FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Search-Based Model Transformations

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Martin Fleck, MSc
Matrikelnummer 1248308

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Priv.-Doz. Mag. Dr. Manuel Wimmer

Diese Dissertation haben begutachtet:

Gerti Kappel | Marouane Kessentini

Wien, 20. April 2016

Martin Fleck

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Search-Based Model Transformations

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Martin Fleck, MSc
Registration Number 1248308

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Mag. Dr. Manuel Wimmer

The dissertation has been reviewed by:

Gerti Kappel

Marouane Kessentini

Vienna, 20th April, 2016

Martin Fleck

Erklärung zur Verfassung der Arbeit

Martin Fleck, MSc
Barawitzkagasse 34/4/50
1190 Wien
Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. April 2016

Martin Fleck

To Niki.

Acknowledgements

Writing these acknowledgements is the final activity before my thesis is finished. At this point, I must confess that I struggle to find the right words to thank the many people who have supported me along the way. Without the invaluable contributions of these people, this thesis would not have been possible. Not everyone is mentioned here by name, but I am eternally grateful to all of you.

I want to start by thanking my advisor, Dr. Manuel Wimmer, for always encouraging me to explore interesting research ideas and continuously supporting me in spreading these ideas.

I want to thank Prof. Marouane Kessentini for supporting me with valuable feedback on my research and for providing me with insights into search-based software engineering.

I am also grateful to Prof. Gerti Kappel, who gave me the opportunity to do my PhD in her outstanding research group. Her vast experience and her keen eye for detail (always double-check your references!) helped improve many of my papers and presentations.

Of course, this thesis would not have been possible without the stimulating environment created by my colleagues, who have become dear friends over the past years. This includes my fellow ARTISTs Alex, Javi, Michi, and Patrick, Robert, and Manuel, Philip, and Tanja from the other side of the hallway (from the room with the good coffee machine). Thank you for many inspiring and motivational discussions as well as the great moments we shared both inside and outside the office.

I am particularly indebted to Tanja, who has always listened to me patiently and encouraged me to not give up in times when I was not sure how to continue. Thank you.

I also want to acknowledge my friends outside of work, who helped me balance the long nights in the office with long nights in pubs in Vienna and Linz. Thank you Koza, Anni, Steffi, Schügi, Marina, Manu, Berndi, Michi, Silvia, Adi, Natascha, Matthias, Hubsi, Emi, Julian, Birgit, Berni, René, Andi, Dennis, and Catherine.

I furthermore owe my gratitude to my loving family who has always been a source of happiness for me. Thank you Mom, Dad, Dre, Iveta, Vivi, Mischi, Markus, Ines, Nori, Mani, Emily, Hilde, Elmar, Gabi, Arthur, Maxi, Leo, and Diego.

Finally, I am deeply thankful to my boyfriend Niki, who has always supported me unconditionally. Words cannot express how grateful I am for the strength he has given me during this PhD and how fortunate I consider myself for our life together.

Kurzfassung

Model-Driven Engineering (MDE) ist ein Paradigma, in der Modelle als zentrale Artefakte zur Problemlösung eingesetzt werden. Die Problemdomäne wird durch eine domänenspezifische Modellierungssprache definiert und Modelle repräsentieren konkrete Probleminstanzen, welche von der Realität abstrahieren um unnötige Komplexität zu vermeiden. Im Kern von MDE verwendet man Modelltransformationen um jede systematische Änderung an diesen Modellen durchzuführen. Die Orchestrierung dieser Transformationen um konkrete Probleme zu lösen ist jedoch eine komplexe Aufgabe, da der zu durchsuchende Transformationsraum sehr groß bis unendlich groß sein kann. Daher wird diese Aufgabe entweder automatisiert durchgeführt, indem Regeln so lange wie möglich angewendet werden, was jedoch nicht immer zufriedenstellende Resultate liefert, oder die Aufgabe wird an den Modellierer zur manuellen Lösung abgegeben. Dies führt dazu, dass MDE nur in geringem Maße dazu eingesetzt werden kann, Probleme zu lösen, die einen unendlich großen Lösungsraum haben oder manuell schwer lösbar sind.

Aus diesem Grund stellen wir in dieser Arbeitet einen Ansatz vor, der es ermöglicht derartige Probleme zu lösen indem die zu optimierenden Eigenschaften durch modellbasierte Analysetechniken operationalisiert werden und metaheuristischen Methoden auf Modellebene gehoben werden um optimale Transformationsorchestrierungen zu finden. Im ersten Schritt präsentieren wir einen Ansatz, der dynamische, zeitbasierte Eigenschaften unter Berücksichtigung des Resourcenbedarfs direkt auf Modellebene mittels fUML analysieren kann. Im zweiten Schritt kodieren wir das Transformationsorchestrierungsproblem generisch, wodurch eine Vielzahl verschiedener metaheuristischen Methoden eingesetzt werden können. Anschließend, entwickeln wir auf Basis dieser Kodierung einen Ansatz, der ein deklaratives Lösen von Problemen auf Modellebene ermöglicht, indem ein Modellierer das Problemmodell und die jeweiligen Modelltransformationen bereit stellt und die zu optimierenden Eigenschaften und die Zwangsbedingungen deklariert. Die Konfiguration wird durch eine dedizierte Sprache unterstützt, welche allgemeine Informationen bietet und Feedback zur aktuellen Parametrisierung liefert. Als Resultat stellen wir die orchestrierten Transformationen, die daraus resultierenden Lösungsmodelle, die Werte der Optimierungseigenschaften und Bedingungen sowie zusätzliche Analyseinformationen bereit. Unser Ansatz basiert auf Graphtransformationen und wurde als quelloffenes Framework namens MOMoT implementiert. Die Effektivität unseres Ansatzes wird in einer intensiven Evaluierung auf Basis von verschiedenen Fallstudien und zwei neuen Problemdefinitionen aus den Bereichen Software Engineering und MDE validiert.

Abstract

Model-Driven Engineering (MDE) is a paradigm that promotes the use of models as the central artifacts for solving problems. In MDE, problem domains are specified using domain-specific modeling languages and models are concrete problem instances that abstract from reality to reduce complexity. At the heart of MDE, model transformations are used to systematically manipulate these problem models to find good solutions to the problem at hand. However, reasoning about how the transformation needs to be orchestrated to find good solutions is a non-trivial task due to the large or even infinite transformation space. As a result, this task is either performed automatically, e.g., by following an apply-as-long-as-possible approach, which does not necessarily produce satisfactory results, or it is carried out manually by the respective engineer. This, in turn, hampers the application of MDE techniques on complex problems which usually cannot be solved manually or by enumerating all possible solutions.

Therefore, we present in this thesis an approach that facilitates to solve these problems by stating clear objectives operationalized through model-based analysis techniques and elevating search-based optimization methods to the model level to find optimal transformation orchestrations. As first contribution, we introduce a model-based analysis approach that measures dynamic, timed properties that consider the contention of resources directly on the model level using the fUML standard. As second contribution, we provide a generic encoding of the transformation orchestration problem on which many different optimization methods can be applied. Using this encoding, we propose an approach that enables to solve problems by providing a model, a set of transformation rules, a set of objectives that are optimized during the process and a set of constraints that mark invalid solutions. The optimization process is configured through a dedicated language which provides information on the optimization concepts and immediate feedback for the concrete configuration. The results consist of the respective orchestrated transformations, the solution models, the objective and constraint values as well as analysis details about the optimization process. Our approach is based on graph transformations and has been implemented as an open-source framework called MOMoT. Based on this implementation, we provide an extensive evaluation of our approach using several case studies from the area of model-driven software engineering as well as two novel problem formulations that tackle the modularization of model transformations and the generic modularization of modeling languages. The obtained evaluation results validate the effectiveness of our approach and give rise to interesting lines of research.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Model-Driven Engineering (MDE) is a paradigm that promotes the use of models as a central artifact [BCW12, Béz05] to solve problems on a higher level of abstraction. In MDE, the respective problem domain is expressed using domain-specific modeling languages and models are the concrete problem instances that conform to this language. One core principle of MDE is abstraction. By abstracting [Kra07] from reality and focusing on the specific purpose of a model, we can deal with the world in a simplified manner and are able to avoid unnecessary complexity [Rot89].

A very successful research line that has originated from this principle is the field of model-based software engineering (MBSE) where software engineering is the problem domain under consideration. In MBSE, models are used throughout the software engineering life cycle to simplify the design process and increase productivity. Indeed, it is recommended to test and optimize the system already in the design phase [MNB+13], since errors that are detected in this phase are much cheaper and easier to correct than those detected later on [CLR+09, CHM+02]. Therefore, models are used to describe complex systems from various viewpoints and at multiple levels of abstraction using appropriate modeling formalisms.

At the heart of MDE are model transformations [SK03] which provide the essential mechanism for manipulating models. Their application field ranges from simple editing operations, visualization, and analysis to the translation of models from one language to another [LAD+14]. For instance, transformations may be used to refactor a model, i.e., improve the models structure while preserving its observable behaviour, abstract software models such as class models from existing source code using reverse-engineering techniques [BCJM10], or obtain knowledge about the system by extracting domain models out of a class diagram [BGWK14]. In general, model transformations are executed

using a transformation engine and expressed using model transformation languages. Several distinct categories of model transformation languages have been identified [CH06, LAD+14]. In broader terms, there exist declarative, imperative, and hybrid languages, most of which are expressed by means of transformation rules.

A crucial aspect when dealing with rule-based model transformations is the orchestration of the rules that compose them. This orchestration consists of the rule parameterization, i.e., the setting of the rule parameters, and the rule scheduling, i.e., the determination of the order in which the rules need to be executed. The transformation can be orchestrated implicitly or explicitly [CH06]. In an implicit orchestration, the order in which the rules are triggered is decided by the transformation engine and the developer has no control mechanism. This is typically the case with purely declarative languages, such as QVT Relations [Kur08], Triple-Graph-Grammars [Sch95], and many graph-based transformation languages [RDV09, Tae03, Agr03]. Other languages often provide mechanisms to explicitly define the rule scheduling. For instance, ATL [JABK08] is a hybrid transformation language that offers the possibility of partially orchestrating the rules by explicitly calling rules from the declarative part of the language. Other languages provide more dedicated mechanisms to schedule rules, such as VIATRA [CHM+02] which uses abstract state machines. As such, in an explicit rule orchestration, the complex task of finding a rule orchestration is shifted to the model engineer.

1.2 Problem Statement

When we apply MDE to solve problems on a higher level of abstraction, a model engineer defines transformation rules to manipulate an input model representing the problem instance to obtain good solutions, i.e., models with desired characteristics. However, reasoning about the orchestration of these rules to find good solutions is a non-trivial task suffering from four major challenges. First, the effect a rule application has on the characteristics of the resulting model is implicitly encoded in the behavior of the rule. Second, two rule applications may be in conflict or may enable each other [BAHT15], i.e., they offer alternatives or a given rule may check for some information produced by another rule. Third, the number of rule combinations may be very large or even infinite, especially when considering the parameters a rule may have, making an exhaustive exploration of rule orchestrations difficult or even impossible. And finally, the expected solutions may need to optimize several, potentially conflicting characteristics which further complicates the reasoning process. Thus, deciding on a rule orchestration to optimize a given model is a challenging task which hampers solving of complex problems on the model level.

The process of tackling such decision problems consists generally of four steps, as depicted in Figure 1.1 [Tal09]. First, the decision problem is identified and the objectives of the problems are outlined. Second, an optimization model is built for an abstracted version of the problem. Third, an optimization method generates solutions for the problem and depending on the problem and the respective optimization method, these solutions may or may not be optimal. And finally, a selected solution is implemented and evaluated to

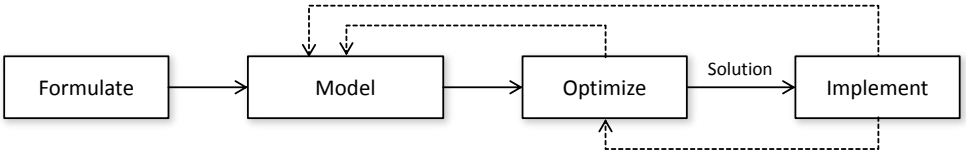


Figure 1.1: Classical Process in Decision Making [Tal09]

see whether it is acceptable or not. If a selected solution is not acceptable, either the optimization model or the solving method may be adapted until an acceptable solution is found.

The field of mathematical optimization is concerned with solving decision problems. Specifically, in search-based optimization we are able to tackle highly complex problems with large or even infinite search spaces. A solution to such a problem is encoded as a set of decision variables that are optimized by the respective optimization method according to a given set of objectives and constraints. Common types of decision variables include real-valued variables, binary variables, permutations, and trees. The optimization method manipulates these decision variables using dedicated operators to explore the solution space and to guide the search towards solutions with good objective values while considering the specified constraints. As optimization methods, we can distinguish between exact optimization methods that can guarantee the optimality of the retrieved solutions and approximate methods that are able to generate high-quality solutions in a reasonable time, but in turn cannot guarantee their optimality. If possible, the use of exact optimization methods is preferred. However, if the problem is complex, i.e., NP-complete, an exact method needs at least exponential time to produce optimal solutions [Tal09]. This is not feasible for most real world problems in which case approximate methods can be used instead. Metaheuristics are a family of general-purpose approximation methods that are applicable to most optimization problems and are often applied when it is infeasible to use exact methods [ZTL+03]. The goal of metaheuristics is to explore the solution space efficiently in order to find (near-)optimal solutions [BR03].

Recently a lot of research has been done in the field of search-based software engineering (SBSE) [HJ01], where metaheuristic methods are applied on software engineering problems to deal with their inherent complexity. In a similar spirit, we propose in this thesis the integration of metaheuristics with MDE to solve complex problems on the model level.

However, there is a gap between the formalisms used in search-based optimization and the techniques used in MDE. In MDE, the problem domain is constructed using modeling languages and solutions to these problems are models whereas in search-based optimization solutions to problems are encoded as a set of decision variables. Furthermore, in MDE there is no dedicated technique to tackle large transformation search spaces based on a set of declaratively given objectives like in search-based optimization. Instead domain-specific model transformation rules are used to modify models and the optimization is performed by orchestrating the transformation, a task often delegated to the model

engineer. However, there are problems where a manual or exhaustive exploration of the transformation search space is not feasible. In order to tackle these problems using metaheuristics, a model engineer would need to manually translate the transformation rules into search operators, invent a dedicated encoding of decision variables for the problem domain and map any domain-specific constraints and the necessary objectives onto this encoding. Furthermore, to subsequently process the solutions, they need to be converted back from the specific encoding to the model formalism. Besides, translating transformation rules to another formalism and inventing an encoding for a dedicated problem is considered a complex process. Moreover, once an encoding has been defined, integrating changes may become quite expensive as objective calculations, constraint definitions, search operators and the translation back depend on that encoding.

1.3 Aim of the Work

In this work, we aim to bridge this gap between search-based optimization and MDE formalisms and facilitate the formulation of an optimization problem directly on model level. This way, model engineers can use the languages and tools they are familiar with and at the same time profit from the declarative objective specification and the exploration capabilities of metaheuristic methods. Specifically, we focus on the following research questions:

(*i*) How can we measure properties on model level to declaratively express objectives and constraints?

(*ii*) How can we define a generic encoding that combines metaheuristics and transformations to solve complex problems on model level?

(*iii*) What optimization methods can we apply on such an encoding?

(*iv*) What are the resulting advantages and challenges to be tackled?

In this thesis, these research questions are addressed by two main contributions. Before we discuss each contribution in detail, we outline the envisioned model-level optimization approach (see Figure 1.2) which constitutes the context of the contributions. Our approach facilitates the formulation of a search problem through MDE techniques and seamlessly integrates search-based optimization methods that a model engineer can use to solve complex problems on model level. The core idea of our approach is to formulate the transformation orchestration problem as a search-based problem and therefore deal with the large or even infinite transformation space [Kur05] that can occur for problems defined on model level. As such, we provide a generic encoding based on model transformations and elevate the necessary input such as the configuration of the search process that optimizes the model. Finally, we propose an analysis method to retrieve the values of common properties on model level that can be used in the objective and constraint specification of the optimization approach.

