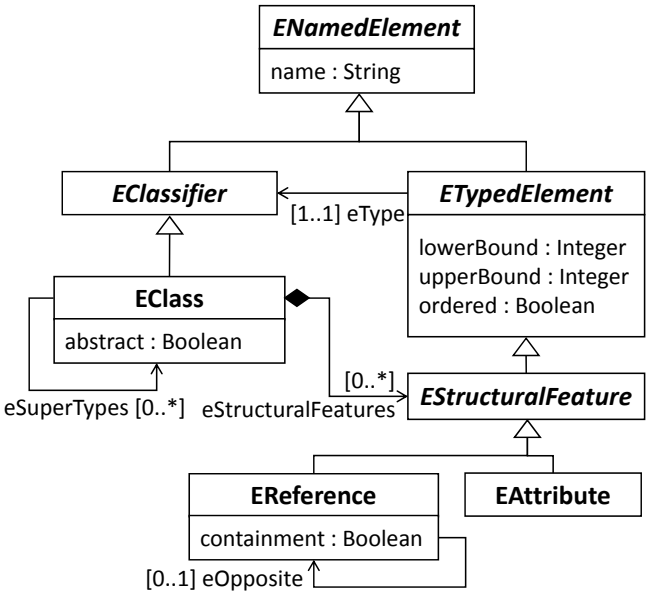

Figure 1: Input metamodel of the transformation example: The Ecore metamodel.

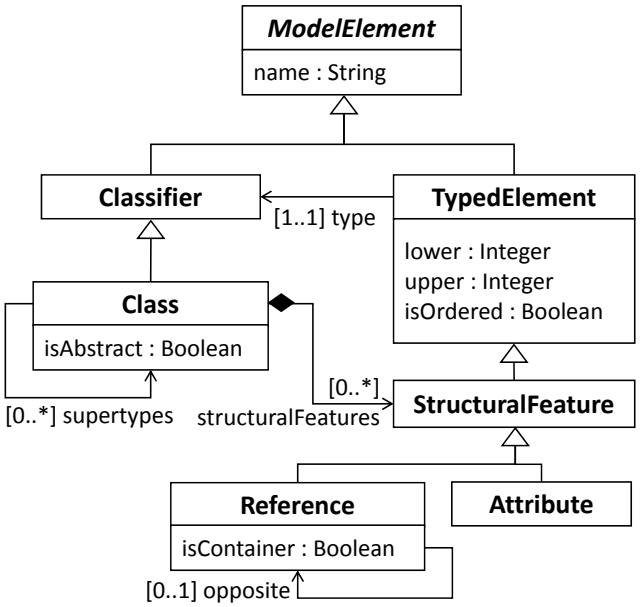

Figure 2: Output metamodel of the transformation example: The KM3 metamodel.

Listing 2: Excerpt of the refactored Ecore 2 KM3 transformation

```
module Ecore2KM3;
create OUT : KM3 from IN : MOF;

abstract rule ModelElement {
    from i : MOF!ENamedElement
    to o : KM3!ModelElement (
        name <- i.name
    )
}

rule Class extends ModelElement {
    from i : MOF!EClass
    to o : KM3!Class (
        structuralFeatures <-i.eStructuralFeatures,
        supertypes <- i.eSuperTypes,
        isAbstract <- i."abstract"
    )
}

abstract rule TypedElement extends ModelElement {
    from i : MOF!ETypedElement
    to o : KM3!TypedElement (
        type <- i.eType,
        lower <- i.lowerBound,
        upper <- i.upperBound,
        isOrdered <- i.ordered
    )
}

rule Attribute extends TypedElement {
    from i : MOF!EAttribute
    to o : KM3!Attribute ()
}

rule Reference extends TypedElement {
    from i : MOF!EReference
    to o : KM3!Reference (
        opposite <- i.eOpposite,
        isContainer <- i.containment
    )
}
```

understanding how a particular rule may be executed by considering the exact rule inheritance semantics of ATL.

The need for improving the quality of ATL-based model transformations has been also highlighted in previous studies which focused on analyzing existing model transformations stored in the largest collection of ATL-based model transformations, the model transformation zoo [6, 7]. In particular, these studies show that model transformations can grow to complex and large artefacts consisting of thousands of lines of code, having hundreds of different transformation rules, and having a low application rate of modularization and abstraction features such as rule inheritance and transformation superimposition although the metamodel structures would allow for such applications [6]. Just to give one example, out of 168 transformations, only 4% use rule inheritance and not a single transformation is using superimposition. One explanation for these results is that these concepts for transformation reuse and modularization have been proposed in later language versions of ATL.

2.3. Challenges

This simple example and previous studies [6, 29, 7] already point out the main challenges of refactoring ATL transformations. Optimizing the different design metrics which have been proposed for ATL [14, 9, 10, 23] may lead to different, potentially conflicting, decisions on how to refactor a particular ATL transformation [25, 31, 32]. Even more challenging, there may not only exist one refactoring solution, but a huge set of possible refactoring solutions that are associated with different design metrics configurations. As ATL transformations may become large containing over 100 rules and several helper definitions [6] as well as a large set of ATL refactoring operations has been proposed [14, 23], the refactoring space of ATL transformations is enormous and enumerative approaches may fail to successfully explore this space efficiently. Therefore, we

propose in the next section a search-based approach to refactor ATL transformations. Before introducing the search-based approach, a comprehensive quality model is required for ATL in order to extend existing work on design metrics for ATL in our search-based framework.

3. Search-Based Refactoring of the Model Transformations

In this section, we start introducing an adaptation of the QMOOD model for model transformations, then we give an overview of our approach, followed by a detailed description of how we formulated the refactoring recommendation process as a multi-objective optimization problem in addition to the multi-objective algorithm's (NSGA-II) adaptation.

3.1. QMOOD for Model Transformations

The quality of a software heavily relies on its design. In software, assessing quality means measuring several conflicting attributes. The quality value, however, depends on multiple factors and circumstances. For instance, what is considered very critical to one developer or designer might be less important for others since people have different preferences when they are designing or implementing a system. For instance, when the requirements of the transformations are not very clear (e.g., some rules need to be added or deleted), the flexibility attribute could be very important. When we are close to the release date, other attributes might be more critical to maintain. Thus, it is useful to somehow be able to quantify the quality of model transformations in order to make it easier for developers to compare and select between multiple refactoring paths. If we know where we stand–in terms of design quality—-then we would be able to make better decisions as to where to move forward and what corrective steps need to be performed to improve the model transformation programs.

In this regard, the authors in [25] linked object-oriented design properties to quality attribute to measure the quality of the software's design formally and validated the QMOOD model empirically on many projects [32]. In this paper, we are adapting their QMOOD approach to assist in computing the quality of model transformations (i.e., ATL). It is important to note that model transformation languages are different from object-oriented programming languages, thus, some design properties and metrics need to be mapped to their closest equivalent counterparts in the context of ATL. In other words, we are using the hierarchical model, QMOOD, as a foundation for the quality attributes computation formulas (Table 2), ATL design metrics (Table 3) and the relationships between them (Table 4).

We did a mapping of the low-level OO metrics into ATL metrics by taking inspiration from existing work [9, 10, 33, 34] while keeping the high-level definitions of the quality attributes. For instance, ATL programs are composed by a set of rules instead of methods and its syntax and semantic reflects that possible analogy between methods and rules. As described later in the validation section, the experiments confirmed our hypothesis and choices during that mapping phase by generating correct and useful refactorings as manually validated by the participants.

Table 2: Computation Formulas for Quality Attributes. [25]

Quality Attribute	**Index Computation Equation**
Reusability	0.415 * Cohesion – 0.085 * Coupling + 0.67 * Design Size
Flexibility	0.583 * Composition – 0.166 * Coupling + 0.583 * Polymorphism
Understandability	0.385 * Cohesion – 0.275 * Abstraction – 0.275 * Coupling – 0.275 * Polymorphism – 0.275 * Complexity – 0.275 * Design Size
Functionality	0.175 * Cohesion + 0.275 * Polymorphism + 0.275 * Design Size + 0.275 * Hierarchies
Extendibility	0.5 * Abstraction – 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.25 * Abstraction + 0.25 * Composition + 0.25 * Inheritance + 0.25 * Polymorphism

Table 3: Design Metrics Description.

ATL Metric	Name	Description
DSM	Design Size in Modules	The count of the total number of modules in the program
NOH	Number of Hierarchies	The count of the number of rule hierarchies
ANA	Average Number of Ancestors	The average number of rules from which a rule inherits information.
DMC	Direct Module Coupling	The count of the number of different modules that a module is directly related to.
CAR	Cohesion Among Rules	The metric computes the relatedness (semantics similarity) among rules of a module.
MOA	Measure of Aggregation	The metric counts the number helpers in ATL programs
MFA	Measure of Functional Abstraction	The metric is the ratio of the number of rules inherited by another rule to the total number of rules accessible by member rules of the module.
NOP	Number of Polymorphic Rules	This metric is a count of the rules that can exhibit polymorphic behavior.
NOR	Number of Rules	Total number of rules defined in a module

Table 4: Relationship Between Design Properties and Design Metrics.

Design Property	Derived Design Metric
Design Size	Design Size in Modules (DSM)
Hierarchies	Number of Hierarchies (NOH)
Abstraction	Average Number of Ancestors (ANA)
Coupling	Direct Module Coupling (DMC)
Cohesion	Cohesion Among Rules (CAR)
Composition	Measure of Aggregation (MOA)
Inheritance	Measure of Functional Abstraction (MFA)
Polymorphism	Number of Polymorphic Rules (NOP)
Complexity	Number of Rules (NOR)

The equations of Table 2 provide explanations on how the different quality attributes are calculated. The details regarding how we are going to use this QMOOD model in practice to improve the quality of model transformation programs, in our automated ATL refactoring endeavors, will be described in the following sub-sections.

3.2. Approach Overview

The approach can be illustrated in the high-level overview shown in Figure 3. An ATL Analyser is applied to the ATL code in order to compute the various design metrics listed in Table 3. These values are used later to measure the quality attributes shown in Table 2, which will eventually be used in the fitness function. The other input of the algorithm is the possible refactoring operations along with their pre- and post-conditions. The main target of the approach is to find the best sequence of refactorings that meets the following optimization objectives: (1) Maximize the quality attributes values (Table 2), (2) minimize the number of rules, and (3) minimize the number of changes.

The objectives mentioned above are not necessarily proportional. In fact, most of them are contrasting with each other. What makes the matters more complicated is the fact that there are multiple refactoring routes. In other words, the order in which we apply the refactoring operations makes a significant difference. Thus, with the substantial number of possible refactorings routes, and the conflicting objectives, we use a multi-objective genetic algorithm (NSGA-II) which will be detailed along with its adaptation to our refactoring problem in the subsequent sections.

3.3. NSGA-II

Most real-world optimization problems encountered in practice involve multiple criteria to be considered simultaneously. These criteria, also called objectives, are often conflicting. Usually, there is no single solution that is optimal

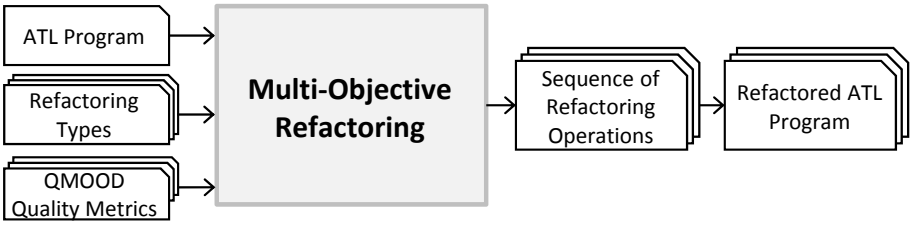

Figure 3: Overview of the multi-objective ATL refactoring approach.

with respect to all these objectives at the same time, but rather many different designs exist which are incomparable per se. Consequently, contrary to Single-objective Optimization Problems (SOPs) where we look for the solution presenting the best performance, the resolution of a Multi-Objective Optimization Problem (MOP) yields a set of compromise solutions presenting the optimal trade-offs between the different objectives. When plotted in the objective space, the set of compromise solutions is called the Pareto front.

The resolution of a MOP yields a set of trade-off solutions, called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front. Hence, the resolution of a MOP consists in approximating the whole Pareto front.

In this paper, we adapted one of the widely used multi-objective algorithms called NSGA-II [26]. It is a powerful search method stimulated by natural selection that is inspired by the theory of Darwin. Hence, the basic idea of NSGA-II is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. As described in Algorithm 1, 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 an initial population R_0 of size N (line 5). Consequently, NSGA-II starts by generating an initial population based on a specific representation that will be discussed later, using an exhaustive list of ATL refactoring types given as input. Thus, this population stands for a set of possible solutions represented as sequences of refactorings that are selected and combined.

To summarize, the main NSGA-II loop goal is to make a population of candidate solutions evolve toward the best sequence of refactoring, i.e., the sequence that minimizes the number of rules, number of recommended refactorings (solutions size) and maximize the quality attributes values. During each iteration t, an offspring population Q_t is generated from a parent population Pt using genetic operators (selection, crossover, and mutation). Then, Q_t and P_t are assembled to create a global population R_t. Then, each solution Si in the population R_t is evaluated using three fitness functions described in the next subsection.

Algorithm 1 NSGA-II overview [26]

1: Create an initial population P_0
2: Generate an offspring population Q_0
3: $t = 0$
4: **while** stopping criteria not reached **do**
5: $R_t = P_t \cup Q_i$
6: F = fast-non-dominated-sort (R_t)
7: $P_{t+1} = \phi$ and i=1;
8: **while** $|P_{t+1}|+|F_i| \leq N$ **do**
9: Apply crowding-distance-assignment(F_i);
10: $P_{t+1} = P_{t+1} \cup F_i$;
11: $i = i + 1$
12: **end**
13: Sort($F_i, < n$)
14: $P_{t+1} = P_{t+1} \cup F_i[1 : (N - |P_{t+1}|)]$;
15: Q_{t+1} = create-new-pop(P_{t+1});
16: $t = t + 1$;
17: **end**

Listing 3: Specification of the Pull Up Binding refactoring

```
refactoring PullUpBinding (b:Binding, sub:Set(Rule), super:Rule)
pre: sub -> forAll(r|r.containsBinding(b) and r.superrule = super) and
    not super.containsBindingForFeature(b.featureToSet) and
    super.outputElements -> exists (oe|oe.type.containsFeature(b.featureToSet
        ↪)) and
    super.inputElements -> exists (ie|b.featuresOfValueComp -> forAll(f|ie.
        ↪type.containsFeature(f))
post: sub -> forAll(r|not r.containsBinding(b)) and super.containsBinding(b)
```

3.4. Search-Based Formulation

3.4.1. Solution representations

A solution consists of a sequence of n refactoring operations involving one or multiple rules/modules of the ATL program to refactor. The vector-based representation is used to define the refactoring sequence. Each vector's dimension has a refactoring operation and its index in the vector indicates the order in which it will be applied. For every refactoring, pre- and post-conditions are specified to ensure the feasibility of the operation as detailed in [14]. In Listing 3, we illustrate the pre- and post-conditions for the refactoring operation *Pull Up Binding* specified in the Object Constraint Language (OCL) . We utilize OCL for the specification of the refactoring operations as OCL provides a general means to specify pre- and post-conditions for operations. More details about the adapted pre- and post-conditions for refactorings can be found in [14].

The initial population is generated by randomly assigning a sequence of refactorings to a randomly chosen set of rules or modules. The different types of refactorings considered in our experiments are *Extract Helper/Rule*, *Inline Helper/Rule*, *Merge Rule*, *Split Rule*, *Extract Superrule*, *Eliminate Superrule*, *Pull Up Binding*, *Pull Up Filter*, *Push Down Binding*, *Push Down Filter* [14], *Extract Module*, *Merge Modules*, and *Move Rule/Helper* [23]. We also use a placeholder concept as an additional possibility to the given refactorings in order to simulate random length of the solution vectors. The placeholder concept is

ExtractSuperrule(„TypedElement“, [Attribute, Reference])	PullUpBinding(type, [Attribute, Reference], TypedElement)	PullUpBinding(type, [Attribute, Reference], TypedElement)

Figure 4: Example of a simplified solution representation.

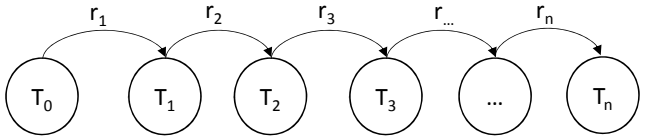

Figure 5: Computing a refactored transformation version (T_n) from its initial version (T_0) by applying a sequence of refactoring operations ($r_1 - r_n$) step-by-step.

equivalent to the no-operation instruction.

The size of a solution, i.e., the vector's length is randomly chosen between upper and lower bound values. The determination of these two bounds is similar to the problem of bloat control in genetic programming where the goal is to identify the tree size limits. Since the number of required refactorings depends mainly on the size and quality of the ATL program, we performed, for each target project, several trial and error experiments using the HyperVolume (HP) performance indicator [26] to determine the upper bound after which, the indicator remains invariant. For the lower bound, it is arbitrarily chosen. The experiments section will specify the upper and lower bounds used in this study.

Figure 4 shows a simplified example of a solution including three refactorings applied to the ATL program described in Listings 1 and 2. The solution includes two refactoring types with the following controlling parameters: *ExtractSuperrule*(name, subrules), *PullUpBinding*(binding, subrules, superrule).

The solution representation is computed by applying the refactoring operations step-by-step on the initial transformation version as it is shown in Figure 5. By following this process, for each refactoring step, the pre- and post-conditions of the refactoring can be check on the predecessor version and successor version of the transformation, respectively.

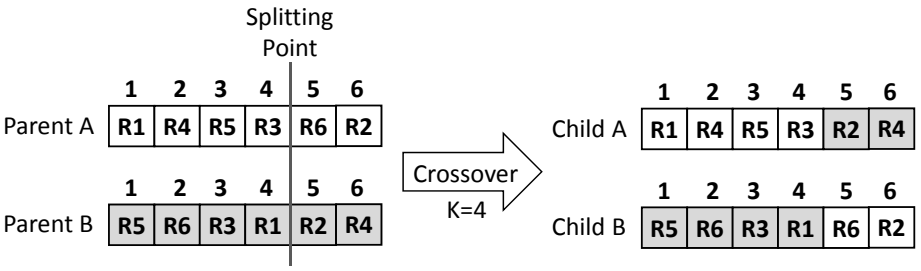

Figure 6: Example of the crossover operation.

Please note that we are only computing an initial population of refactoring solutions which are not representing all possible refactoring solutions (this would be computationally too expensive) but a subset of these which are subsequently further modified by the evolutionary search to compute new, potentially better, solutions.

3.4.2. Solution variation

In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions.

For the crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent solutions. Then, this operator 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 vice versa for the second child. This operator must ensure the respect of the length limits by eliminating randomly some refactoring operations. It is important to note that in multi-objective optimization, it is better to create children that are close to their parents in order to have a more efficient search process. An example of this operation is illustrated in Figure 6.