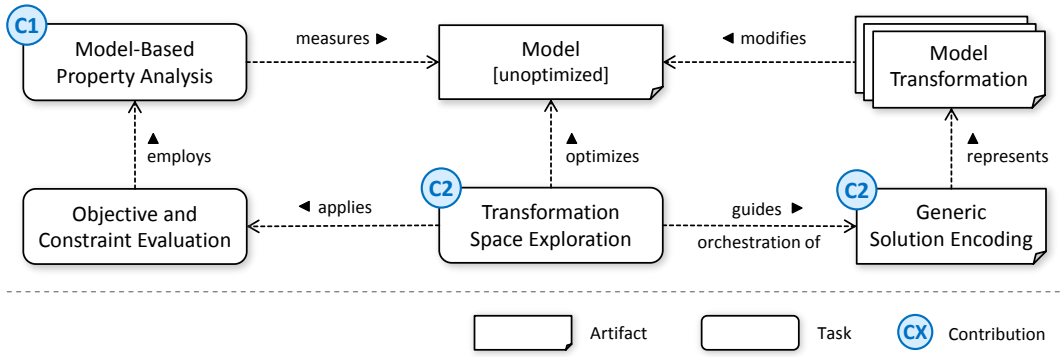


Figure 1.2: Model-Level Optimization Approach and Contributions of this Thesis

The goal of our approach is to support model engineers in solving problems which are either too large or too expensive to solve through an exhaustive or manual approach. Specifically, we provide an approach which enables model engineers to solve their problems declaratively through objectives and constraints and not by specifying the transformation orchestration. For such an approach, the following requirements are desirable.

Generic The approach must be problem- and optimization method-agnostic so that no assumptions about the problem and the selected optimization method need to be made in advance. Switching between methods should be simple and the changes necessary to switch from one problem to another problem should be minimal to reduce the learning effort.

Transparent In order to understand the approach, the executed process must be transparent to the model engineer. This means that the approach must (*i*) take the artifacts produced by the model engineer as input, i.e., the model which constitutes the problem instance and the set of model transformation rules that can manipulate this model, (*ii*) produce output on the same level as the input, i.e., an output model conforming to a metamodel and a sequence of orchestrated transformation rules, and (*iii*) provide analysis information about the search process.

Declarative The goal of the transformation orchestration, i.e., the objectives and constraints of the output model, must be separated from the model transformation specification and elevated to a first-class citizen. By making the objectives, which are typically implicitly defined as part of the model transformations semantics, explicit, we make both the transformation and the objective specifications re-usable for other problems. Furthermore, explicit objectives can serve as documentation for the problem solving process. The language(s) in which the objectives and constraints can be defined should be known to the model engineer.

Supportive Since it cannot be assumed that model engineers are proficient in search-based optimization methods, a support system must be in place that provides

general information about these methods and prevents common mistakes when configuring the methods for a specific problem.

In this thesis, there are two main contributions that support such an approach in order to answer the research questions.

C1: Model-Based Property Analysis. In our optimization approach, objectives and constraints can be given declaratively on model level. Objectives are properties of an output model that need to be optimized whereas constraints are properties that an output model needs to satisfy in order to be considered valid. In general, such properties may be categorized into static or dynamic properties and functional or non-functional properties. Static properties can be analyzed using static methods, such as model queries, whereas dynamic properties can only be evaluated by considering the runtime behavior of the respective model, e.g., through monitoring. Functional properties relate to aspects of the system that are concerned with its behavior, e.g., whether the correct functionality is provided or not. Non-functional properties on the other hand focus on how well this functionality is provided, e.g., the performance of a function or the considered security model. In general, the retrieval of these properties is done using dedicated model-based analysis techniques.

In this thesis, we propose a novel analysis approach that allows the retrieval of dynamic, non-functional properties for UML models directly on model-level. Previous strategies often translate the respective models into dedicated formalisms such as queuing networks. However, such translational approaches introduce inevitably an additional level of indirection, additional notations, and hence additional complexity, such as the consistent propagation of UML model changes to the analysis model and the translation of the analysis results back on the UML model. Our proposed approach, however, analyzes properties directly on model level by using the operational semantics of the recent Foundational UML (fUML) [OMG16] standard to obtain model execution traces on which further analysis can be performed. The application of our approach is shown for performance properties considering the contention of resources.

Our model-based analysis approach addresses research question (i).

C2: Marrying Optimization and Model Transformations. The second contribution of this thesis focuses on the integration of metaheuristics with MDE. Specifically, in this contribution we realize a generic encoding to tackle the transformation orchestration problem. In this encoding, a decision variable is a transformation unit, i.e., transformation rule or transformation component providing control-flow semantics, which can be adapted by the optimization method. Consequently, a solution consists of an ordered sequence of these decision variables, i.e., a complete transformation orchestration. In our encoding, we abstract from concrete transformation languages by defining a set of supported transformation units. As a result, our encoding may be realized through any rule-based transformation language that can implement these units.

Based on our encoding, we realize our model-level optimization approach that aims to fulfil the requirements stated before. The input for this approach are the model that represents the problem to be tackled, a set of transformation units to modify that model, the objectives and constraints that need to be optimized and fulfilled by a solution to this problem, and a configuration for the search and the selected optimization methods. Our approach offers generic interfaces for different metaheuristic methods and provides a set of method implementations. In order to support the model engineer of our approach in configuring the methods and the search process, we offer a dedicated configuration modeling language that provides general information about the concepts used in metaheuristic optimization and provides feedback regarding the current state of the configuration. Hence, all input for our approach can be provided using techniques with which model engineers are familiar. As result of applying our approach, the model engineer obtains the transformation orchestrations, the resulting output models, the objective and constraint values of those models, and a statistical analysis of the optimization process.

Our model-level optimization approach addresses research questions (*ii*), (*iii*), and (*iv*).

Implementation. The artifacts developed in the course of this thesis have been realized as research prototypes. These prototypes are integrated with the Eclipse Modeling Framework [SBPM08, Ecl16b] and are published on our websites [BIG16, FTW16a].

1.4 Methodological Approach

For carrying out this thesis, we apply the design science approach as underlying methodological approach. Design science is a constructive methodological approach where knowledge is created by building and evaluating innovative artifacts. Hevner et al. [HMPR04, Hev07] introduced a conceptual framework as well as seven guidelines for applying design science in software systems research. These seven guidelines for conducting design science in information systems research are applied in this thesis.

1. Design as an Artifact. The aim of this thesis is to design an approach for automatically deriving orchestrations of model transformations in order to facilitate the solving of complex problems on model level satisfying the defined objectives and constraints. More precisely the following artifacts have been built in the course of the thesis.

(i) A strategy to calculate dynamic, non-functional properties that consider the contention of resources on model level has been developed (cf. Chapter 3).

(ii) A generic solution encoding to tackle the model transformation orchestration problem has been defined (cf. Chapter 4).

(iii) An algorithm- and problem-agnostic approach to solve optimization problems on model level using metaheuristic optimization and model transformations has

been developed by building upon the Eclipse Modeling Framework (EMF) (cf. Chapter 4).

(iv) A set of case studies and experiments to evaluate the approach have been created and are publicly available to enable re-execution and the reproduction of the results (cf. Chapter 5).

2. Problem Relevance. In MDE, problem domains can be represented using dedicated modeling languages and problems can then be expressed using the concepts available from that domain. In general, solving problems on model-level is often cheaper and easier as models abstract from the concrete reality of the problem and we can focus on aspects relevant to solving the problem. Model transformations are used to manipulate models in order to find solutions for the respective problem. However, reasoning about these model transformations for complex problems is a non-trivial task with several drawbacks hampering the tackling of those problems on model level. Therefore, we need a mechanism that automatically determines the optimal transformation orchestration for a specific scenario in a setting where the number of possible rule combinations may be very large or even infinite. For instance, in the research project ARTIST [TBF+15, BBCI+13] the aim is to migrate existing software to the cloud by manipulating the reverse engineered software models according to a set of migration objectives. However, the space of potential model optimizations is very large and the optimization has been identified as very challenging.

3. Design Evaluation. The artifacts developed in the course of this thesis are evaluated by applying the overall approach and single artifacts in a number of well-defined and representative case study setups. More precisely, initial experiments have been conducted using very simple case studies for single artifacts. Iteratively, the complexity of the case studies have been increased to evaluate the boundaries of the proposed approach in terms of objectives and complexity of model transformations. All experiments have been documented in a structured way to enable re-execution and the reproduction of the results.

4. Research Contributions. The presented generic optimization process as well as the presented artifacts implemented for this process constitute the contributions of this thesis. All conclusions drawn from developing the artifacts and running different experiments have been documented and added to the knowledge base of the MDE and SBSE community. Details about the contributions have been outlined in the previous section and are further described in the following chapters.

5. Research Rigor. Research exists about different model-based property analysis approaches and the application of search-based optimization methods on models. The provided research is taken into account when developing the optimization approach and the model-based analysis approach. Existing work combining SBSE and MDE is taken

into account when exploring different strategies to explore the transformation search space. Overall, intensive literature studies are carried out prior to designing the artifacts, and existing approaches and tools have been reviewed. Furthermore, evaluation methods applied by previous related research have been investigated for their applicability in evaluating the artifacts built within the thesis. The built artifacts have been contrasted and compared with the artifacts built in related research, specifically the application of different optimization methods have been compared using a number of statistical tests, as proposed by Arcuri and Briand [AB11].

6. Design as a Search Process. The artifacts developed in course of the thesis are iteratively developed and evaluated. These iterations have been driven by different problem case studies. This means that the artifacts have been first designed and built to support exemplary case studies. By doing so, design alternatives for the artifacts have been explored, the problem domain has becomes better understood, and experience has been gained. As a result, more complex case studies have been considered subsequently, and the artifacts have been further refined.

7. Communications of Research. The contributions of the thesis have been communicated through well-known publication venues in the MDE and SBSE community as well as in the broader software engineering community.

1.5 Structure of the Work

This thesis is structured according to the elaborated contributions. In the following, we provide an overview of this thesis by briefly describing the contents of each chapter. Some of the contributions of this thesis have already been published in peer reviewed workshops, symposia, and journals. Hence, the contents of these publications overlap with the contents of this thesis.

Chapter 2: Preliminaries. The aim of this chapter is to introduce the basic concepts on which the contributions are founded, namely MDE and search-based optimization. Specifically, we describe the metamodeling stack which defines how conformance constraints are specified on models, we explain how domain-specific modeling languages can be developed and illustrate how model transformations are used in the area of MDE. Second, we describe the fundamental aspects of search-based optimization. In particular, we introduce the different aspects of an optimization problem and give an overview of metaheuristic methods that can be used to solve such problems.

Chapter 3: Model-Based Property Analysis. In order to apply metaheuristic optimization methods on model level, a model engineer needs to specify objectives that should be optimized and constraints that need to be considered. Such objectives and constraints are given in the form of properties which are realized through dedicated model-based analysis techniques. This chapter provides an overview of such techniques by

explaining how properties can be categorized and by describing general ways of retrieving properties using MDE techniques. After the general overview, we introduce our novel approach that is able to analyze dynamic, non-functional properties that consider the contention of resources on model level. While many approaches accomplish this task by translating the model into a dedicated analysis formalism, our approach is different in the sense that it can analyze these properties directly using model execution traces. In this chapter, we describe our approach in detail and evaluate it for several performance properties. The approach has also been published in [FBL+13].

Chapter 3: Model-Based Property Analysis. Using model-based analysis techniques to retrieve properties as part of the objective and constraint specification, we have all necessary inputs to realize our model-level optimization approach in this chapter. Specifically, the approach, named MOMoT, is realized by formulating the transformation orchestration problem as a search problem through a generic encoding and elevate the necessary inputs to the model level. In this chapter, we describe all relevant parts of the realization in detail and exemplify them on a running example. Furthermore, we introduce the dedicated configuration language that provides the model engineer with additional information to use our approach. Finally, we outline how the approach is implemented as a framework to allow the application of our approach on concrete problem instances. The approach has been published in [FTW15, FTW16b].

Chapter 5: Evaluation. In this chapter, we provide an extensive evaluation of the MOMoT approach. Specifically, we focus on the applicability, the runtime overhead, and the search features of our approach and evaluate these aspects on four case studies. This part of the evaluation has been published in [FTW15, FTW16b]. Moreover, we use our approach to tackle two existing problems in the field of model-driven engineering. First, we tackle the problem of modularizing model transformations by formulating this problem as a search problem using MOMoT. For the results, we provide an extensive evaluation using both statistical analysis of the solutions and an evaluation based on user studies. The problem formulation and presented solutions are published in [FTKW16]. Based on the results retrieved from the model transformation modularization problem, we provide an approach to tackle the problem of generically modularizing concepts in modeling languages. This approach is demonstrated and evaluated using Ecore as modeling language. It has been published in [FTW16c].

Chapter 6: Conclusion and Future Work. Finally, in this chapter, we summarize the contributions of this thesis and discuss overall conclusions. In addition, we point out the limitations of the contributions and give directions to interesting lines of research for the future.

CHAPTER 2

Preliminaries

This chapter provides an overview of the two worlds that to combine in this thesis: model-driven engineering (MDE) and search-based optimization (SBO). First, we provide an introduction into the main concepts of MDE including metamodeling, the development of modeling languages, and an overview of model transformations in Section 2.1. These concepts are also applied in two simple examples to further consolidate their understanding. Second, we describe the main terminology and concepts used in SBO in Section 2.2. In particular, we demonstrate how an optimization problem can be formulated, how solutions to these problems are encoded, and which metaheuristic methods exist to tackle these problems.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [dS15] is a methodology that advocates the use of models as first class entities throughout the development life cycle or the problem solving process. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design and promoting communication between individuals and teams working on the system. In short, a model can be seen as an abstraction of a system or a real-world concept.

2.1.1 Metamodeling

Using models as the central artifacts in the development and problem solving process gives rise to the question of how these models can be defined. In MDE, the answer to this question is given by so called metamodels. A metamodel formally defines the structure, semantics and constraints for a family of models [MSUW04]. Thus a metamodel can be seen as a model that describes a set of models in the same way a model describes an abstraction of a system or reality [BCW12]. Interestingly, when we consider metamodels

to be models, we can create a metamodel to define a family of metamodels. This metamodel is called meta-metamodel and may itself be seen as a model. In theory, this process of abstraction can be repeated infinitely often. However, it has been shown that in practice the meta-metamodel already provides sufficient concepts to be define itself and that further abstractions do not yield a real benefit [BCW12]. As a result, we have the following four-layer metamodeling stack that is also implemented in most metamodeling tools.

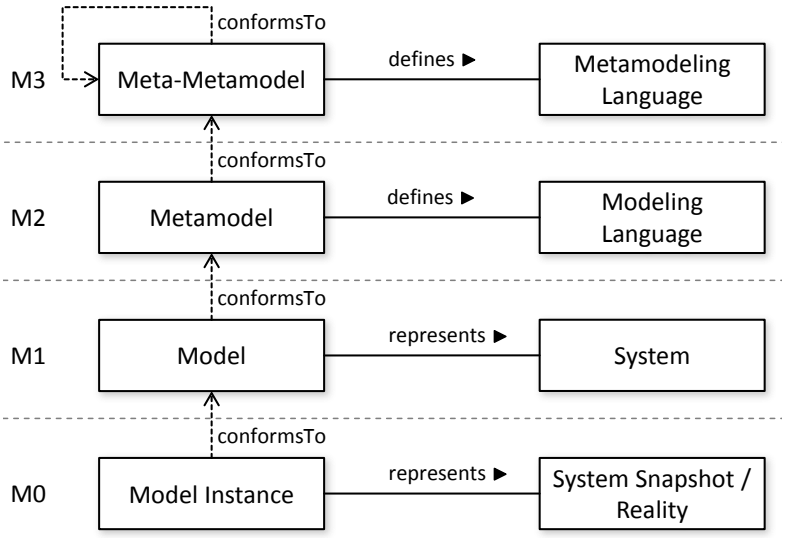


Figure 2.1: Four-Layered Metamodeling Stack (adapted from [BCW12, Küh06])

An overview of this stack is depicted in Figure 2.1 [BCW12, Küh06]. The upper half of this figure is concerned with language engineering, i.e., building models for defining modeling languages, while the lower half is concerned with domain engineering, i.e., building models for particular problem domains. Between each layer and its describing meta-layer, we assume a conformance relationship (*conformsTo*), i.e., a model on one layer has to fulfil all constraints given by the model one layer above. In terms of graph theory, a model can be seen as a graph consisting of nodes and edges and the model it conforms to as its type graph. An exception to this rule is the top-most layer, M3, which is defined upon itself and therefore needs to conform to itself. As inverse relationship to the conformance relationship, we define that a model is an *instance of* a metamodel, a metamodel is an instance of a meta-metamodel and a meta-metamodel is an instance of itself.