For mutation, we use the bit-string mutation operator that picks probabilis-

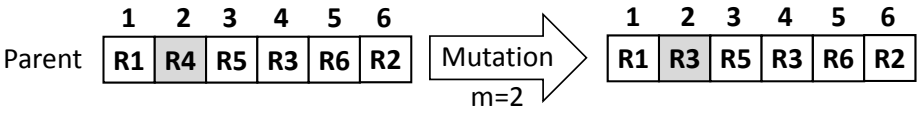

Figure 7: Example of the mutation operation.

tically one or more refactoring operations from its or their associated sequence and replaces them by other ones which are applicable on the current state of the transformation from the list of possible refactorings types as shown in Figure 7.

When applying the changed refactoring sequences, the different pre- and post-conditions of the refactoring operations (such as exemplarily shown in Listing 3) are checked to ensure the applicability of the newly generated refactoring solutions. We also apply a repair operator that randomly computes new refactoring applications to replace existing refactorings which are no longer applicable n the computed refactoring sequence because of invalid pre- and post-conditions.

3.4.3. Solution evaluation

The generated solutions are evaluated using three fitness functions as detailed in the following.

Maximize the quality attributes values: the formulas listed in Table 2 gives us the advantage of calculating the values of the various quality attributes easily. Worth mentioning that we are treating the quality attributes equally in this paper. Whereas in some practical situations, the developer might want to give more weight to one or more attributes depending on the circumstances and the objective of the refactoring operations.

> ***FF1: Max(x)*** *where x is the sum of Reusability, Flexibility, Understandability, Functionality, Extendibility, and Effectiveness.*

Minimize the number of recommended refactorings: The application of a specific suggested refactoring sequence may require an effort that is comparable to that

of re-implementing part of the system from scratch. Taking this observation into account, it is essential to minimize the number of suggested refactorings in the solution since the designer may have some preferences regarding the percentage of deviation with the initial ATL program design. In addition, most developers prefer solutions that minimize the number of changes applied to their design and rules modification. Thus, we formally defined the fitness function as the number of recommended refactorings.

FF2: Min (n) *where n is the number of recommended refactorings.*

Minimize the number of rules: the metric can be easily calculated on ATL programs. The reason to use this metric is to avoid that some refactorings such as split rule or extract rule will generate a high number of new rules when optimizing the remaining objectives.

FF3: Min(r) *where r is the number of rules.*

In fact, the use of multiple quality attributes to guide the search for relevant refactorings may increase dramatically the number of rules such as intensive use of extract rules to improve the extendibility quality attributes.

4. Validation

In order to evaluate the ability of our automated refactoring approach to generate good refactoring recommendations for ATL programs, we conducted a set of experiments based on several transformation programs available in the ATL Zoo [27]. Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. Thus, we executed our the search algorithms 30 times on each of the 7 ATL transformation programs, and the obtained precision and recall results are subsequently statistically analyzed.

In the following, we first present our research questions and the pilot study, and then describe and discuss the obtained results.

4.1. Research Questions

We defined five research questions that address the applicability, performance, and usefulness of our refactoring approach. The five research questions (RQs) are as follows:

RQ1: Search validation (sanity check). To validate the problem formulation of our approach, we compared our multi-objective formulation with a random search algorithm (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 improve the quality of ATL programs using the combination of multi-objective search and QMOOD? In particular, is it possible with our approach to find refactoring solutions which improve the quality of the transformations and are relevant for transformation developers.

RQ3: How does our multi-objective QMOOD-based refactoring approach for ATL programs perform compared to our previous multi-objective refactoring work published in MODELS 2016 [24] and a mono-objective approach aggregating all the three objectives used in this paper? A multi-objective algorithm provides a trade-off between the four objectives where developers can select their desired refactoring solution from the Pareto-optimal front. A mono-objective approach uses a single fitness function that is formed as an aggregation of the four normalized objectives and generates as output only one refactoring solution. This comparison is required to ensure that the solutions provided by NSGA-II provide a better trade-off between the three objectives than a mono-objective approach. Otherwise, there is no benefit to our multi-objective adaptation. Furthermore, it is important to compare the

performance of our new multi-objective QMOOD formulation to our previous multi-objective work of MODELS2016 [24] to evaluate the relevance of considering new quality attributes on the relevance of refactoring recommendations.

RQ4: How does the proposed multi-objective QMOOD-based refactoring approach for ATL programs perform compared to an existing semi-automated ATL refactoring approach [14] not based on heuristic search? While it is interesting to show that maybe our proposal outperforms random search or a mono-objective refactoring approaches, developers will consider our approach useful, if it can outperform other existing tools that are not based on optimization techniques. Thus, we compared our approach to the semi-automated refactoring approach proposed in [14]. In this approach, the developers first have to manually select the required refactoring operations and their application points. Subsequently, the refactoring execution, i.e., rewriting the transformation, is automatically performed.

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

RQ5 (Insight): Can our ATL refactoring approach be useful for software developers in practice? We conducted a post-study questionnaire with the subjects of our experiments that collects their opinions of our tool.

4.2. Case Studies

Our research questions are evaluated using the following seven case studies. Each case study consists of one model transformation and all the necessary artifacts to execute the transformation, i.e., the input and output metamodels and a sample input model. Most of the case studies have been taken from the ATL Zoo [27], a repository where developers can upload and describe their ATL transformations. We briefly describe in the following the different ATL transformation programs used in our study.

Ecore2Maude: This transformation takes an Ecore metamodel as input and generates a Maude specification. Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications.

OCL2R2ML: This transformation takes OCL models as input and produces R2ML (REWERSE I1 Markup Language) models as output.

R2ML2RDM: This transformation is part of the sequence of transformations to convert OCL models into SWRL (Semantic Web Rule Language) rules. In this process, the selected transformation takes a R2ML model and obtains an RDM model that represents the abstract syntax for the SWRL language.

XHTML2XML: This transformation receives XHTML models conforming to the XHTML language specification version 1.1 as input and converts them into XML models consisting of elements and attributes.

XML2Ant: This transformation is the first step to convert Ant to Maven. It acts as an injector to obtain an XMI file corresponding to the Ant metamodel from an XML file.

XML2KML: This transformation is the main part of the KML (Keyhole Markup Language) injector, i.e., the transformation from a KML file to a KML model. Before running the transformation, the KML file is renamed to XML and the KML tag is deleted. KML is an XML notation for expressing geographic annotation and visualization within Internet-based, two-dimensional maps and three-dimensional Earth browsers.

XML2MySQL: This transformation is the first step of the MySQL to KM3 transformation scenario, which translates XML representations used to encode the structure of domain models into actual MySQL representations.

We have selected these case studies due to their difference in size, structure and number of dependencies among their transformation artifacts, i.e., rules

and helpers. Table 5 summarizes the number of rules, the number of helpers and the number of dependencies between rules.

To answer RQ1, RQ2, RQ3, and RQ4, it is important to validate the proposed refactoring solutions. We evaluated the improvements in the different quality metrics used by our approach before and after refactorings. Since the metrics improvement evaluation is not sufficient, we asked a group of developers, as detailed later, to manually identify several refactoring opportunities and apply several refactorings to fix the detected possible quality improvements on the selected transformation programs. Table 5 summarizes the number of expected refactorings for every ATL program. Then, we calculated precision and recall scores to compare between refactorings recommended by our approach and those suggested manually by the subjects:

$$RC_{Recall} = \frac{\text{suggested operations} \cap \text{expected operations}}{\text{expected operations}} \in [0,1] \qquad (1)$$

$$PR_{Precision} = \frac{\text{suggested operations} \cap \text{expected operations}}{\text{suggested operations}} \in [0,1] \qquad (2)$$

We also asked the group of potential users of our tool to evaluate, manually, whether the suggested refactorings are feasible and efficient at improving the ATL program quality and achieving their maintainability objectives. We define the metric Manual Correctness (MC) to mean the number of meaningful/relevant refactorings divided by the total number of suggested refactorings. MC is given by the following equation:

$$MC = \frac{\#\text{coherent applied refactorings}}{\#\text{proposed refactorings}} \qquad (3)$$

To avoid the computation of the MC metric being biased by the developer's

feedback, we asked the developers to manually evaluate the correctness of the recommended refactorings on the ATL programs that they did not refactor using our tool.

To answer the first research question RQ1, a random multi-objective algorithm was implemented where at each iteration the population is randomly created without the use of change operators. The random search used the same fitness functions of our QMOOD formulation but without the use of the change operators. The obtained best refactoring solution was compared for statistically significant differences with NSGA-II using PR, RC, MC and the execution time (CT). To answer RQ2, we evaluate the results of our NSGA-II algorithm using all the above evaluation metrics. To answer RQ3, we compared our approach to a mono-objective Genetic Algorithm where all the objectives were normalized in the range [0..1] and aggregated into one objective to minimize. To answer RQ4, we compare NSGA-II to an existing semi-automated ATL refactoring approach [14] where the refactoring operations have to be explicitly triggered by the user and only the execution of the manually identified refactorings is automated. We used all the above evaluation metrics to perform the comparisons in RQ3 and RQ4 as well.

Table 5: Statistics of the Case Studies.

Case Study	**#Rules**	**#Helpers**	**#Dependencies**	**#Expected Refactorings**
Ecore2Maude	40	40	27	19
OCL2R2ML	37	11	54	16
R2ML2RDM	58	31	137	22
XHTML2XML	31	0	59	18
XML2Ant	29	7	28	16
XML2KML	84	5	0	37
XML2MySQL	6	10	5	9

Our study involved 27 participants from the University of Michigan. Participants include 21 master students and 8 Ph.D. students in Software Engineering.

All the participants are volunteers and familiar with ATL and model transformations. All the graduate students have a strong background in refactoring and software quality since they all took a graduate course on Software Quality Assurance extensively covering these topics. The experience of these participants on programming and refactoring ranged from 2 to 16 years in industry. Eleven out the twenty-seven participants are active programmers in software companies.

To answer RQ5, we used a post-study questionnaire that collects the opinions of developers on our tool. Participants were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their programming experience, their familiarity with software refactoring and ATL. In addition, all the participants attended one lecture about ATL and software refactoring, and passed six tests to evaluate their performance to evaluate and suggest refactoring solutions for ATL programs.

Each participant in the study received a questionnaire, a manuscript guide to help them to fill the questionnaire, the tools and results to evaluate, and the ATL source code of the studied transformations. Since the application of refactoring solutions is a subjective process, it is normal that not all the developers have the same opinion. In our case, we considered the majority of votes to determine if suggested solutions are correct or not. Each participant evaluates different refactoring solutions for the different techniques and systems. We conducted this manual validation since a correct refactoring in terms of behavior preservation may not mean that programmers will apply it. The refactoring process is subjective and it is impossible to validate the recommendations without human studies to see if programmers find these refactorings relevant or not. For each program, the participants evaluated the ATL programs before and

after refactorings to estimate the benefits of the refactoring and not only the correctness that is checked with the pre/post conditions. We consider that a sequence of refacorings has benefits if it is accepted by the programmers during the manual validation process.

We asked every participant to manually suggest and apply refactorings to improve the quality of the ATL programs. As an outcome of this first scenario, we calculated the differences between the recommended refactorings and the expected ones (manually suggested by the developers). In the second scenario, we asked the developers to manually evaluate the best recommended solution by our algorithm and the remaining techniques. We performed a cross-validation between the participants to avoid the computation of the MC metric being biased by their manual recommendations. In the third scenario, we asked the participants to use our tool during a period of two hours on the different programs and then we collected their opinions based on a post-study questionnaire that will be detailed later. The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study.

For each case study and algorithm, we select one solution using a knee point strategy [35]. The knee point corresponds to the solution with the maximal trade-off between all fitness functions, i.e., a vector of the best objective values for all solutions. In order to find the maximal trade-off, we use the trade-off worthiness metric proposed by [35] to evaluate the worthiness of each solution in terms of objective value compromise. The solution nearest to the knee point is then selected and manually inspected by the subjects to find the differences with an expected solution. While the knee point selection may not be the perfect way, it is the only strategy to ensure a fair comparison with the mono-objective and deterministic approaches since they generate only one solution (sequence of

refactorings) as output. Subjects were aware that they are going to evaluate the quality of our solutions, but were not told from which algorithms the produced solutions originate.

4.3. Experimental Setting

Parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each algorithm and each ATL program, we perform a set of experiments using several population sizes: 50, 100, 200, 300 and 500. The stopping criterion was set to 100,000 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: crossover probability = 0.7; mutation probability = 0.4 where the probability of gene modification is 0.2. Each algorithm is executed 30 times with each configuration and then the comparison between the configurations is done using the Wilcoxon test. The upper and lower bounds on the chromosome length used in this study are set to 10 and 50 respectively.

We note that it is not required to make 30 runs to find the knee-point. The 30 runs are executed for the automated evaluation metrics (e.g., precision and recall) to make sure that the results are statistically significant due to the randomness involved in computational search algorithms. For the manual correctness, we just selected the median execution for the comparison with the remaining algorithms. Thus, it is not required that the user should run the multi-objective algorithm 30 times to obtain the best solutions since we just did these runs only for the statistical tests purpose of these experiments.

4.4. Statistical Test Methods

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 based on 30 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 ($\alpha = 5\%$). The latter tests the null hypothesis, H_0, that the obtained results of two algorithms are samples from continuous distributions with equal medians, 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 obtained by comparing the results of the different algorithms with our approach. In this way, we determine whether the performance difference between our technique and one of the other approaches is statistically significant or just a random result. The results presented were found to be statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 95% confidence level ($\alpha < 5\%$). The Wilcoxon rank sum test verifies whether the results are statistically different or not; however, it does not give any idea about the difference in magnitude. Thus, we used the Vargha-Delaney A measure which is a non-parametric effect size measure. In our context, given the different performance metrics (e.g., PR and RC), the A statistic measures the probability that running an algorithm B1 (NSGA-II based on QMODD) yields better performance than running another algorithm B2 (such as RS, Mono-objective GA, etc.). If the two algorithms are equivalent, then A = 0.5. More details can be found in the statistical tests guildline discussed in [36].

As described in Table 6, we have found the following results: a) on small scale ATL programs (XML2MySQL, XHTML2XML, and XML2Ant) our approach is better than all the other algorithms based on all the performance metrics with an

A effect size higher than 0.92; and b) on medium and large scale ATL programs (XML2KML, Ecore2Maude, OCL2R2ML, and R2ML2RDM), our approach is better than all the other algorithms with an A effect size higher than 0.89.

Table 6: Statistical test results.

Comparison	**Systems**	**p-value**		**VDA**	
		Precision	*Recall*	*Precision*	*Recall*
QMOOD/NSGA-II vs NSGA-II	Ecore2Maude	< 0.01	< 0.01	0.89	0.9
	OCL2R2ML	0.021	< 0.01	0.89	0.91
	R2ML2RDM	< 0.01	< 0.01	0.89	0.9
	XHTML2XML	< 0.01	0.032	0.93	0.91
	XML2Ant	< 0.01	< 0.01	0.94	0.93
	XML2KML	0.017	< 0.01	0.9	0.89
	XML2MySQL	0.026	< 0.01	0.95	0.96
QMOOD/NSGA-II vs RS	Ecore2Maude	< 0.01	< 0.01	0.89	0.92
	OCL2R2ML	< 0.01	< 0.01	0.91	0.9
	R2ML2RDM	< 0.01	< 0.01	0.92	0.91
	XHTML2XML	0.028	< 0.01	0.93	0.94
	XML2Ant	< 0.01	< 0.01	0.95	0.93
	XML2KML	< 0.01	< 0.01	0.9	0.91
	XML2MySQL	< 0.01	0.034	0.93	0.96
QMOOD/NSGA-II vs GA	Ecore2Maude	< 0.01	< 0.01	0.89	0.9
	OCL2R2ML	< 0.01	< 0.01	0.89	0.91
	R2ML2RDM	< 0.01	< 0.01	0.89	0.92
	XHTML2XML	< 0.01	< 0.01	0.96	0.94
	XML2Ant	< 0.01	< 0.01	0.98	0.94
	XML2KML	< 0.01	< 0.01	0.89	0.91
	XML2MySQL	0.024	< 0.01	0.97	0.94
QMOOD/NSGA-II vs Wimmer et al.	Ecore2Maude	< 0.01	< 0.01	0.9	0.89
	OCL2R2ML	< 0.01	0.023	0.89	0.91
	R2ML2RDM	< 0.01	< 0.01	0.92	0.9
	XHTML2XML	0.033	< 0.01	0.96	0.98
	XML2Ant	< 0.01	< 0.01	0.94	0.97
	XML2KML	< 0.01	0.027	0.9	0.89
	XML2MySQL	< 0.01	< 0.01	0.93	0.96

4.5. Results and Discussions

Results for RQ1: The results for the first research questions are summarized in Figures 8, 9 and 10. It is clear that QMOOD-NSGA-II is better than random search based on the different metrics of PR, RC and MC on all the 7 ATL case studies. The average precision, recall and manual correctness values of random search on the different ATL programs are lower than 28%. This can be

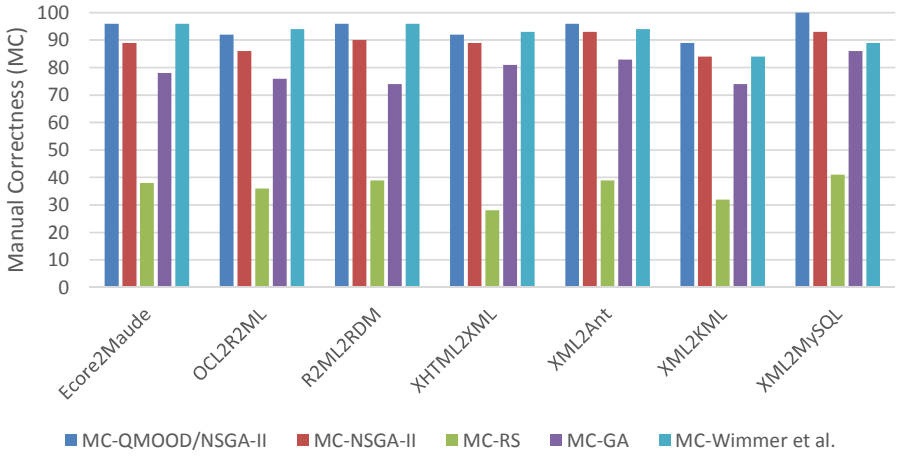

Figure 8: Median Manual Correctness (MC) over 30 runs on all the 7 ATL programs using the different ATL refactoring techniques.

explained by the huge search space to explore to generate relevant refactorings. Figure 11 shows that the execution time (CT) of random search is lower than QMOOD-NSGA-II however the difference is just limited to an average of 15 minutes. Furthermore, ATL refactoring is not requiring strict time constraints unlike real-time application which is not the case here. We do not dwell long in answering the first research question, RQ1, which involves comparing our approach based on QMOOD-NSGA-II with random search. The remaining research questions will reveal more about the performance, insight, and usefulness of our approach. We conclude that there is empirical evidence that our multi-objective formulation based on QMOOD surpasses the performance of random search thus our formulation is adequate (this answers RQ1).

Results for RQ2: As reported in Figure 8, the majority of the refactoring solutions recommended by our multi-objective approach were correct and approved by developers. On average, for all of our seven studied projects, 94% of the proposed ATL refactoring operations are considered as feasible, improve the quality and are found to be useful by the software developers of our experiments.