M3: Meta-Metamodel. A meta-metamodel defines a metamodeling language that can be used to define metamodels. Meta-metamodels are defined reflexively, i.e., a meta-metamodel must conform to itself. This means, that every element in the meta-metamodel can be expressed using the very same meta-metamodel. The standard metamodeling language defined by OMG is the Meta Object Facility (MOF) [OMG15b] language.

MOF is based on a core subset of UML class diagrams and provide concepts for classes, attributes, and references. It was standardized by the OMG in 1997 to ensure the interoperability of metamodels and modeling environments [BG01, May14]. Another prominent metamodeling language is Ecore, the meta-metamodel used in the Eclipse Modeling Framework (EMF). Ecore is also based on a subset of UML class diagrams, but was tailored towards Java implementations.

M2: Metamodel. A metamodel conforms to a meta-metamodel and defines a modeling language to describe models. In that sense, every element in the metamodel is an instance of an element in the meta-metamodel. From a language engineering point of view, the metamodel describes the abstract syntax of a modeling language (cf. Section 2.1.2), i.e., the language concepts, their attributes and the relationships among them. There is a plethora of different modeling languages available. Among the most prominent examples, we have the Unified Modeling Language (UML) [OMG15c] for system development, the Knowledge Discovery Metamodel (KDM) [OMG11a] for architecture-driven modernization, and the Common Warehouse Metamodel (CWM) [OMG03] to manage data warehouses, all of which are standardized by the OMG and defined with MOF.

M1: Model. A model describes instances of domain concepts which represent real-world entities and describe real world phenomena. The structure, semantics, and constraints of a model are given by the metamodel to which the model conforms.

M0: Model Instance. And finally, a model instance is an actual object or phenomena in the real world from which we abstract.

Conformance Relationship. Each model of a specific layer has to conform to the constraints defined by the respective meta-layer. By viewing models as graphs and type graphs, we are able to define constraints based on the node and edge structure of the graph. However, there are several constraints that cannot be defined using only graphical elements such as constraints based on objects and values of a concrete model. For instance, we cannot express that in UML the names of all parameters in a method signature must be unique. In order to express such constraints in MDE, we use a dedicated *constraint language*. The most prominent representative of these languages is the Object Constraint Language (OCL) [OMG14b]. OCL is a standardized, typed, and side-effect free language to describe expressions on any MOF-compliant models. The specified expressions can be used to define model queries, invariants on states, and pre- and post conditions of operations, among others. Only if all constraints are fulfilled we consider a model to be *valid*.

2.1.2 Developing Modeling Languages

In order to develop our own modeling language, we need to define the abstract syntax of the language using concepts of a meta-metamodel, a set of concrete syntaxes to express

instances of the language and language semantics. In a metamodel-centric language design approach [BCW12] a metamodel is used to define the abstract syntax to which we map from the concrete syntaxes and from which we map to the semantic domain.

In general, we can distinguish between general purpose languages and domain-specific languages. General purpose languages can be applied to any sector or domain [BCW12], e.g., UML [OMG15c], Petri-nets, or state machines. Domain-specific languages (DSLs), on the other hand, deal with the concepts of a concrete problem domain. DSLs tend to support higher-level abstractions than general-purpose modeling languages, and are closer to the problem domain than to the implementation domain. Thus, a DSL following the domain abstractions and semantics, allowing modelers to perceive themselves as working directly with domain concepts. Furthermore, the restrictions of the domain can be included into the language as constraints, disallowing the specification of illegal or incorrect models. DSLs enable domain experts to create models independently from the implementation platforms using their domain vocabulary. The development of a DSL is worthwhile if the language allows a particular type of problem or solution to be expressed more clearly than in an existing general purpose language and if the type of problem in question reappears sufficiently often.

Abstract Syntax. The abstract syntax of a modeling language defines the set of possible modeling concepts when creating a model. Each concept consists of specific attributes and the relations to other concepts. In MDE, metamodels are the standard means to specify the abstract syntax of modeling languages [Küh06].

Concrete Syntax. The concrete syntax of a language defines the notation used for representing the models that can be described with the language. Generally, we distinguish between a graphical and a textual concrete syntax. In a graphical syntax models are described in a diagrammatic manner, using symbols to represent their elements and relationships among them. For instance, UML offers rectangular boxes to represent classes and instances, and lines to represent associations and links. In a textual syntax models are described using sentences composed of strings, similar to programming languages. For instance, the OCL standard provides a textual concrete syntax for the expressions used in queries and constraints. It is important to note that all concrete syntaxes map to the same abstract syntax, i.e., they are defined upon the concepts available in the metamodel. Therefore, there is no restriction to the number of concrete syntaxes a single modeling language may have.

Semantics. Finally, the semantics is used to express the meaning of a model that can be created using the modeling language [HR04]. In that sense, the abstract syntax of a modeling language is mapped to a semantic domain. In MDE, two approaches for formally defining the semantics of a modeling language have been applied: translational semantics and operational semantics [CCGT09, Kle08, May14]. In the translational semantics approach, the semantics is defined by a translation from the modeling language to another language whose semantics is formally defined. In the operational semantics

approach, the semantics is defined by an interpreter that specifies the computation steps required for executing a model. Of course, the semantics of a modeling language may also be defined in a less formal way, e.g., using natural language. For instance, the semantics of OCL is described using UML and the semantics of UML is described in English prose as the semantic domain.

2.1.3 Model Transformations

In MDE where models are the central artifacts of, model transformations provide the essential mechanisms to manipulate models and are the key technique to automate different tasks [SK03, BCW12]. In fact, model transformations play such a crucial role in the process that they are considered *the heart and soul* of MDE [SK03].

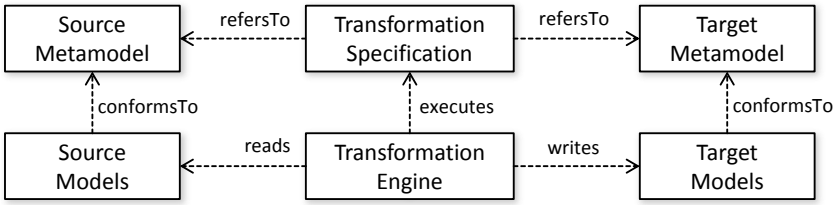


Figure 2.2: Model Transformation Pattern [CH06]

In general, a model transformation is considered a program that is executed by a transformation engine. The transformation engine takes one or more models as input and produces one or more models as output by executing the transformation specification, as illustrated by the model transformation pattern [CH06] in Figure 2.2. The input models, also referred to as source models, conform to a source metamodel, and the output models, also called target models, conform to a target metamodel. For endogenous transformations, the source and target metamodel are the same. It is important to note that a model transformation is defined on the level of the source and target metamodel and is therefore applicable on all valid source models.

Model Transformation Intents

Typically, a model transformation realizes a specific transformation intent, such as the refinement or the refactoring of a model, the merging of two models, or the translation of a model from one language to another. In this section, we give a short overview on the main intent categories that have been identified by Lúcio et al. [LAD+14].

Refinement. The aim of refinement is to add precision and detail to an existing model and reduce ambiguities. For instance, in the MDA approach for forward engineering [OMG14a], a platform-independent model might be refined into a platform-specific model by adding platform-related knowledge. Other examples for refinement include the generation of code called synthesis and the serialization of models for storing purposes.

Abstraction. Abstraction can be seen as the inverse of refinement, i.e., abstraction aggregates information or removes information from a model to produce a simpler or more high-level version of the source model. In this sense, model queries can also be seen as abstractions since they retrieve parts of the information contained in a model. Also reverse engineering falls under this category, e.g., the abstraction from a platform-specific model into a platform-independent model or the generation of UML class diagrams from source code. Another interesting abstraction is the approximation of model transformations, where a certain transformation is equivalent to another transformation up to a certain error margin.

Semantic Definition. As already described in Section 2.1.2, two approaches can be applied to define the semantics of a modeling language. In the translational approach we transform the concepts available in the source modeling language to a target language that has a clearly defined semantic domain. In the operational approach we provide the computational steps to interpret or execute the language. Each of these computational steps can be realized through a model transformation, where the source model is the current state and the target model is the following state after one step.

Language Translation. Under language translation we subsume the mapping of one modeling language to a target modeling language and the migration of a software model from one version of a language to another version of that language. For instance, the transformation from UML class diagrams to an equivalent relational database scheme is considered a translation whereas the transformation from UML class diagrams conforming to EJB2 to class diagrams conforming to EJB3 is considered a migration.

Constraint Satisfaction. In this category the aim of the transformation is to produce a set of output models given a set of constraints. Here, we can distinguish between model generation and model finding. In model generation we automatically generate models conforming to the constraints defined in the metamodel, In model finding we are interested in models that satisfy a given set of constraints, e.g., when exploring the design space of a system.

Analysis. When the intent of the transformation is to analyze the model, we are interested in certain properties or parts of the model. Numerous model analysis methods based on transformation exist which are considered research fields on their own, such as model-based property analysis or model checking, cf. Chapter 3.

Editing. Transformations in this category aim to manipulate the source model directly. In its simplest form, the transformation might just add, delete or modify elements in the model. More elaborate editing intents are the optimization of models or the refactoring of models. Optimization aims at improving operational qualities of a model such as scalability and efficiency, and in refactoring a model is restructured to improve its internal quality characteristics without changing its observable behavior [Fow99].

Also the reduction of the syntactic complexity of a model through normalization and canonicalization fall under the category of model editing.

Model Visualization. The aim of transformations in model visualization is to deal with the abstract and concrete syntaxes of a modeling language (cf. Section 2.1.2). In particular, we include the rendering of model elements in the abstract syntax into a concrete syntax and the parsing of a concrete syntax into its abstract representation in this category. Also transformations that visualize a simulation, i.e., animate the process, realize model visualization.

Model Composition. Model composition is dealing with the integration of individual models into a compound model. This is the case when several models are merged into one target model, when changes made in the individual models are synchronized in a common global model or when we define correspondence links between corresponding entities of several models.

Model Transformation Languages

In the last decade, many different model transformation languages have emerged in the MDE field [CH06, MG06]. In this section, we provide an overview on the classification of these languages based on the features defined by Czarnecki and Helsen [CH06]. In this overview, we focus on features and aspects that are relevant to this thesis, for details on all features, we refer the reader to the full article.

Major Categories. There are three major categories under which model transformations can be subsumed: model-to-text (M2T) transformations, model-to-model (M2M) transformations, or text-to-model (T2M) transformations.

Conceptually, there is not much difference between M2T and M2M transformations besides the target of the transformation. Whereas in M2T transformations the target is plain text, e.g., source code or documentation, the target in M2M transformations is a model conforming to a metamodel. We can divide M2T transformations into visitor-based approaches that traverses the internal representation of a model and template-based approaches where we have a textual template with special commands to incorporate information from the model. Available frameworks and languages include among others MOFM2T [OMG08], XPand [Ecl16e], Acceleo [Ecl16a], JET [Ecl11], and AndroMDA [And14].

In M2M transformations, we distinguish between direct-manipulation, structure-driven, operational, template-based, relational, graph transformation-based, and hybrid approaches. Direct manipulation approaches are the most low-level approach and usually provide just some API to manipulate the internal representations of models. Structure-driven approaches focus on the creation of the target models in two phases, by creating the hierarchical structure first and then setting attributes and references. Operational approaches are similar to direct manipulation approaches, but usually provide more

detailed support for model transformations. Template-based approaches facilitate the use of model templates with embedded metacode that computes the variable parts of the resulting target model. Relational approaches build upon the mathematical definition of relations and can be seen as a form of constraint solving. Graph-based approaches deal with models as typed, attributed, and labeled graphs whereas the metamodel is considered as the type graph. Approaches that do not clearly fit into any of the described categories or combine different aspects of the categories, are considered hybrid approaches In general, a M2M transformation where the source and target metamodel are the same is called *endogenous* or *rephrasing* whereas a transformation with different source and target metamodel is called *exogenous* or *translation*. There is a plethora of frameworks and languages to define M2M transformations, such as AGG [Tae03], Maude [CDE+07], AToM^3 [dLV02], e-Motions [RDV09], VIATRA [CHM+02], QVT [GK10], ATL [JK06], Henshin [ABJ+10], Kermeta [JBF11], and JTL [CDREP10].

And finally, T2M transformations have text as source and produce a model conforming to a metamodel. As such, all parsing approaches fall under this category.

Transformation Rules. In this thesis, we consider transformation rules as the smallest units of transformations in a transformation language. The logic of these rules, i.e., the computation steps and constraints, may be expressed using a certain paradigm such as object-orientation or a functional approach. In a *declarative* logic, we specify the relationship between the source and the target models without specifying the concrete computational steps that are necessary to produce one from the other. The actual computation is delegated to the transformation engine. In an *imperative* logic, we give an explicit sequence of computation steps that should be executed by the transformation engine. A merge of these two logics can be found in *hybrid* languages such as ATL where the rules can be defined declaratively, but imperative calls to rules or helpers are also possible.

Parameterization. In order to adapt the behavior of a transformation rule or pass information between rules, a transformation rule may have parameters. The simplest form of parameters are control parameters which pass simple flags to dictate how a rule should behave. If the parameter value can be any data type, including model elements, we refer to them as generic parameters. They may be used to provide values that can be used to select or adapt specific model elements or to pass contextual information between rules. If rules themselves can be used as parameters in a rule, we consider the rule to be a higher-order rule. Higher-order rules can be used to realize higher-order transformations (HOTs), i.e., transformations whose input/output are transformations [TJF+09, VP04, ALS08]. A HOT operates on a more abstract level and enables the generation of transformations from different information sources, the composition and decomposition of transformations, the analysis of transformations, and the modification of transformations [TJF+09].

Rule Application Conditions. In some approaches, transformation rules may have explicit conditions before they can be applied. For instance, in QVT, we can define a

when-clause that specifies a condition under which the specified relationship must hold. In Henshin, a graph-based approach, we have explicit positive application conditions (PACs) and negative application conditions (NACs) that define whether a specified graph pattern needs to be present or absent in order for the rule to be applicable.

Application Control. In order to apply a transformation on a model, two aspects have to be considered: scheduling and location determination.

Scheduling determines the order in which the rules are applied on the respective model and may be done implicitly or explicitly. In implicit scheduling, the user has no control over the actual scheduling process and the transformation engine takes care of the execution order of the rules. In such a case, the rules may be selected through an explicit condition or non-deterministically. In a transformation approach that supports an explicit scheduling, a user may influence the order in which rules are executed through an external mechanism or internally as part of the transformation rules. For instance, VIATRA allows the external scheduling of rules through finite state machines and ATL provides mechanisms to explicitly call a rule from another rule. As part of the scheduling mechanism, a rule iteration mechanism may be defined such as recursive rule iterations, looping concepts and fix-point iterations, i.e., applying a rule as long as it produces changes on the model.

In general, in order to check whether a rule is applicable or not, we need consider the current structure of the model and the application conditions of the rule. If a rule is applicable, we can produce a set of matches that specify in which parts of the model the rule may be applied. These parts of the model are also referred to as source or application *location* and these locations from the set of matches may be calculated deterministically, non-deterministically or interactively. In a deterministic strategy, we always select the location using the same technique, e.g., breadth-first traversal of the model elements. A non-deterministic strategy, on the other hand, chooses one location at random or applies the transformation to all matching locations concurrently. Concurrent application is supported in AGG, AToM^3, and VIATRA. In an interactive strategy, a user may be asked to select the location in the model on which the rule should be applied.

Rule Organization. This feature describes what mechanisms a transformation language provides to support the organization of rules into modules and the re-use of rule functionality. For instance, ATL, VIATRA, Henshin, and QVT provide an explicit concept of modules or packages in their language through which individual transformation units can be organized and imported. In order to re-use the functionality of transformation rules, similar concepts as in object-oriented programming may be applied, namely inheritance between rules, inheritance between modules, and logical composition.

Source-Target Relationship. We already briefly discussed the source-target relationship with respect to the type of artifact that is used as input or output, i.e., model or text. In M2M transformation, we also consider whether the source model and the target

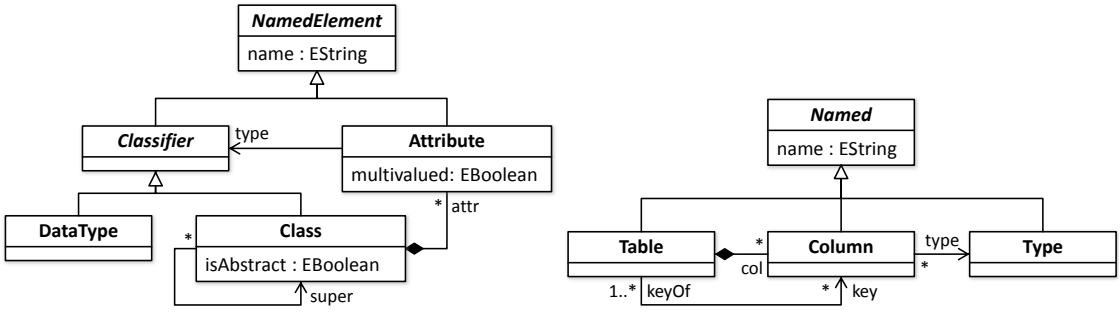
model are the same. If a M2M transformation creates new models from scratch, for instance when reverse-engineering code into models, it can be categorized as *out-place*. If a M2M transformation rewrites the input models until the output models are obtained, like in refactoring tasks, it is categorized as *in-place*. A transformation language may support one or both of these modes. For instance, the default mode in ATL is to create models from scratch, i.e., out-place transformations, but enables the execution of the rules as in-place by switching to the so called refining mode.

Directionality. For now, we have considered that a transformation takes a source model conforming to a source metamodel as input and produces a target model conforming to a target metamodel as output. In the sense of directionality, such a transformation is considered *unidirectional*, i.e., the execution proceeds from source to target. However, a transformation may also be *multidirectional*, i.e., executable from source to target and from target to source. Multidirectional behavior may be achieved using multidirectional rules provided by the language or by defining several complementary unidirectional rules.

2.1.4 Example: Class To Relational

In this section, we apply the concepts of metamodeling and model transformations that we have discussed in the previous sections on an example taken from the ATL Transformation Zoo [Ecl]. The goal of this example is to demonstrate the application of an out-place model transformation that translates class diagrams into relational models. We achieve this by (*i*) defining the respective metamodels, (*ii*) specifying the model transformation, and (*iii*) applying the transformation on a concrete input model.

Metamodels. As a first step to solve our problem, we define languages to express two involved problem domains, i.e., we need to develop a metamodel for class diagrams and a metamodel for relational databases. In order to represent class diagrams, we could use UML, however as the UML metamodel is quite large and for sake of simplicity, we define our own metamodel which may be considered as a core subset of UML class diagrams.



(a) Class Diagram Metamodel (b) Relational Metamodel

Figure 2.3: Class To Relational Example Metamodels

Figure 2.3a depicts the class diagram metamodel which will serve as source metamodel for the transformation. In this metamodel, a class diagram consists of a classes which have a name inherited from the abstract class `NamedElement`. Each class has a set of attributes which have a name, a type and can be defined as multi-valued attributes using a Boolean flag. Additionally, a class may have one or more super classes in order to express inheritance relations. Moreover, we support the concept of primitive data types in our language which opposed to classes are only specified by their name and do not have any attributes or relations. Figure 2.3b depicts the relational metamodel which will serve as target metamodel for the transformation. The relational model consists of a set of named tables which in turn consist of a set of columns. Each column as has a name and a type and may serve as a key column to identify entries in the table.

Model Transformation with ATL. In order to transform class diagram models into relational models, we need to convert each class to a table, each data type to a type, each single-valued attribute to a column and create a new table for multi-valued attributes as they cannot be represented naturally in a relational model. To define this transformation, we use the ATL language. The Atlas Transformation Language (ATL) [JABK08, Ecl15] is a hybrid model transformation language containing a mixture of declarative and imperative constructs in order to execute transformations both as out-place and as in-place. Listing 2.1 displays an excerpt of the transformation that generates a relational model from a class model. In the excerpt, we have included one imperative helper function and two declarative rules, so-called *matched rules* in ATL.

Listing 2.1 Excerpt of the Class To Relational ATL Transformation

```
 1: module Class2Relation;
 2: create OUT : RelationalMM from IN : ClassMM; -- referenced metamodels
 3:
 4: helper def : objectIdType : Relational!Type =
 5:   Class!DataType.allInstances()->select(e | e.name = 'Integer')->first();
 6:
 7: rule ClassAttribute2Column {
 8:   from
 9:     a : Class!Attribute (a.type.oclIsKindOf(Class!Class) and not a.multiValued)
10:   to
11:     foreignKey : Relational!Column (
12:       name <- a.name + 'Id',
13:       type <- thisModule.objectIdType)
14: }
15:
16: rule Class2Table {
17:   from
18:     c : Class!Class
19:   to
20:     out : Relational!Table (
21:       name <- c.name,
22:       col  <- Sequence {key}->union(c.attr->select(e | not e.multiValued)),
23:       key  <- Set {key}),
24:     key : Relational!Column (
25:       name <- 'objectId',
26:       type <- thisModule.objectIdType)
27: }
```

The first rule, named `ClassAttribute2Column`, translates elements of type `Attribute` from the class diagram metamodel whose `type` is `Class` and whose `multiValued` attribute is set to `false`. This is specified by the guard condition of the rule in Line 9. Elements satisfying this condition are translated into elements of type `Column` with a name set to the value of the respective attribute concatenated with the String *Id*, cf. Line 12. The element referenced by the `type` relationship of a column is retrieved by the helper function.

This helper function selects a type for the relational model that should be used for references. Here, we assume that a type with the name *Integer* is specified in the class model and always select this one as the usage of numeric values as ids is a common practice when defining relational models.

The second rule, `Class2Table`, takes an element of type `Class` as input and creates two output elements. The first element is of type `Table`, cf. Line 20, and gets the same name as the corresponding class element. As table columns, all columns created by the corresponding attribute elements are chosen together with the key column created the in the next step. The second element this rule creates is a dedicated key column for the table, cf. Line 24. This key column has a fixed name of *objectId* and its `type` is assigned with the helper function, i.e., Integer. In order to retrieve all columns for the table, no explicit call is necessary. Instead, ATL performs a transparent lookup of output model elements for given input model elements. Therefore, for each element of type `Class!Attribute`, ATL automatically retrieves the corresponding `Relational!Column` elements.

The other transformation rules to translate the remaining parts of the metamodels are defined in a similar way.

Applying the Transformation. While the model transformation is defined on the metamodel level, the execution is performed on the model level. Therefore, in order to apply our transformation, we define an input model that conforms to the class diagram metamodel, cf. Figure 2.4.

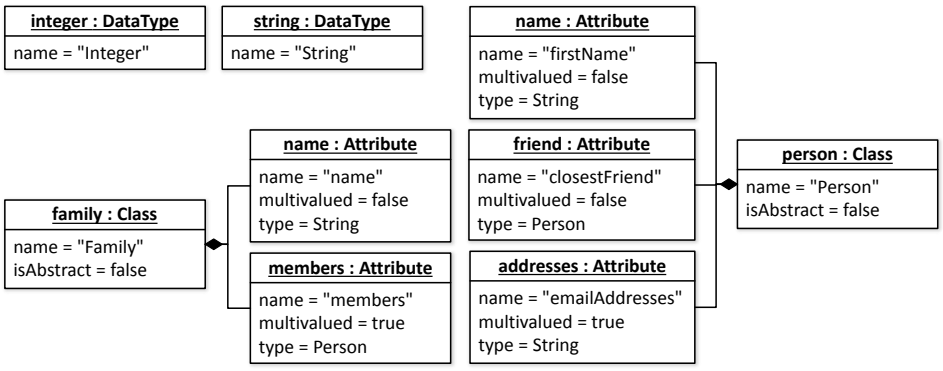


Figure 2.4: Source Model Conforming to the Class Diagram Metamodel

In this figure, we use the object diagram notation, i.e., the head of each box gives the unique identifier of the object and the objects type, i.e., the corresponding metaclass. The values for each object are given as key-value pairs in the body of the box. Links between the objects are given by the arrows and named like the corresponding relationships in the metamodel. For presentation purposes, links between attributes and their types are also shown as key-value pair in the body of the box.

In order to produce an output model where the elements are connected correctly, ATL uses an internal trace mechanism to lookup which elements have been created and how the source elements are connected. Every time a rule is executed, a new trace is created and stored in the internal trace model, as depicted in Figure 2.5. On the left-hand side of this figure, we have an input model and the right-hand side shows the model produced by the `Class2Relation` transformation. In the central part of the figure we can see the traces that have been produced during the execution of the two described rules. The traces keep track of which output elements are created from which input elements and by which rule. Thus, rule `ClassAttribute2Column` creates element `co1` from `at1` and produces `Trace 1`. Then, rule `Class2Table` creates elements `t1` and `co2` from `c1` and produces `Trace 2`. In order to properly set the column reference of the element `t1`, the engine searches in the trace model for the traces where attributes of `c1` serve as input. Therefore traces of type `Trace 1` are selected and the respective column elements are connected to `t1.col`.

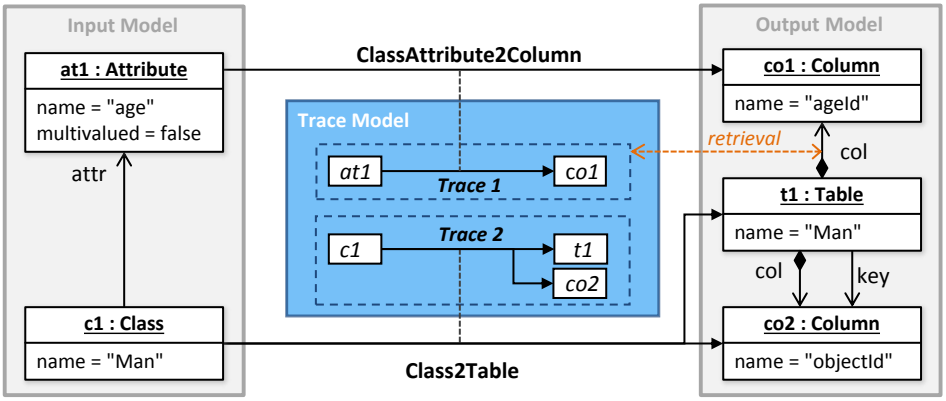


Figure 2.5: Excerpt of the Trace Model Used by ATL During Execution

When we apply the transformation on the input model defined in Figure 2.4, we create the output model as depicted in Figure 2.6. As expected, one table element has been created for each class element and for each multivalued attribute. Each table has been connected correctly with its corresponding columns and the correct key links have been set for the newly created `objectId` columns. Please note, that in the figure, attribute types are shown as values and not explicitly as links.

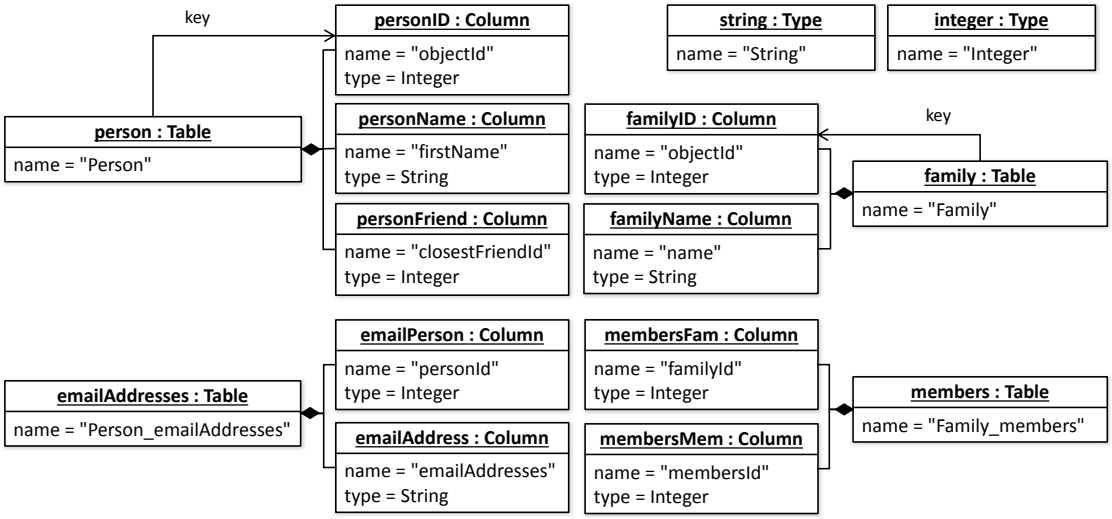
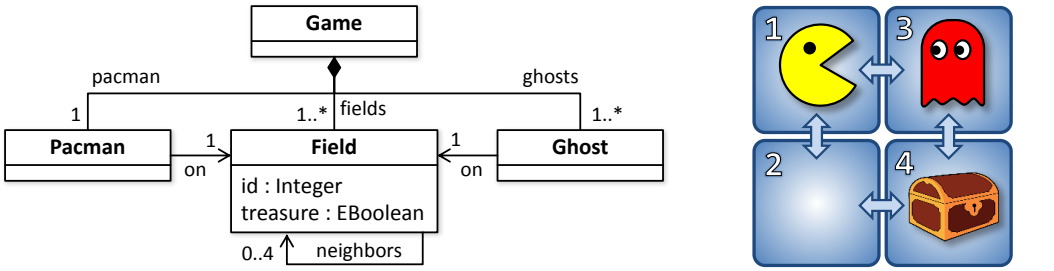


Figure 2.6: Target Model Conforming to the Relational Metamodel

2.1.5 Example: Pacman

In this section, we demonstrate and further consolidate the concepts of MDE with a second example. In particular, we show how an in-place model transformations can be applied to change the state of a model in order to play a game of Pacman. More specifically, we use two model transformations to move the ghost and the Pacman player around. A variation of this example has been published, for instance, in [Hec06] and in [BGKS14].



(a) Pacman Metamodel (b) Pacman Example Model

Figure 2.7: Pacman Game Specification

Metamodel. As opposed to the previous example, with in-place model transformations we only have a single metamodel to which both the source and the target model of the transformation need to conform. For this example, we model a variation of the well-known Pacman game as depicted in Figure 2.7a. Pacman is a game that is played on a board of several square fields which may be connected to each other on their horizontal and vertical axis, resulting in at most four neighbors per field. Each field has a unique `id`

and may or may not be the final treasure field, indicated by a Boolean flag. The game is played as the Pacman character and the goal is to get to the treasure field by moving around the board without encountering a ghost. In each game, there is at least one ghost which moves around the board. If the ghost catches the Pacman, i.e., moves to a field where the Pacman is located, then the game is over and the player has lost the game. If the Pacman finds the treasure field before being caught, the player wins. Consequently, a ghost is not allowed to move to a treasure field to prevent the player from winning. Figure 2.7b depicts an instance of the Pacman game with four fields, a treasure on field four, Pacman on field one, and a ghost on field three.

Model Transformations with Henshin. As a particular game of Pacman is realized as a model conforming to our game specification, we can use model transformations to implement the move operations. For this example, we use Henshin [ABJ+10] as model transformation language and engine. Henshin follows a graph-based transformation approach and offers a rich language and associated tool set for in-place M2M transformations of Ecore-based models. In Henshin, graphs are attributed, and nodes, edges, and attributes refer to `EClass`, `EReference` and `EAttribute` classes of the Ecore metamodel. Henshin comes along with a powerful declarative model transformation language that has its roots in attributed graph transformations and offers the possibility for formal reasoning.