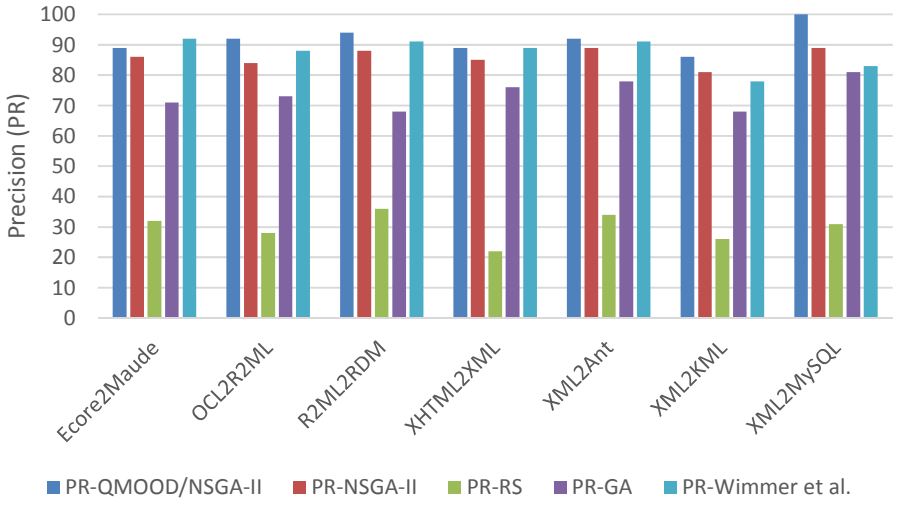

Figure 9: Median Precision (PR) over 30 runs on all the 7 ATL programs using the different ATL refactoring techniques with a 95% confidence level ($\alpha < 5\%$).

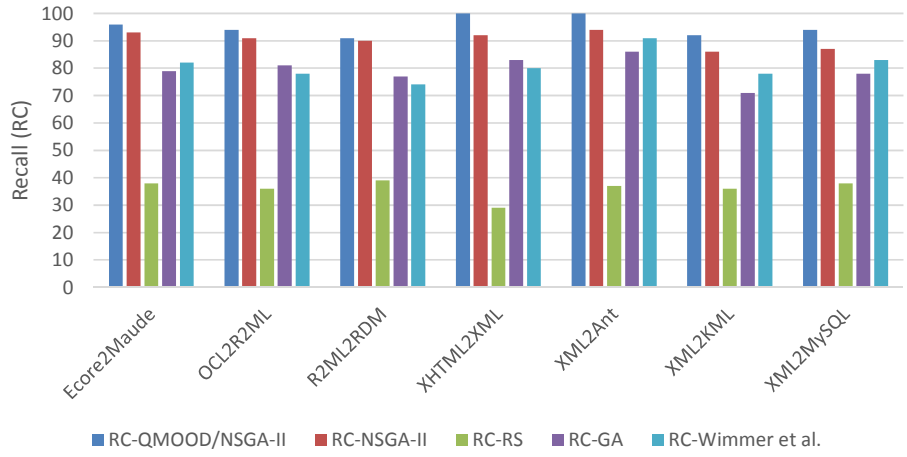

Figure 10: Median Recall (RC) over 30 runs on all the 7 ATL programs using the different ATL refactoring techniques with a 95% confidence level ($\alpha < 5\%$).

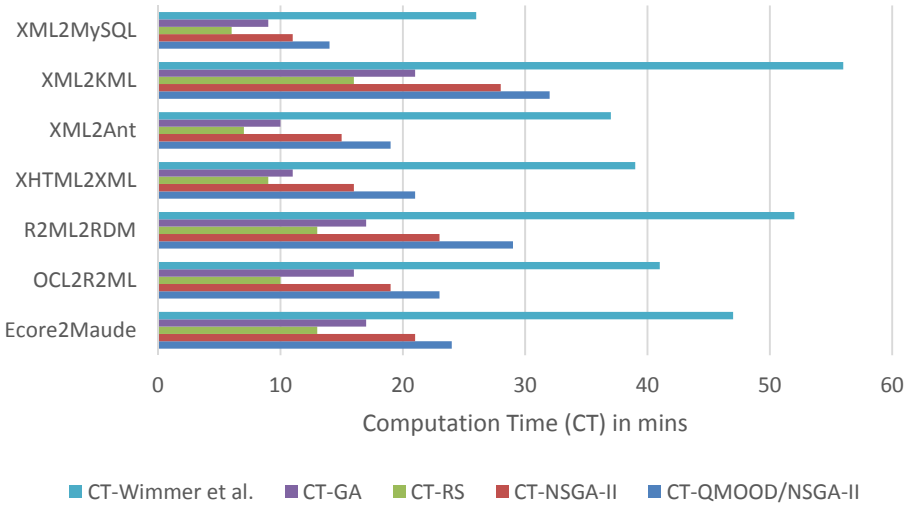

Figure 11: Median computation time (CT) in minutes over 30 runs on all the 7 ATL programs using the different ATL refactoring techniques.

The highest MC score is 100% for the XML2MySQL program and the lowest score is 89% for XML2KML program. Thus, it is clear that the results are independent of the size of the ATL programs and the number of recommended refactorings. Most of the refactorings that were not manually approved by the developers were found to be either violating some post-conditions or introducing design incoherence.

Since the MC metric just evaluates the correctness and not the relevance of the recommended refactorings, we also compared the proposed operations with some expected ones defined manually by the different participants for several ATL code fragments extracted from the seven programs. Figure 9 and Figure 10 summarize our findings. We found that a considerable number of proposed refactorings, with an average of more than 91% in terms of precision and 96% of recall, were already applied by the software development team and suggested manually (expected refactorings). The recall scores are higher than precision ones since we found that the refactorings suggested manually by developers are

incomplete compared to the solutions provided by our automated approach and this was confirmed by the manual evaluation (MC). In addition, we found that the slight deviation with the expected refactorings is not related to incorrect operations but to the fact that different refactoring strategies are equivalent in terms of quality even if the applied refactoring types are different. Furthermore, the use of the fitness function to minimize the number of refactorings may help to reduce the noise in the recommended solutions and focus mainly on the refactorings which improved the quality metrics.

We decided to evaluate the performance of our approach using evaluation metrics different than the fitness functions (quality metrics) to ensure a fair comparison with existing techniques as detailed in the next research questions. To summarize and answer RQ2, the experimentation results confirm that our QMOOD based multi-objective approach helps the participants to refactor their ATL programs efficiently by finding the relevant refactorings and improve the quality of all the five programs.

Results for RQ3: Figures 8, 9 and 10 confirm the average superior performance of our QMOOD multi-objective approach compared to our previous work based on NSGA-II (and limited to coupling and cohesion) [24] and a mono-objective GA aggregating all the objectives in an equal way. Figure 8 shows that our approach provides significantly higher manual correctness results (MC) than NSGA-II (as used in [24]), a mono-objective formulation having MC scores between 89% and 79% on the different ATL programs. The same observation is valid for the precision and recall as described in Figures 9 and 10. Thus, it is clear that all the different objectives considered in our formulation are conflicting justifying the outperformance of NSGA-II whether based on QMOOD or not. Furthermore, the results confirm that the QMOOD metrics formulation is more aligned with the preferences of ATL developers than the limited use of

fan-in and fan-out.

Since our proposal is based on multi-objective optimization, it is important to evaluate the execution time (CT). It is evident that both NSGA-II adaptations require higher execution time than RS and GA since NSGA-II is considering a higher number of objectives and change operators. In addition, the use of QMOOD metrics made the execution time slower comparing to our previous multi-objective work. All the search-based algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 8 GB RAM. Overall, RS, GA and NSGA-II algorithms were faster than QMOOD-NSGA-II. In fact, the average execution time for QMOOD-NSGA-II NSGA-II, GA and RS were respectively 23, 19, 15 and 10 minutes. However, the execution for QMOOD-NSGA-II is reasonable because the algorithm is not executed daily by the developers and the refactoring of ATL programs is not a real-time problem.

To conclude, our QMOOD multi-objective approach provides better results, on average, than our previous multi-objective work, a mono-objective refactoring algorithm aggregating the different objectives (answer to RQ3).

Results for RQ4: Since it is not sufficient to compare our proposal with only search-based work, we compared the performance of QMOOD-NSGA-II with the semi-automated refactoring approach proposed in [14]. Figures 8, 9 and 10 summarizes the results of the precision, recall and manual correctness obtained on the 7 ATL programs. The precision of the semi-automated refactoring approach is slightly lower than NSGA-II in all the programs on an average of 90%; however, the precision scores are lower than our proposal on all the programs. The manual precision of both approaches is comparable and almost the same.

In fact, the good precision achieved by the semi-automated approach can be easily explained by the fact that the refactorings are manually detected by the

programmers but just automatically executed. In addition, the recall is lower than QMOOD-NSGA-II because it is time-consuming for the programmers to identify a large set of relevant refactorings which is automatically generated using QMOOD-NSGA-II. It is also clear that the semi-automated refactoring approach [14] is time consuming with an average of more than 45 minutes however our QMOOD-NSGA-II algorithms can recommend and apply relevant refactorings in a time frame lower than 25 minutes as described in Figure 11. To conclude, our QMOOD-NSGA-III adaption also outperforms, on average, an existing semi-automated approach not based on meta-heuristic search (RQ4).

Results for RQ5: We have asked the participants to take a post-study questionnaire after completing the refactoring tasks using our multi-objective refactoring tool and all the techniques considered in our experiments. The post-study questionnaires collected the opinions of the participants about their experience in using our approach compared also to the semi-automated refactoring tool [14] and our previous multi-objective work not based on QMOOD [24]. The post-study questionnaire asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements:

- The automated refactoring recommendations are a desirable feature in ATL.
- The multi-objective automated manner of recommending refactorings by our approach is a useful and flexible way to refactor ATL model transformation programs compared to semi-automated or manual refactorings.
- The use of QMOOD quality attributes is relevant to improve the quality of ATL programs.