The applicability of graph transformations for model transformations rests upon the fact that most models exhibit a graph-based structure, e.g., consider the underlying structure of UML class diagrams or state machines, whereas the metamodels of the models act as type graphs [Hec06]. The initial graph representing a model evolves through the application of graph transformation rules until the execution stops and we obtain the output graph, i.e., the output model. In general, a graph transformation rule $r = (L, R)$ consists of a name r, and a pair (L, R) whose structure is compatible, i.e., nodes with the same identity in L and R have the same type and attributes and edges with the same identity have the same type, source, and target. The left-hand side (LHS) L of a rule defines the graph patterns and conditions to be matched (preconditions) and the right-hand side (RHS) R describes the changes to be applied (effect, post-conditions). The LHS of a rule may have positive and negative application conditions (PACs and NACs), which specify the mandatory presence and absence of graph patterns before the rule may be applied.

In Henshin, the LHS and RHS of a transformation rule are combined into a single notation and their role is only given through dedicated keywords. The two rules for moving the Pacman and the ghosts within a game are depicted in Figure 2.8. These rules are called `movePacman` and `moveGhost` and are applied on the root element of our model, i.e., the Game instance, indicated by `@Game`. Each rule has two named and typed parameters, `p` and `f` for moving the Pacman and `g` and `f` for moving the ghost, to capture the Pacman and ghost element as well as the field to which they are moved. While these parameters are not necessary in this example, as we do not process the parameters any

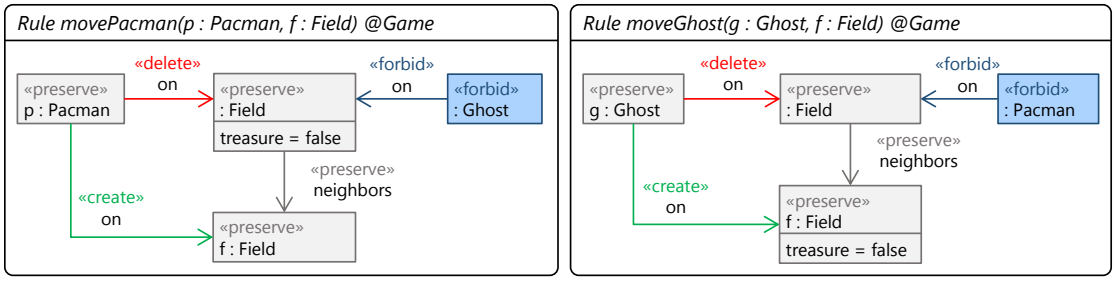


Figure 2.8: Rules to Move the Pacman and Ghosts

further, this is an important concept later when combining model transformations with search-based optimization methods. In Henshin rules, nodes and edges with the keyword *preserve* need to be matched in the underlying model graph before the respective rule can be applied. Negative application conditions, such that the Pacman may not move when he was already caught by a ghost or that a ghost does not move once it catches the Pacman, are marked with the keyword *forbid*. After a suitable match for the LHS of a rule has been found, the resulting changes applied on the graph for the RHS of the rule are denoted with *delete* to remove nodes and edges, and *create* to produce new nodes and edges. In our rules, we use these keywords to update the location relation `on` for the moved Pacman or ghost.

Applying the Transformation. When we apply a rule on a model, a graph transformation engine rewrites the given model graph by performing the following three steps [Hec06]:

(*i*) Find a match $m : L \mapsto M$ of the left-hand side L in the given model graph M.

(*ii*) Delete from M all nodes and edges matched by $L \setminus R$.

(*iii*) Paste to the result a *copy* of $R \setminus L$.

The result of these three steps is a derived model graph M' on which further rules may be applied. By performing these steps multiple times, we create a sequence of applied transformation rules that illustrate the progress of a single game of Pacman.

At this point, it should be noted, that finding matches of LHS rule patterns in graphs is a non-trivial task and corresponds to the subgraph isomorphism problem which is NP-complete in the worst case [Coo71]. In Henshin, this problem is tackled by using constraint solving technique [TKL13] where each node in the graph is considered a variable with a corresponding domain slot that is the solution space for that variable. Impossible solutions for each variable are then removed according to the given constraints. The constraints are extracted from the provided patterns and the metamodel and are categorized into different groups, e.g., type constraints, reference constraints or containment constraints.

To improve performance, the order in which variables and constraints are processed is determined heuristically on-the-fly based on the current graph. The final matches are constructed by locking variables to specific solutions after the impossible solutions have been removed.

2.2 Search-Based Optimization

This section provides an introduction to the concepts of search-based optimization. Search-based optimization methods can be categorized as metaheuristic methods that deal with large or even infinite *search spaces* in an efficient manner. Therefore these methods are often applied on problems where it is infeasible to use exact methods [ZTL+03]. In this section, we describe how a search-based optimization problem can be formulated, followed by an overview of optimization methods that can be applied to tackle these problems.

2.2.1 Optimization Problem

An optimization problem may be defined as a couple (X, f) where X is the set of solutions representing the search space, and $f : X \mapsto Z$ is the fitness function that assigns to each solution $x \in X$ in the search space a value that indicates the quality of the solution.

Decision Variables

A solution x to a given optimization problem can be encoded as a vector of n decision variables x_i in the search space X. This is formally defined in Equation 2.1 [CLV07], where T indicates the transposition of the column vector to a row vector.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{also written as} \quad x = \begin{bmatrix} x_1, & x_2, & \ldots, & x_n \end{bmatrix}^T \tag{2.1}$$

During the optimization process, an algorithm manipulates these decision variables to produce high quality solutions. Depending on the problem, a solution may be encoded in one of several ways. For instance, real-valued variables may be used to represent numeric values in a natural way, e.g., 4.0 or 3.14, binary variables can be used to represent individual choices, e.g., 1001 0100 1010, permutations can be used to represent order and scheduling problems, e.g., 4-0-1-3-2, and trees can be used to represent hierarchies and expressions [Had16b, Had16a].

Constraints

Depending on the problem, not all solutions representable in the search space X may be feasible. To express these restrictions, constraints on the solutions can be specified. These

constraints are expressed in the form of mathematical inequalities (cf. Equation 2.2) or mathematical equalities (cf. Equation 2.3) [CLV07].

$$g_i(x) \leq 0 \quad \text{where } i = 1, \ldots, m \tag{2.2}$$
$$h_j(x) = 0 \quad \text{where } j = 1, \ldots, p \tag{2.3}$$

A problem is said to be *overconstraint* if the number of equality constraints p is greater or equal to the number of decision variables, i.e., $p \geq n$ [CLV07]. In that case, no degrees of freedom are left for optimization. A solution satisfying all $(m + p)$ constraints is said to be feasible and the set of all feasible solutions defines the feasible search space Ω.

Fitness

In order to evaluate the quality of a solution, an objective or fitness function $f : X \mapsto Z$ maps a given solution from the search space X to an objective vector in the objective space Z. A metaheuristic approach uses these mappings to manipulate the decision variables of a solution in such a way that we reach good values in the objective space. The notion of good values relates to the direction of the optimization; typically, an objective value in the objective vector needs to be minimized or maximized. In order to have a uniform optimization direction for all objectives, a maximization objective can be easily turned into a minimization objective by taking its negative value and vice versa. In the remainder of this section, we assume a minimization problem.

A fitness function with k objectives on a solution x is defined in Equation 2.4. If the fitness function has more than one objective, we call it commensurable if all objectives are measured in the same unit and non-commensurable otherwise.

$$f(x) = \Big[f_1(x), \quad f_2(x), \quad \ldots, \quad f_k(x)\Big]^T \tag{2.4}$$

For many real-world problems, multiple *conflicting* objectives need to be considered in order to find desirable solutions. Two objectives are said to be in conflict, if the improvement of one objective leads to the worsening of the other and vice versa. Typical examples include quality versus cost or from the software engineering domain coupling versus cohesion.

For most metaheuristics, the evaluation of the fitness function is the most expensive part [Tal09]. If the computational complexity of the evaluation needs to be reduced, *surrogate functions* may be applied instead. A surrogate function does not calculate the exact value of the fitness of a solution but only approximates the value. This speeds up the time needed for the evaluation, but produces less accurate results in general.

The number of objectives that need to be optimized, i.e., the size of the objective vector, is also used to categorize optimization problems, as described in the next wo sections.

Single-Objective Optimization Problem

A single-objective or mono-objective problem deals with one objective and can be defined as follows:

$$\begin{cases} Min\, f(x) & \\ g_i(x) \geq 0 & i = 1, \ldots, m; \\ h_j(x) = 0 & j = 1, \ldots, p; \\ x_l^L \leq x_l \leq x_l^U & l = 1, \ldots, n; \end{cases}$$

In this formulation, we have one objectives that needs to be minimized, m inequality constraints, p equality constraints, and the lower and upper bounds of the decision variable x_l defined by x_l^L and x_l^U. A solution x consists of a vector of n decision variables which are optimized by the metaheuristic algorithm. The objective value for a specific solution is calculated by the provided objective function f.

Having only a single objective to optimize, solutions can be compared with each other using their respective fitness values. We can define, that a solution $x_1 \in X$ is better than another solution $x_2 \in X$, if $f(x_1) < f(x_2)$ in case the aim is to minimize the objective or if $f(x_1) > f(x_2)$ in case the aim is to maximize the objective. Using this notion of total order, we can define a globally optimal solution, i.e., a solution that is better than all other solutions.

Definition 1: Global Optimum. A solution $x^* \in \Omega$ is a global optimum if it is has a better objective function than all solutions of the search space, that is,

$$\forall x \in \Omega : f(x^*) \leq f(x)$$

In single-objective optimization problems, the goal of a metaheuristic approach is to find one of the globally optimal solutions.

Multi-Objective and Many-Objective Optimization Problem

A multi-objective problem (MOP) deals with at least two objectives and can be expressed as [MKS+15]:

$$MOP = \begin{cases} Min\, F(x) = [f_1(x), f_2(x), \ldots, f_k(x)]^T & k > 1 \\ g_i(x) \geq 0 & i = 1, \ldots, m; \\ h_j(x) = 0 & j = 1, \ldots, p; \\ x_l^L \leq x_l \leq x_l^U & l = 1, \ldots, n; \end{cases}$$

In this formulation, we have at least two objectives k that need to be minimized, m inequality constraints, p equality constraints, and x_l^L and x_l^U corresponding to the lower and upper bounds of the decision variable x_l. A solution x consists of a vector of decision

variables which are optimized by the metaheuristic algorithm. The objective value for a specific solution is calculated by the provided objective function f_i and the aggregation of all objective functions defines the fitness function f.

Recently, due to the limits of how many objectives different algorithms can handle, a distinction is made between multi-objective problems and *many-objective problems*. A many-objective problem, as opposed to a multi-objective problem, is a problem with at least four objectives, i.e., $k > 3$ [1] [DJ14].

As opposed to the single-objective case, for multi-objective problems defining an order between solutions in order to compare them is more complicated. There are four common strategies to deal with this issue [Tal09]. Scalar approaches transform the multi-objective problem into a mono-objective problem by converting all objectives into a single objective. Criterion-based approaches treat each non-commensurable objective separately. Dominance-based approaches use the concepts of Pareto dominance and Pareto optimality to deal with all objectives at the same time. And finally, indicator-based approaches [ZK04] use dedicated quality indicators that can be seen as an extension to the Pareto dominance relation and that compare two set of solutions and produce a single quality value.

Scalar Approaches. One way to create an order between solutions is to use scalarization, i.e., the conversion of all objectives into a single objective through a scalarization function, e.g., a weighted sum function as depicted in Equation 2.5 [Tal09, HM79] with weights λ:

$$F(x) = \sum_{i=1}^{n} \lambda_i f_i(x), \text{ for } x \in X \text{ where } \lambda_i \in [0 \dots 1] \tag{2.5}$$

However, this approach can only be used if all objective values are defined on the same scale. If this is not the case, the objectives need to be *normalized*, cf. Equation 2.6 [Tal09] where f_i^{max} and f_i^{min} are the upper and lower bounds of the respective objective f_i.

$$F(x) = \sum_{i=1}^{n} \lambda_i \frac{f_i(x) - f_i^{min}}{f_i^{max} - f_i^{min}} \tag{2.6}$$

Other methods for scalarization include the definition of an aspiration or reference point and minimizing the distance to this point. In any case, scalarization only produces optimal results if the solution space is convex. Otherwise, no matter what weights have been chosen, some solutions may never be reached [Tal09, CLV07, CD08].

[1] In the remainder of this thesis, we refer to multi-objective and many-objective problems simply as multi-objective problems. If a distinction between multi-objective and many-objective problems is necessary, it is mentioned explicitly.

Criterion-based Approaches. Criterion-based approaches treat each non-commensurable objective separately, either by parallelizing the search by splitting the population into sub-populations for each objective or by using a lexicographic approach. In a lexicographic approach, the decision maker defines a priority between the objectives, giving them a total order [Fis74]. In the first step, the most significant objective is optimized. If only one solution is found in that step, then that solution is considered the optimum. If more than one solution is found, the second most significant objective is optimized for these solutions. If only one solution is returned, that solution is the optimum. Otherwise, the next objective is optimized and so on.

Dominance-based Approaches. Dominance-based approaches consider all objectives at the same time by using the concept of partial ordering over a dominance relation. The most common adopted dominance relation, is the Pareto dominance relation [CLV07] which builds upon the concept of Pareto optimality proposed by Francis Ysidro Edgeworth [Edg81] in 1881 and generalized by Vilfredo Pareto [Par96] in 1896. The Pareto dominance relation is defined as follows [Tal09].

Definition 2: Pareto Dominance. An objective vector $u = (u_1, \ldots, u_n)$ is said to dominate another vector $v = (v_1, \ldots, v_n)$, denoted by $u \prec v$, if and only if no component of v is smaller than the corresponding component of u and at least one component of u is strictly smaller, that is,

$$\forall_{i \in \{1,\ldots,n\}} : u_i \leq v_i \wedge \exists_{i \in \{1,\ldots,n\}} : u_i < v_i$$

Under Pareto optimality, it is impossible to find a solution that improves the value of one objective without decreasing the value of at least another objective.

Definition 3: Pareto Optimality. A solution $x^* \in \Omega$ is Pareto optimal if for every $x \in \Omega$, $F(x)$ does not dominate $F(x^*)$, that is,

$$F(x) \not\prec F(x^*)$$

A solution that is Pareto optimal is also called non-dominated. Each non-dominated solution can be considered an optimal trade-off between all objectives. However, there may exist several optimal trade-offs between the individual objectives. Therefore, a problem may yield a set of Pareto-optimal solutions, often denoted as the Pareto optimal set.

Definition 4: Pareto Optimal Set. The Pareto optimal set for a set of feasible solutions Ω is defined as

$$P^* = \{x \in \Omega \mid \nexists x' \in X : F(x') \prec F(x)\}$$

The corresponding objective values from the Pareto optimal set of solutions is referred to as Pareto front.

Definition 5: Pareto Front. For a Pareto optimal set P^* the Pareto front is defined as the objective vectors for all solutions in that set, that is,

$$PF^* = \{u = F(x) \mid x \in P^*\}$$

The aim of a metaheuristic method is to find those solutions that yield the true Pareto front for a given problem. It should be noted that we assume that the 'true' Pareto front of a problem PF_{true}, i.e., the subset of values which are all Pareto optimal, is impossible to derive analytically and impractical to produce through exhaustive search [HT07]. Therefore, each set produced using metaheuristic search is an *approximation* set to this, often unknown, 'true' Pareto front. A good approximation therefore aims to have objective values that are spread uniformly across the Pareto front using the limited number of solutions.

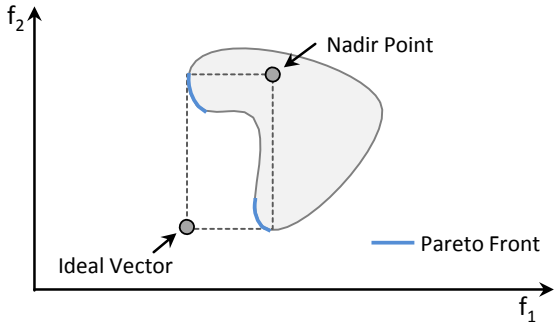


Figure 2.9: Ideal Vector and Nadir Point in a MOP (adapted from [Tal09])

Some information about the ranges of the Pareto font may be given through the ideal vector and the Nadir point, cf. Figure for a bi-objective problem 2.9 [Tal09].

Definition 6: Ideal Vector. A point $y^* = (y_1^*, \ldots, y_n^*)$ is an ideal vector if it minimizes each objective function f_i in $F(x)$, that is,

$$y_i^* = \min(f_i(x)) \text{ where } x \in X, i \in [1, n]$$

As such, the ideal vector does not need to be a feasible solution as it considers each objective separately. If the optimal objective values are not known in advance, the ideal vector can be substituted by a reference vector given by a decision maker. This reference vector gives an acceptable or desirable value for each objective.

Definition 7: Nadir Point. A point $y^* = (y_1^*, \ldots, y_n^*)$ is the Nadir point if it maximizes each objective function f_i of $F(x)$ over the Pareto set, that is,

$$y_i^* = \max(f_i(x)) \text{ where } x \in P^*, i \in [1, n]$$

It is important to note that by using Pareto dominance, we can determine whether one solution is better than another, but not measure by *how much* it is better. Therefore, the final solution needs to be selected by the decision maker.

Indicator-Based Approaches. Indicator-based approaches define a quality indicator that are used to guide the search. Unary indicators are functions that assign to each Pareto front approximation a scalar value, i.e., $I : \Omega \mapsto \mathbb{R}$, where Ω is the space of all Pareto front approximations. In order to evaluate whether one approximation A is better than another approximation B, binary indicators can be used, i.e., $I : \Omega \times \Omega \mapsto \mathbb{R}$ [HJ98]. Using binary indicators, the optimization goal can be formulated as in Equation 2.7, where R is a known reference set for the Pareto front.

$$\arg\min_{A \in \Omega} I(A, R) \tag{2.7}$$

If a reference set cannot be given in advance, a unary indicator can be used instead.

2.2.2 Metaheuristic Methods

In order to solve mono- or multi-objective problems, metaheuristic methods may be applied. Metaheuristic search methods can be defined as upper level general methodologies that can be used as guiding strategies in designing underlying heuristics to solve specific optimization problems. As such, metaheuristics can be seen as one family of methods to solve optimization problem. Figure 2.10 gives an overview of how metaheuristics are related to other main families of classical optimization methods [Tal09].

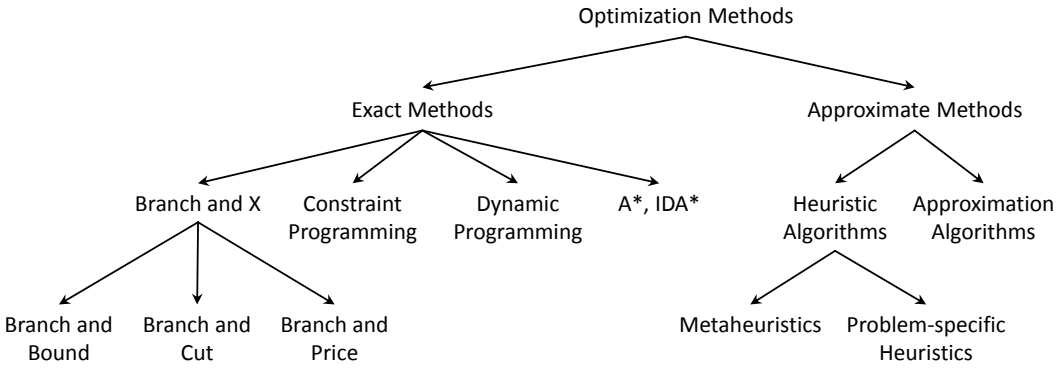


Figure 2.10: Metaheuristic Optimization in Classical Optimization Methods [Tal09]

Depending on the complexity of a given problem, an exact or approximate method can be used. In general, exact methods can obtain optimal solutions for a given optimization problem and guarantee their optimality. Dynamic programming, branch and bound algorithm, the A* family of search algorithms, and constraint programming are all in the class of exact methods. However, if the problem is complex, i.e., NP-complete, an exact method needs at least exponential time to produce optimal solutions [Tal09] and may only be applicable to a small instance of the problem. Such time constraint is not feasible for most real world problems in which the problem instances may be quite large. In these cases, approximate methods can be used. Approximate methods are able to generate high-quality solutions in a reasonable time, but cannot guarantee the optimality of these

solutions. In the class of approximate methods we can distinguish between approximation algorithms and heuristic algorithms. Approximate algorithms provide solutions with a provable but not optimal quality and within provable run-time bounds. Heuristics, on the other hand, do not provide any guarantee on quality of the solutions, but are able to handle large-sized problem instances. Heuristics may further be divided into specific heuristics that are tailored and designed to solve a specific problem or problem instance and metaheuristics. At this point, it is interesting to note that there is no agreed upon definition of the term metaheuristic [BR03] and some literature uses the terms heuristic and metaheuristics interchangeably [Yan10].

At its core, a metaheuristic is a general-purpose algorithm that is applicable to most optimization problems and may incorporate domain-specific knowledge in the form of specific heuristics to solve the problem at hand. The goal of metaheuristics is to explore the search space efficiently in order to find (near-)optimal solutions [BR03]. In this search, two contradicting criteria need to be considered: the diversification of the search space (*exploration*) and the intensification of the best solutions found (*exploitation*) [Tal09]. In exploration, the metaheuristic aims to cover as much area of the search space as possible while in exploitation promising areas are searched more thoroughly to find better solutions. For instance, random search can be considered at the exploration end of this spectrum as new candidate solutions are generated in each iteration without considering how good already found solutions are. On the other, local search algorithms (cf. Section 2.2.2) are closer to the exploitation end of the spectrum as they start with one solution that is iteratively improved.

Classification of Metaheuristics

In order to classify metaheuristic methods, many different criteria may be used. In the following, we give a summary of the common criteria defined in [CLV07, Yan10, Tal09].

Deterministic versus Stochastic. A deterministic metaheuristic solves an optimization problem by making deterministic decisions, i.e, in a given situation the method always chooses the same option. This means that for the same problem instance, a deterministic metaheuristic will always produce the same solutions. A stochastic metaheuristic, on the other hand, solves an optimization problem by making decisions with a certain degree of randomness. Therefore, for the same problem instance, a stochastic metaheuristic will produce different solutions every time it is executed.

Nature Inspired versus Non-Nature Inspired. Most metaheuristic algorithms are inspired by nature [Tal09, Yan10], e.g., the evolution process, the behavior of animals or physics. As such we have algorithms like evolutionary algorithms [Hol92], ant colony optimization [Dor92], particle swarm optimization [KE95], artificial bee colony [See95, YK96], cuckoo search [YD09] or simulated annealing [KJV83]. Of course, metaheuristics which have not been inspired by nature do also exist, such as Tabu Search [Glo86] and Hill Climbing.

Memory Usage versus Memoryless. An important feature of metaheuristics is the use of memory, i.e., the storage of dynamic information collected during the search process. For instance, in Tabu Search [Glo86] we may use different memories to remember the recently visited solutions (short term), focus or prohibit moves towards solutions with specific characteristics (intermediate) or even reset the search when we get stuck on a plateau, i.e., no improvements have been made for a a certain amount of time (long term). However, there are also memoryless metaheuristics that do not use any dynamic information from the search. They perform a Markov process as in each step they only use the current state of the search as information to decide the next action. For instance, basic local search only considers the neighbourhood of the current solution in order to move through the search space.

Dynamic versus Static Objective Function. Metaheuristics may also be classified according to how they use the fitness function. Most metaheuristics use the fitness function exactly as it is given in the problem formulation. However, other methods such as Guided Local Search are also able to incorporate dynamic information collected during the search into the fitness function during the search process in order to escape from local optima.

Population-Based versus Single Solution-Based. A very common criterion to categorize metaheuristics is the number of solutions they handle at a given time. Single solution-based methods, also referred to as trajectory methods [BR03], aim to optimize a single solution at a time and are therefore more exploitation-oriented [Tal09]. Examples include Tabu Search, Iterated Local Search, or Simulated Annealing. Population-based methods deal with a set of solutions, called a population, at a time and as such cover larger areas of the search space and are more exploration-oriented. Examples include particle swarm optimization or evolutionary algorithms.

Iterative versus Greedy. In order to manipulate solutions, a metaheuristic may either follow an iterative approach or a greedy method. Greedy methods, also called constructive methods, start with an empty solution and add decision variables step-by-step to the solution. These methods are usually the fastest approximation methods but often yield low quality solutions [BR03]. In an iterative approach, we start with a complete solution and aim to replace this solution with a better solution through manipulation in each iteration. The necessary manipulations are realized in terms of search operators (cf. Section 6 for the operators used in evolutionary algorithms). Most metaheuristics follow an iterative approach [Tal09].

In order to gain a more insight into the way metaheuristic algorithms work, we have a closer look at two major categories: local search as a representative of non-nature inspired, single solution-based methods, and evolutionary algorithms [Hol92] as representatives of nature inspired population-based methods.

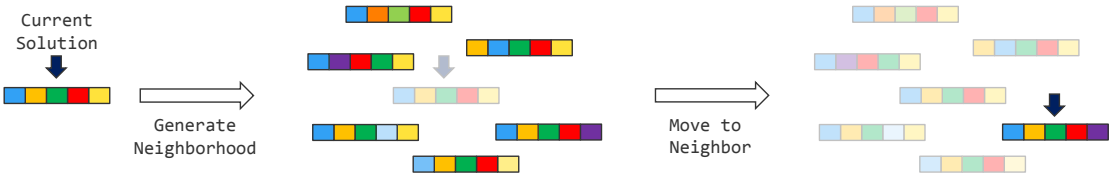


Figure 2.11: Two Steps Performed by Local Search Methods in Each Iteration

Local Search

Local search methods start with an initial solution and then perform two steps in each iteration: generation and moving. The local search process is shown in Algorithm 2.1 [Tal09] and visually depicted in Figure 2.11. In the first step, a set of candidate solutions called a neighborhood is generated from the current solution. In the second step, a solution from the generated neighborhood is selected to replace the current solution. The whole process iterates until the given stopping criteria is fulfilled, e.g., a specific solution is found, a solution with sufficient quality is found or the algorithm has run a given number of iterations.

Algorithm 2.1: High-level Process of Local Search [Tal09]

Input: Initial Solution s_0
Output: Best Solution Found

1: $t = 0$
2: **repeat**
3: $\quad Generate(N(s_t))$ `// Generate partial or complete neighborhood`
4: $\quad s_{t+1} = Select(N(s_t))$ `// Select a neighbor to replace current solution`
5: $\quad t = t + 1$
6: **until** *Stopping criteria satisfied*

Neighborhood. This set of candidate solutions that is generated from the current solution is called a neighborhood and should consist of solutions that represent a small move from the current solution, i.e., a small change in the representation of the current solution. If this is the case, we say that the neighborhood has a strong locality. If this is not the case, i.e., when the neighborhood has a weak locality, then the algorithm may move from the current solution to a solution that is very different from the current one and in the extreme case converges towards random search.

Definition 8: Neighborhood Structure. Formally, a neighborhood structure is a function $\mathcal{N} : \mathcal{S} \mapsto 2^{\mathcal{S}}$ that assigns to each solution s of the search space $\mathcal{S}$ a set of neighbor solutions $\mathcal{N}(s) \subseteq \mathcal{S}$. $\mathcal{N}(s)$ is called the neighborhood of s and each solution $s' \in \mathcal{N}(S)$ is called a neighbor.

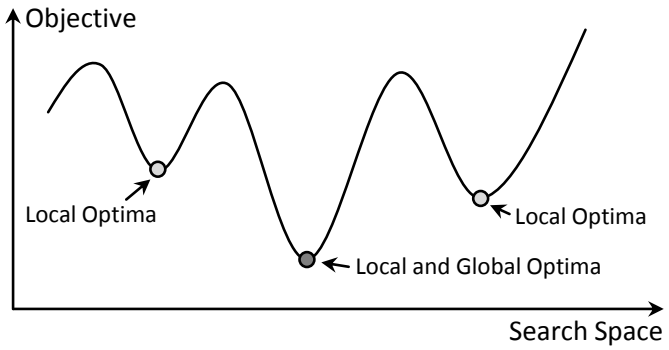


Figure 2.12: Local Optimum and Global Optimum in a Search Space (from[Tal09])

Local Optima. Having the concept of a neighborhood of a solution allows us to think of the concept of local optimality in contrast to global optimal solutions in a search space.

Definition 9: Local Optimum. Formally, a locally optimal solution with respect to a neighborhood function $\mathcal{N}$ is a solution $\hat{s}$ s.t. $\forall s \in \mathcal{N} : f(\hat{s}) \leq f(s)$. Furthermore, we call $\hat{s}$ a *strict* locally minimal solution if $\forall s \in \mathcal{N} : f(\hat{s}) < f(s)$.

The relation between local and global optima is depicted in the fitness landscape of Figure 2.12 [Tal09] for a minimization problem. In a search space, many neighborhoods with different local optima may exist, however only a subset of these local optima are actually global optima. In many cases, there is even only one globally optimal solution. Knowing the fitness landscape can help to select an algorithm that is able to not get stuck in a local optimum. For instance, in Hill Climbing or Gradient Descent [RN09], we start with an initial, random solution in the search space and always select a neighbor with a better fitness than the current solution. As such, depending on where in the fitness landscape we start, we may find a local optimum or even a global optima. However, once a local optimum is found, there is no way of exploring other parts of the search space and therefore there is no guarantee to find a global optimum.

Therefore many different local search algorithms have been proposed. These algorithms differ in whether they use a memory or are memoryless, in how they generate the initial solution, in how they generate the neighborhood and in how they select the next solution. In general, there are two ways to generate an initial solution, either we generate it randomly or we use a greedy approach. The definition of the neighborhood depends strongly on the representation associated with the problem at hand, e.g., in a binary encoding a neighborhood may be defined with a Hamming distance of 1, i.e., one bit-flip. How the resulting neighborhood is searched is also up to the algorithm. For large neighborhood, heuristic search or exact search is possible. Finally, the algorithm needs to select a neighbor to replace the current solution. This may be done by only accepting solutions with a better fitness value or by accepting also worse solutions. For instance, Simulating Annealing accepts worse solutions more likely in the beginning but less likely

as the search continues in order to provide a more explorative search in the beginning but intensify the search process later on.

Evolutionary Algorithms

Evolutionary algorithms [Hol92] are stochastic, population-based algorithms that build upon the Darwinian principles of evolution [Dar59], i.e., the struggle of individuals to survive in an environment with limited resources and the process of natural selection. In the 1980s, different approaches to incorporate this process into algorithms have been proposed. For instance, Genetic Algorithms [Hol62, Hol75], Evolution Strategies [Rec65, Rec73], Evolutionary Programming [Fog62, Fog66], or Genetic Programming [Koz92]. Nowadays, some widely used evolutionary algorithms include the Strength Pareto Evolutionary Algorithm 2 (SPEA2) [ZLT01], the Non-Dominated Sorting Genetic Algorithm-II [DAPM02] (NSGA-II), and the NSGA-III [DJ14]. Evolutionary algorithms are the most studied population-based algorithms [Tal09] and the field of Evolutionary Multi-objective Optimization (EMO) is considered one of the most active research areas in evolutionary computation [DJ12].

Algorithm 2.2: High-level Process of Evolutionary Search [Tal09]

Output: Best Solutions Found
1: $Generate(P(0))$
2: $t = 0$
3: **while** *not termination_criterion(P(t))* **do**
4: $\quad Evaluate(P(t))$
5: $\quad P'(t) = Selection(P(t))$
6: $\quad P'(t) = Reproduction(P'(t))$ `// Recombination and Mutation`
7: $\quad Evaluate(P'(t))$
8: $\quad P(t+1) = Replace(P(t), P'(t))$
9: $\quad t = t + 1$
10: **end**

The high-level process of an evolutionary algorithm is shown in Algorithm 2.2. In general, evolutionary algorithms start with a population of individual solutions. This initial population is often randomly generated or may be produced in some other form. Every individual in the population is a solution with a specific fitness assigned by the fitness function. In order to manipulate the population towards good areas of the search space, evolutionary algorithms typically use three search operators and a replacement scheme. The algorithm stops when the defined stopping criterion is satisfied, e.g., a specific number of iterations or evaluations have been performed or no improvement has been achieved for a given number of iterations.