The agreement of the participants was 4.8, 4.4 and 4.8 for the three state-

ments, respectively. This confirms the usefulness of our approach for the software developers considered in our experiments. The remaining questions of the post-study questionnaire were about the benefits and also limitations (possible improvements) of our multi-objective approach. In addition, the questionnaire confirms that the developers found the use of QMOOD attributes is relevant to identify refactoring opportunities for model transformation programs. We summarize in the following the feedback of the developers. Most of the participants mention that our automated approach is faster than semi-automated or manual refactoring since they spent a long time with these techniques to find the locations where refactorings should be applied and which ones to select. For example, developers spend time when they decide to extract a rule to find the elements to move to the newly created rule. Thus, the developers liked the functionality of our tool that helps them to automatically recommend the refactorings and finding quickly the right controlling parameters based on the recommendations. Furthermore, refactorings may affect several locations in the ATL source code, which is a time-consuming task to perform manually, but they can perform it instantly using our tool.

Another important feature that the participants mentioned is that our approach allows them to take the advantages of using multi-objective optimization for ATL refactoring without the need to learn anything about optimization and exploring explicitly the Pareto front to select one "ideal" solution. The implicit exploration of the Pareto front using the Knee point strategy represents an important advantage of our tool.

The participants also suggested some possible improvements to our multi-objective ATL refactoring approach. Some participants believe that it will be very helpful to extend the tool by adding a new feature to apply automatically some regression testing techniques on ATL programs to generate test cases to

test applied refactorings. Another possibly suggested improvement is to use some visualization techniques to evaluate the impact of applying a refactoring sequence. In addition, they did not appreciate sometimes the long list of refactoring suggested by our tool since they want to take control of modifying and rejecting some refactorings. In addition, the validation of this long list of refactorings is time-consuming. Finally, the developers also highlighted that it will be interesting to consider the quality attributes of QMOOD with different weights since they may not be equally important.

5. Threats to Validity

Following the methodology proposed by [37], there are four types of threats that can affect the validity of our experiments. We consider each of these in the following paragraphs.

5.1. Conclusion Validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We addressed conclusion threats to validity by performing 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Wilcoxon rank sum test with a 95% confidence level ($\alpha = 5\%$). 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. The parameters' values used in our experiments are found by trial-and-error. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance. In addition, our multi-objective formulation treats the different types of refactoring with the same weight in terms of complexity

when calculating one of the fitness functions. However, some refactoring types can be more complex than others to apply by developers.

5.2. Internal Validity

Internal validity is concerned with the causal relationship between the treatment and the outcome. We dealt with internal threats to validity by performing 30 independent simulation runs for each problem instance. This makes it highly unlikely that the observed results were caused by anything other than the applied multi-objective approach. Another potential threat is related to the users study that is performed mainly by students. We mitigate this threat by selecting students who are active programmers in industry. They all have a strong background in refactoring and software quality as they all took a graduate course in software quality assurance. We also made sure that they understand ATL concepts by giving them a lecture and applications about it. All the participants took a test to evaluate a set of 5 refactorings before performing the experiments.

5.3. Construct Validity

Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected solutions at the knee point when we compared our approach with the mono-objective GA and random search, but the developers may select a different solution based on their preferences to give different weights to the objectives when selecting the best refactoring solution. The different developers involved in our experiments may have divergent opinions about the recommended refactorings in terms of correctness and readability. We considered in our experiments the majority of votes from the developers. For the selection threat, the participant diversity in terms of experience could affect the results of our study. We addressed the selection threat by giving a lecture and examples of ATL refactorings already

evaluated with arguments and justification. For the fatigue threat, we did not limit the time to fill the questionnaire and we also sent the questionnaires to the participants by email and gave them the required time to complete each of the required tasks.

Another construct threat is related to our choices when formulating the QMOOD model for model transformations. To mitigate this threat, we did a mapping of the low-level OO metrics into ATL metrics and we tried to keep the high-level definitions of the quality attributes. For instance, ATL programs are composed by a set of rules instead of methods and its syntax and semantic reflects that possible analogy between methods and rules. The experiments confirmed our hypothesis and choices during that mapping phase by generating correct and useful refactorings as manually validated by the participants. However, it is possible that some of the quality attributes may not reflect the developers preference due to subjective nature of refactoring.

5.4. External Validity

External validity refers to the generalizability of our findings. In this study, we performed our experiments on seven different ATL programs belonging to different domains and having different sizes. However, we cannot assert that our results can be generalized to other programs, and other practitioners. Future replications of this study are necessary to confirm our findings. In addition, our study was limited to the use of specific refactoring types and ATL metrics. Future replications of this study are necessary to confirm our findings, e.g., if the general approach is also applicable for OCL-related refactorings [38, 39, 40, 41].

6. Related Work

Regarding the contribution of this paper, we discuss the main three threads of related work. First, we discuss the different kinds of work regarding the

evaluation of the quality of model transformations. Second, we discuss the use of search-based software engineering to solve model-driven engineering problems. Finally, we discuss work specifically dealing with the refactoring of model transformations.

6.1. Quality of Model Transformations

Of course, there is substantial work regarding the quality of software, thus, we will only discuss the most closely related work especially those focusing on the quality of model transformations. The authors in [42] defined the characteristics of a quality framework for model-driven engineering (MDE). In [43], the authors discussed the various challenges that affect the quality of model transformations and proposed design patterns as well as quantitative metrics to assess the quality of transformations.

There is a decent amount of work revolving around the definition of metrics to assess the quality of model transformations in general or for a particular transformation language [44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 23]. Both [55, 56] defined metrics for ATL, the latter though categorized ATL metrics into three main groups; rule, unit, and helper metrics. Similarly, in [57] metrics were divided into four groups; rule, helper, dependency, and miscellaneous metrics. In [58], however, the authors defined 27 quality metrics to measure six quality attributes: understandability, modularity, modifiability, reusability, completeness, and consistency. An emphasis of the need to relate metrics to quality attributes for ATL is detailed in [10] and the relation between performance and the size and complexity of input model was put under examination in [59] in addition to a comparison between the performance of execution engines for three transformation languages: ATL, QVTr, and QVTo.

In [60], the authors evaluated the external quality of transformation by applying metrics to both source and target models and evaluate the impact of the

transformation on the model's quality. In [61], a set of metrics were proposed to measure the change impact of ATL model transformations. Other contributions focused on a particular quality attribute; [62] identified the differences between transformation languages in terms of comprehensibility, whereas a set of metrics to measure the maintainability of QVT relational transformations has been proposed in [63]. Finally, in [64] the authors discuss the concept of technical depth for transformation languages by adapting quality flaws based on metrics for program code for different model transformation languages. Bad smells based on metrics for transformations written in the Epsilon Transformation Language (ETL) are reported in [65].

6.2. Search-Based Software Engineering and Model Driven Engineering

SBSE has been used to tackle major MDE challenges for a while, as the associated search spaces have the potential to be very large, SBSE techniques are gaining popularity in both academia and industry since they are very beneficial in terms of finding good solutions in a reasonable time [66].

Model transformation testing is considered as one of the main challenges in MDE as detailed in [67, 68]. The authors in [69, 70, 71, 72, 73] focused on test data generation. Others worked on minimizing the test suite [74], the definition of oracle function [44], and the automatic derivation of well-formedness rules [75].

Besides testing, the SBSE approach is extended to cover various MDE challenges; model versioning or model merging [76, 77, 78], transformation rules orchestration [79, 80, 81, 82, 83], and model refactoring in both design- and code-levels as is discussed next.

There is a number of studies that used an SBSE approach to detect or recommend model-refactoring opportunities; The authors in [84] proposed the REMODEL approach which uses both genetic programming and software met-

rics (based on QMOOD [25]) to generate design refactorings. The main two objectives of REMODEL are: (i) using QMOOD metrics to improve the design quality, and (ii) improving the maintainability of the software by introducing design patterns. A multi-level refactoring approach was presented in [85, 86], where both the source code and the design are taken into consideration during the refactoring process. The developer initially tailors the desired target design that is better in terms of quality metrics or developer's perspective (or both) and the source code will then be refactored accordingly. The model refactoring by example approach was considered in [87], the authors used genetic programming to detect refactoring opportunities concerning multiple model design anti-patterns by analyzing a couple of design defects examples from various systems and using this knowledge to generate defect detection rules. The authors went the extra mile by using an interactive genetic algorithm (IGA) in [88]. By adding the user's feedback to the fitness function, the approach became able to adapt the recommended sequence of refactorings to accommodate the developer's needs since the IGA better understood the semantics of the software system.

The idea of formalizing model transformations as a combinatorial optimization problem was first proposed in [89], several work followed this initiative to use search-based optimization techniques with model transformations for different intents. The pioneer contributions applied the search-based techniques to the model transformation by example either to generate transformation rules [45, 90, 91], recover transformation traces [92], or to generate target models [89, 93].

6.3. Refactoring in Model Driven Engineering

With respect to the automatic exploration of model transformation refactorings opportunities, we discuss in this section related approaches. Compared

to refactorings for different modeling languages, e.g., cf. [5, 94, 95, 51, 77, 96], to mention just a few approaches and surveys, only a few dedicated approaches have been developed for refactoring model transformations.

Dedicated refactoring operators for graph transformations have been presented in [12] with a concentration on certain quality aspects such as changeability, conciseness, and comprehensibility. Henshin-specific model transformation bad smells which have an impact on the performance have been discussed in [97]. The authors in [13] proposed clone detection and a merge-based rule refactoring approach for graph transformations which is related to inheritance-based ATL refactorings. However, the study focusses on the correctness of the merge-based rule refactorings, while we focus on the application of inheritance-based ATL refactorings with respect to quality metrics. Recently, Strüber et al. proposed a variability-based model transformation approach, in order to tackle two issues; the maintainability and the performance of model transformations [98].

In [14], the first refactoring catalog for model transformations is presented which has been implemented for ATL. In our contributions, we build on the refactoring operations presented but go beyond the automation support initially proposed by [14]. While in the previous work, the refactoring process is semi-automated, meaning that the refactoring operations have to be explicitly triggered by the user, in our work we provide a fully automated approach for searching the refactoring space of a model transformation.

6.4. Synopsis

While there have been efforts of defining metrics for determining the quality of model transformations as well as the creation of refactoring catalogs, these efforts have been happening mostly in an isolated manner. Furthermore, only semi-automated refactoring approaches have been proposed which do not allow to deal with the huge search space of model transformation refactoring in

a scalable manner. With this paper, we close the gap between reasoning on the quality of model transformations as well as their improvement in terms of refactoring by proposing a fully automated approach which is able to deal with the huge search space.

7. Conclusion

In this paper, we proposed an automated approach for refactoring ATL programs to find a trade-off between different conflicting objectives. We have also adapted an existing quality model, QMOOD, for the case of model transformations to guide the search for relevant refactorings. Our automated approach allows developers to benefit from search-based refactoring tools without manually identifying refactoring opportunities. To evaluate the effectiveness of our tool, we conducted a user study with several software developers who evaluated the tool and compared it with random search, a multi-objective adaption based on two quality metrics, an existing mono-objective formulation, and an approach not based on heuristic search. Statistical analysis of our experiments showed that our proposal performed significantly better than random search, our previous multi-objective work not based on QMOOD [24], a mono-objective formulation and the manual refactoring selection approach presented in [14] with an average precision and recall of 89% and 95% respectively when compared to manual solutions provided by a set of developers. The software developers, who participated in our experiments, confirmed also the relevance of the suggested refactorings as an outcome of a survey study.

Future work involves the extension of our approach to support automated regression testing since it is important to test the refactorings introduced to the ATL programs. Furthermore, we will address the problem of identifying antipatterns in ATL programs rather than just relying on quality attributes.

Finally, we will consider extending our current quality model with additional quality metrics such as the modularity measures discussed in [9, 10, 33]. In addition, we will adopt our multi-objective approach to consider the inclusion of ATL design patterns, defined in [34], which may increase further the quality of generated ATL programs.

References

References

[1] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, Morgan & Claypool Publishers, 2017.

[2] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, Model-driven software development: technology, engineering, management, John Wiley & Sons, 2013.

[3] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Systems Journal 45 (3) (2006) 621–645.

[4] D. C. Schmidt, Model-driven engineering, IEEE Computer 39 (2) (2006) 25–31.

[5] M. Mohamed, M. Romdhani, K. Ghedira, Classification of model refactoring approaches, Journal of Object Technology 8 (6) (2009) 121–126.

[6] A. Kusel, J. Schoenboeck, M. Wimmer, W. Retschitzegger, W. Schwinger, G. Kappel, Reality Check for Model Transformation Reuse: The ATL Transformation Zoo Case Study, in: Proceedings of the AMT Workshop, 2013, pp. 1–11.

[7] G. M. K. Selim, J. R. Cordy, J. Dingel, How is ATL really used? language feature use in the ATL zoo, in: 20th ACM/IEEE International Conference

on Model Driven Engineering Languages and Systems (MODELS), 2017, 2017, pp. 34–44.

[8] F. Allilaire, J. Bézivin, F. Jouault, I. Kurtev, Atl-eclipse support for model transformation, in: Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France, Vol. 66, Citeseer, 2006.

[9] M. F. van Amstel, M. van den Brand, Quality assessment of ATL model transformations using metrics, in: Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL), 2010.

[10] M. F. van Amstel, M. van den Brand, Using metrics for assessing the quality of atl model transformations, in: Proceedings of the Third International Workshop on Model Transformation with ATL (MtATL), 2011, pp. 20–34.

[11] I. Porres, Model refactorings as rule-based update transformations, in: Proceedings of the International Conference on the Unified Modeling Language (UML), Springer, 2003, pp. 159–174.

[12] G. Taentzer, T. Arendt, C. Ermel, R. Heckel, Towards refactoring of rule-based, in-place model transformation systems, in: Proceedings of the First Workshop on the Analysis of Model Transformations, ACM, 2012, pp. 41–46.

[13] D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, J. Plöger, Rule-merger: automatic construction of variability-based model transformation rules, in: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), Springer, 2016, pp. 122–140.

[14] M. Wimmer, S. M. Perez, F. Jouault, J. Cabot, A catalogue of refactorings for model-to-model transformations., Journal of Object Technology 11 (2).

[15] J. S. Cuadrado, E. Guerra, J. de Lara, Static analysis of model transformations, IEEE Transactions on Software Engineering 43 (9) (2017) 868–897.

[16] X. Ge, Q. L. DuBose, E. Murphy-Hill, Reconciling manual and automatic refactoring, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, 2012, pp. 211–221.

[17] M. Fowler, Refactoring - Improving the Design of Existing Code, Addison Wesley object technology series, Addison-Wesley, 1999.
URL `http://martinfowler.com/books/refactoring.html`

[18] G. Sunyé, D. Pollet, Y. Le Traon, J.-M. Jézéquel, Refactoring uml models, in: International Conference on the Unified Modeling Language, Springer, 2001, pp. 134–148.

[19] T. Mens, G. Taentzer, D. Müller, Model-driven software refactoring, in: Software Applications: Concepts, Methodologies, Tools, and Applications, IGI Global, 2009, pp. 3455–3488.

[20] T. Mens, G. Taentzer, D. Müller, Model-driven software refactoring, in: 1st Workshop on Refactoring Tools (WRT), 2007, pp. 25–27.

[21] G. Sunyé, D. Pollet, Y. L. Traon, J. Jézéquel, Refactoring UML models, in: UML 2001 - 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools, 2001, pp. 134–148.

[22] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, D. S. Kolovos, R. F. Paige, M. Lauder, A. Schürr, D. Wagelaar, Surveying rule inheritance in model-to-model transformation languages, Journal of Object Technology 11 (2) (2012) 3: 1–46.

[23] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, B. Alkhazi, Model trans-

formation modularization as a many-objective optimization problem, IEEE Transactions on Software Engineering 43 (11) (2017) 1009–1032.

[24] B. Alkhazi, T. Ruas, M. Kessentini, M. Wimmer, W. I. Grosky, Automated refactoring of ATL model transformations: a search-based approach, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS), ACM, 2016, pp. 295–304.

[25] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on Software Engineering 28 (1) (2002) 4–17.

[26] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multi-objective genetic algorithm: NSGA-II, IEEE transactions on evolutionary computation 6 (2) (2002) 182–197.

[27] Eclipse Modeling Project, ATL Transformations Zoo (2015).
URL `https://www.eclipse.org/atl/atlTransformations/`

[28] D. Wagelaar, R. V. D. Straeten, D. Deridder, Module superimposition: a composition technique for rule-based model transformation languages, Software and Systems Modeling 9 (3) (2010) 285–309.

[29] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, W. Schwinger, Reuse in model-to-model transformation languages: are we there yet?, Software & Systems Modeling 14 (2) (2015) 537–572.

[30] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Science of computer programming 72 (1-2) (2008) 31–39.

[31] M. Alshayeb, Empirical investigation of refactoring effect on software quality, Information and software technology 51 (9) (2009) 1319–1326.

[32] J. Al Dallal, A. Abdin, Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: a systematic literature review, IEEE Transactions on Software Engineering 44 (1) (2017) 44–69.

[33] M. F. Van Amstel, M. G. Van Den Brand, Model transformation analysis: Staying ahead of the maintenance nightmare, in: International Conference on Theory and Practice of Model Transformations, Springer, 2011, pp. 108–122.

[34] K. Lano, S. Kolahdouz-Rahimi, Model-transformation design patterns, IEEE Transactions on Software Engineering 40 (12) (2014) 1224–1259.

[35] L. Rachmawati, D. Srinivasan, Multiobjective evolutionary algorithm with controllable focus on the knees of the pareto front, IEEE Transactions on Evolutionary Computation 13 (4) (2009) 810–824.

[36] A. Arcuri, L. Briand, 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, 2011, pp. 1–10.

[37] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[38] A. Correa, C. Werner, M. Barros, An empirical study of the impact of OCL smells and refactorings on the understandability of OCL specifications, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2007, pp. 76–90.

[39] A. Correa, C. Werner, Applying refactoring techniques to UML/OCL models, in: International Conference on the Unified Modeling Language, Springer, 2004, pp. 173–187.

[40] A. Correa, C. Werner, Refactoring object constraint language specifications, Software & Systems Modeling 6 (2) (2007) 113–138.

[41] J. Reimann, C. Wilke, B. Demuth, M. Muck, U. Aßmann, Tool supported OCL refactoring catalogue, in: Proceedings of the 12th Workshop on OCL and Textual Modelling, ACM, 2012, pp. 7–12.

[42] P. Mohagheghi, V. Dehlen, Developing a quality framework for model-driven engineering, in: Models in Software Engineering, 2008, pp. 275–286.

[43] E. Syriani, J. Gray, Challenges for addressing quality factors in model transformation, in: Proceedings of the Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 929–937.

[44] M. Kessentini, H. A. Sahraoui, M. Boukadoum, Example-based model-transformation testing, Automated Software Engineering 18 (2) (2011) 199–224.

[45] M. Kessentini, A. Bouchoucha, H. A. Sahraoui, M. Boukadoum, Example-based sequence diagrams to colored petri nets transformation using heuristic search, in: ECMFA'10 Proceedings of the 6th European conference on Modelling Foundations and Applications, 2010, pp. 156–172.

[46] M. Kessentini, H. Sahraoui, M. Boukadoum, M. Wimmer, Search-based design defects detection by example, in: International Conference on Fundamental Approaches to Software Engineering, Springer, Berlin, Heidelberg, 2011, pp. 401–415.