Selection. The first search operator, called the *selection operator*, chooses individuals from the population and selects them for reproduction. Here, different selection strategies

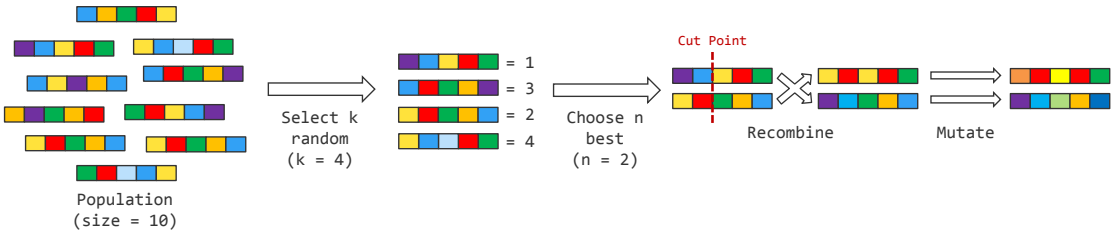


Figure 2.13: Steps Performed by Evolutionary Search Methods in Each Iteration

may be applied, e.g., based on absolute or relative fitness of a solution. In any case, the idea is that solutions with a higher fitness are more likely to reproduce. For instance, in deterministic tournament selection illustrated in Figure 2.13, we select k random solutions from the current population and choose the best n solutions for recombination. In non-deterministic tournament selection, the solution with the highest fitness is selected with a given probability p, the next best solution is selected with probability $p * (1 - p)$, the next best with $p * (1 - p)^2$ and so on. Another selection strategy is the roulette wheel selection where the chance of each individual solution is proportionate to its fitness value, i.e., in a population with N solutions, the chance to be selected is $p_i = {}^{f_i}/_{\sum_{j=1}^{N} f_j}$. The degree to which better individuals are favoured is called *selection pressure* and it is used to regulate the algorithms convergence [MG95]. If the selection pressure is too high, randomly generated solutions with a higher fitness at the beginning strongly shape the solutions found at the end and the algorithm may converge too early to suboptimal solutions (premature convergence). If the selection pressure is too low, the convergence towards optimal solutions may take an unnecessary long amount of time.

Reproduction. In the reproduction phase, we apply variation operators on the selected solutions in order to produce new solutions. Here, we use the terms parent solution and child solutions or parent population and offspring population. The most common variation operators are the recombination, or crossover, operator and the mutation operator.

The *recombination operator* is a n-ary operator that takes n parent solutions and produces n child solutions. The idea is that the characteristics that make a solution good can be given to its children and if two good solutions are (re-)combined an even better solution may emerge. The most basic recombination strategy is the one-point crossover, where two parent solutions are cut at the same random position and the children are created by a crosswise merge of the resulting parts. This strategy is also depicted in Figure 2.13. In constraint optimization problems, such an operator might produce solutions which are no longer valid. This must be considered when choosing a recombination operator. For instance, the partially mapped crossover (PMX) [GL85] can preserve the order of decision variables within an encoding. Common recombination operators beside the PMX and the one-point crossover include the geometrical crossover [MNM96], the unimodal normal distribution crossover (UNDX) [OKK03], the simplex crossover (SPX) [TYH99], the

simulated binary crossover (SBX) [DA95], the parent-centric crossover (PCX) [DAPM02], and the order crossover (OX).

The *mutation operator* is an unary operator that introduces small, random changes into a single solution. The main idea behind mutation is that it guides the algorithm into areas of the search space that would not be reachable through recombination alone and avoids the convergence of the population towards a few elite solutions. The mutation in evolutionary algorithms is related to neighborhood operators of single solution-based algorithms and some of the neighborhood operators may be reused as mutation operators [Tal09]. For instance, in a binary encoding, a mutation may be the flip of one bit in a solution. In general, the mutation probability p_m for each variable of a single solution should be rather low to not generate entirely new solutions and negate the positive effect of recombination.

Replacement. As the population size is constant, after selection and reproduction, we need to choose which solutions from the parent population and the offspring population should be kept for the next iteration. The two extreme cases are the replacement of the complete parent population with the offspring population and the replacement of only one individual in the parent population, e.g., the worst individual, through the best individual from the offspring population. Of course, in practice, any number of individuals may be replaced. Another replacement strategy is elitism, i.e., the selection of only the best solutions from both populations. This leads to faster convergence, but may result in premature convergence.

After giving an overview of the specifics of two families of metaheuristic methods, we can see that selecting and configuring an algorithm is not an easy task and highly depends on the respective optimization problem. Furthermore, it should be noted that no metaheuristic performs uniformly better than any other metaheuristic on all optimization problems, cf. the *no free lunch theorem* for optimization by Wolpert and Macready [WM97].

CHAPTER 3

Model-Based Property Analysis

In this chapter, we describe the first contribution of this thesis, an analysis approach for dynamic, time-based non-functional properties that considers the contention of resources directly on model level using MDE techniques. This approach can be used in general to analyze these properties for further processing, but may also be applied in the context of the objective and constraint specification in our model-level optimization approach described in the next chapter, cf. Figure 3.1.

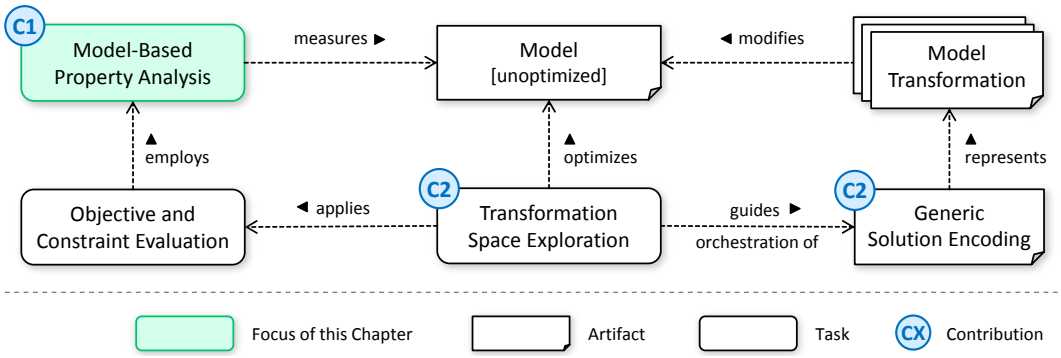


Figure 3.1: Model-Based Property Analysis Contribution

This chapter is structured as follows. First, we give a general overview of how properties can be categorized and describe general analysis techniques to retrieve and measure the value of properties of models in Section 3.1. In Section 3.2 we introduce a running example based on UML which is used to evaluate our property analysis approach presented in Section 3.3. The evaluation is performed in Section 3.4 and Section 3.5 concludes this chapter with an overview of related work.

3.1 Overview

In this section we give a general introduction into the categorization of properties and provide an overview of how these properties can be analyzed on model level.

3.1.1 Property Categorization

In the following, we describe three dimensions which are often used in literature to categorize properties: functional versus non-functional, external versus internal, and static versus dynamic. It should be noted that these dimensions are by no means complete and often properties cannot be strictly placed in one of the dimensions categories. Nevertheless, these dimensions help to broadly specify characteristics of properties which in turn can have an influence on which analysis technique may be applied to retrieve them. Specifically, we consider categories in the software engineering context as our approach is also evaluated on a software system, cf. Section 3.2.

Functional versus Non-Functional. The first dimension we consider is focused on what type of concern the property represents in the system. This is by far the most common dimension under which properties are categorized.

Functional properties relate to the provided functionality of a system, i.e., what concrete behavior is offered and the way the system can be used. For instance, a functional property may measure how much of the intended functionality is already covered by the implemented system or if the functionality behaves as expected. These properties may be evaluated using methods from requirements engineering and dedicated test cases to retrieve the coverage of correct functionality.

Non-functional properties (NFPs), sometimes also referred to as quality properties, are concerned with *how* the system functionality is provided. There is a plethora of non-functional properties relating to a systems performance, security, reliability or maintainability, to name just a few. Many different categorizations of NFPs are available in literature, often related to a specific domain. For instance, in software engineering, dedicated software quality models [MRW77, CM78, BBL76, Gra92, Dro95, ISO11] are defined to categorize software properties and use them as a base for tasks such as requirements elicitation, measurement and evaluation. A software quality model decomposes software quality into characteristics and sub-characteristics and defines relationships between them. In order to evaluate a specific characteristic, it is mapped to a measurable property called a *metric*. For a review of several software quality models, we refer the reader to the work by Miguel et al. [MMR14]. As an example, we depict the quality model from the ISO/IEC 25010:2011 standard [ISO11] in Figure 3.2. This model contains eight top-level characteristics which are then further divided into several sub-characteristics.

Nevertheless, breaking down quality into characteristics and metrics is only the first step. The second step is to actually measure these metrics. Quite often, the research done for

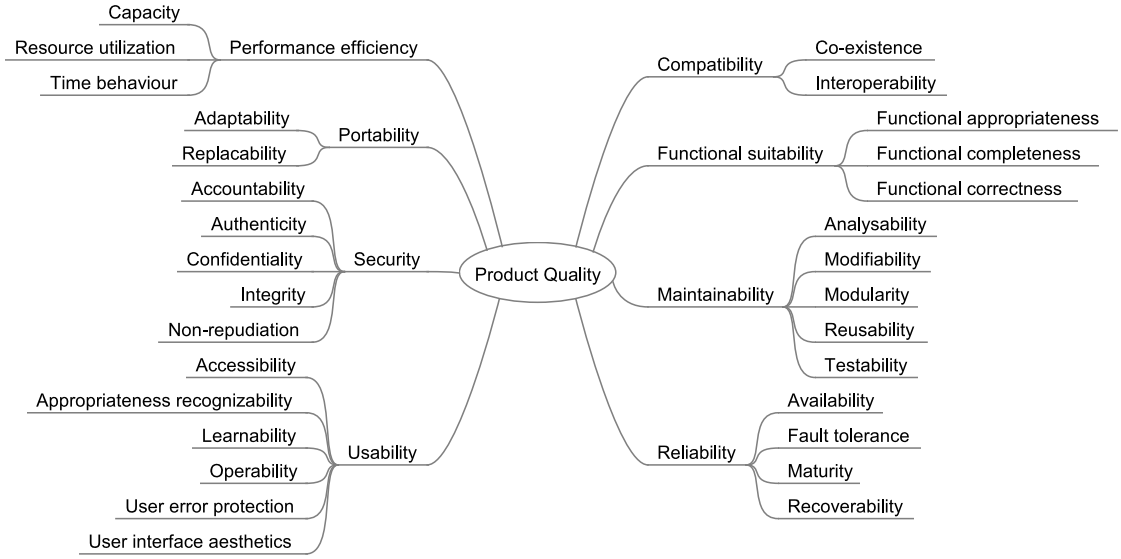


Figure 3.2: Product Quality Model Characteristics (illustrated from [ISO11])

a particular set of properties can be considered a research line on its own. For instance, there is the field of component-based software engineering [Szy02] that aims to tackle the complexity of a software system with a divide-and-conquer approach by modularizing the system into small, reusable components. How well the system is structured into components is then measured with dedicated modularization metrics such as coupling or cohesion. Consequently, having a well-structured system has not only a positive effect on the modularity of the system, but also on other maintainability-related characteristics such as modifiability or testability.

External versus Internal. Another dimension to classify properties is concerned with how they relate to the systems user [McC04, FP09]. External, sometimes also extrinsic [Zsc09], properties are properties which have a direct effect on the user experience as they view the system as a whole and are not concerned with the internal structure of the system or the contention of resources. Examples for external properties are correctness, usability, reliability or efficiency. In contrast, internal or intrinsic properties are concerned with the inner workings of a system. Examples include maintainability, testability, re-usability, and understandability. As noted by McConnell [McC04], internal and external properties may affect each other and therefore there is not always a clear-cut distinction between these two.

Static versus Dynamic. Another dimension, which is used for instance in program analysis, is related to how properties can be analyzed. Static properties are properties that can be analyzed based on the information that is available without having a running system. This includes properties relating to the system design and architecture.

Dynamic properties, on the other hand, are evaluated by executing or simulating a system and observing its execution. This comprises, among others, properties relating to the performance of a system, the fault tolerance and the systems availability.

3.1.2 Model Analysis Techniques

In the following, we briefly describe a few techniques with corresponding examples that illustrate how we can analyze properties on model level. We divide these techniques into static analysis techniques and dynamic analysis techniques. Typically, static analysis techniques are faster but less precise than dynamic techniques [AB05], but some information may only be available at runtime.

Static Analysis

Static property analysis techniques include the traversal of the underlying structure, dedicated model transformations and model checkers to reason about the potential states of a system.

Model Queries. The most basic way of accessing a model is through a dedicated model API provided by the respective metamodeling tool. Using this API, we can read the model into memory, traverse the graph structure and calculate properties of interest using the respective programming language. For instance, in the Eclipse Modeling Framework (EMF) we can automatically generate a Java model API from the information present in the metamodel.

Another way to traverse a model is to use model query languages. These languages provide a concise syntax and are often be integrated directly in the metamodeling tool. For instance, OCL [OMG14b] is a standardized, typed, side-effect free and declarative language to describe expressions on MOF-compliant models. While OCL can be used to express invariants on states, and pre- and post-conditions of operations, it can also state model queries. For instance, in EMF Refactor [Ecl14, AT13] OCL queries are used to calculate different metrics for Ecore and UML models. An example is shown in Listing 3.1 where the number of different classes being referenced from a given class is returned.

Listing 3.1 Metric Calculation Using an OCL Query

```
1: -- NDEROEC metric from EMF Refactor: number of different class references
2: -- query context is EClass, i.e., self is an instance of EClass
3: self.eReferences->select(ref : EReference | ref.eReferenceType <> self)
4:    ->asSet() -- only count unique class references
5:    ->size()
```

Model Transformations. As mentioned in Section 2.1.3, one possible transformation intent is to apply model transformations to extract properties of interest for certain parts of the model. One way to use transformation is to convert the information present in the model directly into the respective property value. Another way is used in EMF

Refactor, where they use the graph transformation engine to calculate numeric properties by counting how often a certain rule matches in the underlying model. Figure 3.3 depicts such a rule for calculating the number of parameters within other classes having the given class as a type. As all elements in this rule must be matched, we can simply count the number of matches. By giving the given class (`context`) a different name than the class containing the parameters, we specify that the containing class must be a different class than the given class.

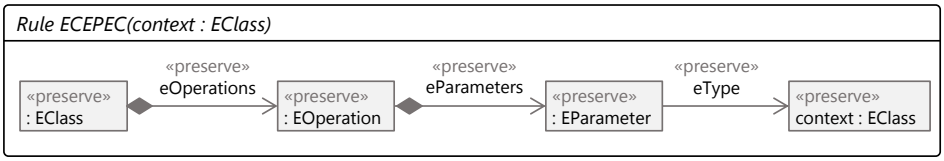


Figure 3.3: Metric Calculation Using a Model Transformation

Model Checking. Model checking is a technique to automatically verify correctness properties of finite-state systems using a state-transition graph to specify all potential states of a model and a temporal logic formula specification. The aim is to verify whether the specification is true in the given graph or not. Two well-known model checkers which are able to perform this verification are SPIN [Hol97, Hol03] and NuSMV [CCGR99, CCGR00, CCG+02]. These model checkers can then be used to verify properties of models trough translation to the respective input language of the model checker, as done in [KW06, KR06, JDJ+06] for UML models or by Meyers et al. [MDL+14] for annotated models of dedicated property specification and verification languages. Other approaches do not translate to dedicated model checkers, but aim to elevate the model checking techniques to MDE. For instance, Bill et al. [BGKS14] present a temporal extension of OCL based on computation tree logic (CTL) to specify constraints over the state space of a model. Their model checker uses the Henshin graph transformation engine to generate the state space and OCL to evaluate the constraint on these states. An example constraint in the extended OCL language for the Pacman example (cf. Section 2.1.5) is depicted in Listing 3.2. Here, we define that it must be true that in all games (`Always`), no matter how it is played (`Globally`), the game is over, i.e., no further state can be reached (`Always Next false`) if Pacman has reached the treasure (`self.pacman.on.treasure`).