[47] A. ben Fadhel, M. Kessentini, P. Langer, M. Wimmer, Search-based detection of high-level model changes, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 212–221.

[48] M. Kessentini, M. Wimmer, H. Sahraoui, M. Boukadoum, Generating transformation rules from examples for behavioral models, in: Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, ACM, 2010, p. 2.

[49] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, S. B. Chikha, Competitive coevolutionary code-smells detection, in: International Symposium on Search Based Software Engineering, Springer, Berlin, Heidelberg, 2013, pp. 50–65.

[50] A. Ouni, R. Gaikovina Kula, M. Kessentini, K. Inoue, Web service antipatterns detection using genetic programming, in: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, ACM, 2015, pp. 1351–1358.

[51] U. Mansoor, M. Kessentini, M. Wimmer, K. Deb, Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm, Software Quality Journal 25 (2) (2017) 473–501.

[52] U. Mansoor, M. Kessentini, B. R. Maxim, K. Deb, Multi-objective code-smells detection using good and bad design examples, Software Quality Journal 25 (2) (2017) 529–552.

[53] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, K. Deb, A robust multi-objective approach to balance severity and importance of refactoring opportunities, Empirical Software Engineering 22 (2) (2017) 894–927.

[54] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, K. Inoue, Search-based software library recommendation using multi-objective optimization, Information and Software Technology 83 (2017) 55–75.

[55] J. B. Tolosa, O. Sanjuán-Martínez, V. García-Díaz, B. C. P. G-Bustelo, J. M. C. Lovelle, Towards the systematic measurement of ATL transformation models, Software - Practice and Experience 41 (7) (2011) 789–815.

[56] A. Vignaga, Metrics for measuring atl model transformations, Tech. rep., Department of Computer Science, Universidad de Chile (2009).

[57] M. M. van Amstel, Assessing and improving the quality of model transformations, Ph.D. thesis, Technische Universiteit Eindhoven (2012).

[58] M. M. van Amstel, C. C. Lange, M. M. van den Brand, Using Metrics for Assessing the Quality of ASF+SDF Model Transformations, in: ICMT '09 Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations, 2009, pp. 239–248.

[59] M. van Amstel, S. Bosems, I. Kurtev, L. F. Pires, Performance in model transformations: experiments with ATL and QVT, in: Proceedings of the 4th international conference on Theory and practice of model transformations (ICMT), 2011, pp. 198–212.

[60] M. Saeki, H. Kaiya, Measuring model transformation in model driven development., in: Proceedings of CAiSE Forum, 2007.

[61] A. Vieira, F. Ramalho, Metrics to Measure the Change Impact in ATL Model Transformations, in: International Conference on Product-Focused Software Process Improvement, 2014, pp. 254–268.

[62] S. K. Rahimi, K. Lano, Integrating goal-oriented measurement for evaluation of model transformation, in: Proceedings of the CSI International Symposium on Computer Science and Software Engineering (CSSE), 2011, pp. 129–134.

[63] L. Kapová, T. Goldschmidt, S. Becker, J. Henss, Evaluating maintainability with code metrics for model-to-model transformations, in: Proceedings of the 6th International Conference on Quality of Software Architectures (QoSA), 2010, pp. 151–166.

[64] K. C. Lano, H. A. A. Alfraihi, Technical debt in model transformation specifications, in: Proceedings of the 11th International Conference on Theory and Practice of Model Transformation (ICMT), 2018, pp. 127–141.

[65] N. Bonet, K. Garcés, R. Casallas, M. E. Correal, R. Wei, Influence of programming style in transformation bad smells: mining of ETL repositories, Computer Science Education 28 (1) (2018) 87–108.

[66] I. Boussaïd, P. Siarry, M. Ahmed-Nacer, A survey on search-based model-driven engineering, automated software engineering 24 (2) (2017) 233–294.

[67] R. V. D. Straeten, T. Mens, S. V. Baelen, Challenges in model-driven software engineering, in: Models in Software Engineering, Workshops and Symposia at MODELS 2008, 2009, pp. 35–47.

[68] B. R. Bryant, J. G. Gray, M. Mernik, P. Clarke, G. Karsai, Challenges and directions in formalizing the semantics of modeling languages, Computer Science and Information Systems 8 (2) (2011) 225–253.

[69] A. A. Jilani, M. Z. Iqbal, M. U. Khan, A search based test data generation approach for model transformations, in: Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT), 2014, pp. 17–24.

[70] J. Shelburg, M. Kessentini, D. R. Tauritz, Regression testing for model transformations: A multi-objective approach, in: Proceedings of the In-

ternational Symposium on Search Based Software Engineering, 2013, pp. 209–223.

[71] W. Wang, M. Kessentini, W. Jiang, Test cases generation for model transformations from structural information, MDEBE@MoDELS (2013) 42–51.

[72] J. J. C. Gomez, B. Baudry, H. Sahraoui, Searching the boundaries of a modeling space to test metamodels, in: Proceedings of the Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 131–140.

[73] D. Sahin, M. Kessentini, M. Wimmer, K. Deb, Model transformation testing: a bi-level search-based software engineering approach, Journal of Software: Evolution and Process 27 (11) (2015) 821–837.

[74] L. M. Rose, S. M. Poulding, Efficient probabilistic testing of model transformations using search, in: Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering, 2013, pp. 16–21.

[75] M. Faunes, J. J. Cadavid, B. Baudry, H. A. Sahraoui, B. Combemale, Automatically searching for metamodel well-formedness rules in examples and counter-examples, in: Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems, 2013, pp. 187–202.

[76] M. Kessentini, W. Werda, P. Langer, M. Wimmer, Search-based model merging, in: Proceedings of the 15th annual conference on Genetic and evolutionary computation, 2013, pp. 1453–1460.

[77] U. Mansoor, M. Kessentini, P. Langer, M. Wimmer, S. Bechikh, K. Deb, Momm: Multi-objective model merging, Journal of Systems and Software 103 (2015) 423–439.

[78] C. Debreceni, I. Ráth, D. Varró, X. D. Carlos, X. Mendialdua, S. Trujillo, Automated model merge by design space exploration, in: Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering, 2016, pp. 104–121.

[79] J. Denil, M. Jukss, C. Verbrugge, H. Vangheluwe, Search-based model optimization using model transformations, in: International Conference on System Analysis and Modeling, 2014, pp. 80–95.

[80] M. Fleck, J. Troya, M. Wimmer, Marrying search-based optimization and model transformation technology, Proc. of NasBASE (2015) 1–16.

[81] S. Gyapay, Ákos Schmidt, D. Varró, Joint optimization and reachability analysis in graph transformation systems with time, Electronic Notes in Theoretical Computer Science 109 (2004) 137–147.

[82] M. W. Mkaouer, M. Kessentini, S. Bechikh, D. R. Tauritz, Preference-based multi-objective software modelling, in: Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering, 2013, pp. 61–66.

[83] H. Abdeen, D. Varró, H. A. Sahraoui, A. S. Nagy, C. Debreceni, Ábel Hegedüs, Ákos Horváth, Multi-objective optimization in rule-based design space exploration, in: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 289–300.

[84] A. C. Jensen, B. H. Cheng, On the use of genetic programming for automated refactoring and the introduction of design patterns, in: Proceedings of the 12th annual conference on Genetic and evolutionary computation, 2010, pp. 1341–1348.

[85] I. H. Moghadam, M. O. Cinneide, Automated refactoring using design differencing, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 43–52.

[86] I. H. Moghadam, Multi-level automated refactoring using design exploration, in: Proceedings of the Third international conference on Search based software engineering (SSBSE), 2011, pp. 70–75.

[87] A. Ghannem, M. Kessentini, G. E. Boussaidi, Detecting model refactoring opportunities using heuristic search, in: CASCON '11 Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research, 2011, pp. 175–187.

[88] A. Ghannem, G. E. Boussaidi, M. Kessentini, Model refactoring using interactive genetic algorithm, in: SSBSE 2013 Proceedings of the 5th International Symposium on Search Based Software Engineering - Volume 8084, 2013, pp. 96–110.

[89] M. Kessentini, H. A. Sahraoui, M. Boukadoum, Model transformation as an optimization problem, in: MoDELS '08 Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, 2008, pp. 159–173.

[90] M. Faunes, H. A. Sahraoui, M. Boukadoum, Genetic-programming approach to learn model transformation rules from examples, in: International Conference on Theory and Practice of Model Transformations, 2013, pp. 17–32.

[91] I. Baki, H. A. Sahraoui, Q. Cobbaert, P. Masson, M. Faunes, Learning implicit and explicit control in model transformations by example, in: International Conference on Model Driven Engineering Languages and Systems, 2014, pp. 636–652.

[92] H. Saada, M. Huchard, C. Nebut, H. A. Sahraoui, Recovering model transformation traces using multi-objective optimization, in: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, 2013, pp. 688–693.

[93] M. Kessentini, H. Sahraoui, M. Boukadoum, O. B. Omar, Search-based model transformation by example, Software and Systems Modeling 11 (2) (2012) 209–226.

[94] J. Zhang, Y. Lin, J. Gray, Generic and domain-specific model refactoring using a model transformation engine, Model-Driven Software Development (2005) 199–217.

[95] M. Misbhauddin, M. Alshayeb, UML model refactoring: a systematic literature review, Empirical Software Engineering 20 (1) (2015) 206–251.

[96] A. Ghannem, G. E. Boussaidi, M. Kessentini, Model refactoring using examples: a search-based approach, Journal of Software: Evolution and Process 26 (7) (2014) 692–713.

[97] M. Tichy, C. Krause, G. Liebel, Detecting performance bad smells for henshin model transformations., in: Proceedings of the 2nd Workshop on the Analysis of Model Transformations (AMT), 2013.

[98] D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, J. Plöger, Variability-based model transformation: formal foundation and application, Formal Aspects of Computing 30 (1) (2018) 133–162.