Listing 3.2 Temporal Query Using OCL Extended with CTL

```
1: -- The keywords Always, Globally, and Next are part of the CTL extension
2: Always Globally (self.pacman.on.treasure) implies (Always Next false)
```

Model checking is usually faster than theorem proving, it provides counterexamples if the specification is not satisfied and allows partial specifications. However, besides the difficulty of writing temporal logic specifications, model checking suffers from the state explosion problem [Cla08].

Dynamic Analysis

Dynamic analysis methods have their root in the execution or simulation of models in order to test a specification, profile a system [TMB14] or analyze properties based on the resulting execution traces. In order to perform dynamic analysis on model level, we need three ingredients: (*i*) precisely defined execution semantics of the modeling language, (*ii*) a corresponding execution platform, and (*iii*) analysis capabilities based on the execution information provided by the platform to retrieve property values.

Unfortunately, many modeling languages lack such a precise semantics definition. For instance, UML [OMG15c], the most common language used to develop software systems, is described using English prose. This has led to many approaches that translate models into dedicated analysis formalisms where these semantics are provided, e.g., [WPP+05, MB04] translate UML models to queuing networks in order to analyze performance-related properties. While the execution of models has proven to be very successful in many application domains to simulate and validate non-functional properties functional behavior with tools like Maude [CDE+07] and MathWorks Simulink [Bor15], translational approaches introduce an additional layer of indirection, provide the results in a different language than the original input and often implement the missing semantics in the transformation.

This problem has also been recognized by the OMG which standardized the execution semantics of a subset of UML with Foundational UML (fUML) [OMG16] in 2011, cf. Section 3.2.1. As a result, new approaches have emerged that allow the analysis of non-functional properties [BLM13] and the testing and debugging of the modeled behavior [MLMK13] directly using fUML models. Also our approach to analyze the contention of resources falls under this category, cf. Section 3.3. Interestingly, using the standardized execution semantics of UML, Mayerhofer et al. [May13, MLW13, May14] even proposes to use fUML as a language to define the execution semantics of other MOF-based modeling languages.

3.2 Running Example

After giving a general overview on property categorization and model-level property analysis techniques in the next section, we describe an example in this section to introduce our approach in the next section. This example represents a simple online store called *PetStore* which is modeled using UML. This example is also used to evaluate the feasibility and applicability of our property analysis approach later on. Before introducing the *PetStore* system, we provide a short overview of UML to facilitate the understanding of the system and our approach.

3.2.1 Unified Modeling Language

UML. The Unified Modeling Language (UML) [OMG15c] is a standardized, MOF-compliant, general-purpose modeling language intended for developing software systems.

For this purpose, UML offers structure diagrams to model the system structure and behaviour diagrams to model the functionality and interactions of a software system. For instance, the UML class diagram is a structure diagram used to model the classes of the system, their interrelationships (including inheritance, aggregation, and association), and the operations and attributes of the classes. Other structure diagrams include the package diagram that shows the dependencies between the packages that make up a model, the component diagram that represents how components are wired together to form larger components and the final software system, and the deployment diagram that shows how the software system is deployed on physical device nodes. In order to express behavior UML offers, among others, activity diagrams to represent stepwise activities and actions within a workflow, sequence diagrams to indicate how objects communicate with each other through messages, and state machine diagrams to express the states of a system and to specify how the system can transition between these states.

UML Profiles. UML profiles are light-weight extensions to the existing UML language. Profiles are used to add additional information to existing UML model elements to extend UML with domain-specific concepts [Sel07]. A profile can define stereotypes, data types, primitive types, and enumerations. Stereotypes extend the model elements on which they are applied [AKH03] with so-called *tagged values*, i.e., new meta properties that can use the defined data types, primitive types, or enumerations. These meta properties are used to hold the domain-specific information and basically consist of key-value pairs. A list of all official UML profile specifications can be found on the OMG website[1]. Examples profiles include the UML Testing Profile (UTP) [OMG13b] used to define testing scenarios on model level and the Modeling and Analysis of Real Time and Embedded Systems Profile (MARTE) [OMG11b] that defines concepts for time and resource modeling and has a dedicated sub package for defining non-functional properties.

Foundational UML. As previously mentioned, Foundational UML (fUML) [OMG16] is an OMG standard that defines the execution semantics of a subset of UML. This subset mainly covers the structural and behavioral kernel for defining the structure of a system using *UML classes* and its behavior using *UML activities*. Specifically, the subset includes parts of the UML packages *Classes*, *Common Behaviors*, *Activities*, and *Actions* and therefore essentially allows the execution of UML activities. The fUML standard is accompanied by a Java-based reference implementation of a *virtual machine*, which allows computing outputs from executing fUML activities with specified input parameters. Furthermore, a Java-like textual syntax for fUML diagrams, called Alf [OMG13a], is also available. By complying with the fUML standard, activity diagrams can be executed directly without having to rely on any non-standardized simulators or translating the activity diagrams into different notations. However, fUML only covers a subset of UML and many high-level concepts known from programming languages are currently not supported such as exceptions, annotations, and reflection. Therefore, model translations from UML models to fUML-compliant models may still be necessary in some cases.

[1] OMG Specifications: `http://www.omg.org/spec/`

3.2.2 PetStore

The *PetStore* is a simple online store where customers can register, log in, browse a catalogue of pets, add them to their shopping cart, and place orders. It is used as a running example to introduce our property analysis approach in the next section and to evaluate the approach in Section 3.4.

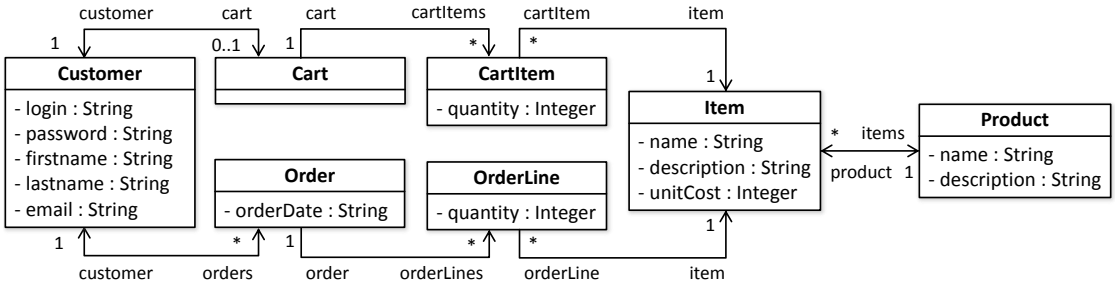


Figure 3.4: *PetStore* Entities

The *PetStore* system consists of the entity classes depicted in Figure 3.4 and explained in the following paragraphs. Please note that in the notation used in the figure, a minus (−) in front of an attribute or method indicates that this feature is marked private and a plus (+) indicates a public features. References are depicted as arrows with their respective cardinality shown at the end of these arrows.

Item The pets and pet supplies that the *PetStore* offers via its online store are represented in the system as items. Each item consists of a unique name and a detailed description providing the features of the item and the price for which the item can be bought.

Product To enhance the browsing experience for store visitors, all items are put into product categories. A product consists of a unique name and an overall description of the category.

Customer In order to purchase items, visitors need to create an account which turns them into customers. A customer has credentials consisting of the user login and the password as well as general information such as the first name, last name and email address that is used for correspondence.

Cart While browsing the store, customers are given a virtual shopping cart in which they can put items they intend to purchase.

CartItem Items in the cart are linked to one specific shopping cart which in turn is connected to a customer account. The quantity attribute of the cart item indicates how many instances of a particular item the customer intends to purchase.

Order If customers are finished with browsing the store and putting items into their cart, they can place an order. For transparency, we capture the exact date and time when the order is placed.

OrderLines One order can contain several lines whereas each line corresponds to one particular item and the respective quantity.

Based on these entity classes, the following service classes are used to provide the functionality for the *PetStore*. Figure 3.5 depicts these classes with their methods and the relations among them.

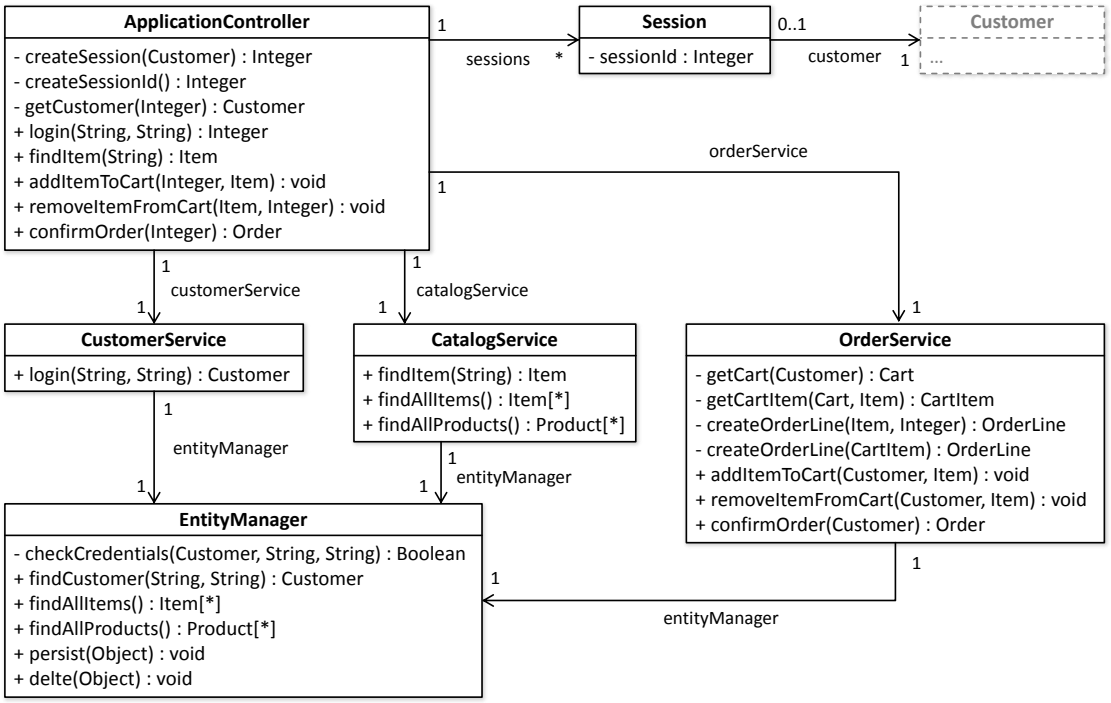


Figure 3.5: *PetStore* Services

Session As soon as a user logs into their customer account, we create a session that captures the interactions of the customer with the *PetStore*. As common in most online stores, a session is identified by a dedicated session id that is transferred with each HTTP request.

ApplicationController On the server side, all sessions are handled by the application controller. This controller is responsible for the creation of a session and acts as interface for all functionality provided to the user. Internally, the functionality is distributed among three separate services based on the respective entity classes.

CustomerService The customer service is responsible for handling all data concerning the customer account. Most notably, this includes the login functionality which yields returns the customer object representing the customer account if the correct credentials are provided.

CatalogService The catalog service handles all interaction relating to products and items, e.g., finding all items and all products or only finding specific items based on their name.

OrderService The order service takes care of the shopping process, i.e., the shopping cart and the placement of orders.

EntityManager All services have access to an entity manager to provide their functionality. The entity manager offers operations to persist, retrieve, and delete *PetStore* data from the data store, e.g., the database.

În our example, we define the behavior of the *PetStore* system with UML activity diagrams. An example is depicted in Figure 3.6 where we realize the `checkCredentials` operation from the `EntityManager` class. The operation consists of three input parameters and one output parameter and compares the provided credentials, i.e., login and password, with a given customer. If the credentials match, we return true and false otherwise.

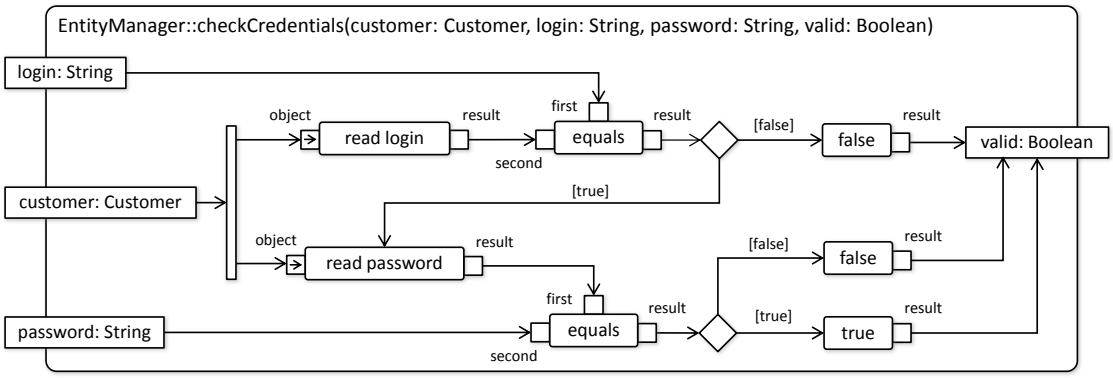


Figure 3.6: *PetStore* EntityManager CheckCredentials Behavior

3.3 Resource Contention Analysis Approach

This section introduces the first contribution of this thesis, i.e., a property analysis approach that analyzes performance-related properties of UML models based on fUML and queuing network theory while considering the contention of resources. This approach has also been published in [FBL+13] and is described in the following paragraphs.

3.3.1 Motivation

With the advent of model-driven software engineering, developers are empowered to raise the level of abstraction during the development using high-level models and to automatically generate executable code. This shift from code to models facilitates also the analysis of non-functional properties at early development stages [PAT12]. UML is currently the most adopted design modeling language whose extensibility, through UML profiles, lead to the emergence of several UML-based approaches for analyzing NFPs of the modeled software. However, due to the lack of formal execution semantics of UML and the lack of UML-based tools for NFP analysis, current approaches translate UML models into different kinds of analysis models, such as queuing networks (QN), for sake of performance analysis. Thus, a semantic gap between UML models and analysis models has to be bridged using often complex chains of model transformations before NFPs can be analyzed.

Although researchers have accomplished significant advances in transforming UML models in combination with applied UML profiles, such as MARTE [OMG11b], to dedicated analysis models, translational approaches suffer from some inherent drawbacks. Transformations have to generate analysis models that correctly reflect the heretofore informal semantics of UML models using concepts of the target analysis modeling language. Implementing such transformations is a complex task that requires deep knowledge not only of the semantics of UML and of the target analysis languages, but also of model transformation techniques, which hampers significantly the development and emergence of novel analysis tools. Even though transformations already exist, such transformation chains introduce inevitably an additional level of indirection, additional notations, and hence additional complexity, such as the consistent propagation of UML model changes to the analysis model and analysis results back on the UML model. This is a very relevant obstacle to the usability of analysis tools, because usually software developers are not trained in understanding formal languages applied for the analysis [BMIS04].

To address these drawbacks, France et al. [FR07] suggested integrating analysis algorithms directly with the source modeling language, such as UML. Following this suggestion, a framework for analyzing NFPs based on executing UML models directly has been proposed by Berardinelli et al. [BLM13]. Instead of translating UML models into different notations, they showed how the execution semantics of fUML can be exploited to obtain model-based execution traces [MLK12] from UML models and how these traces can be processed to compute NFPs, such as resource usage and execution time. However, their framework only supported the analysis of single independent execution traces, and could not consider the *contention of resources*. This aspect, however, is of uttermost importance when it comes to analyzing, for instance, the scalability of cloud-based applications on the IaaS layer or the thread contention in multicore systems.

In our approach, we address this limitation and extend the framework to study the influence of resource contention on NFPs, such as *response time, throughput, and utilization*, which require the consideration of multiple overlapping executions. We enable this

analysis within the fUML-based framework by obtaining *execution traces* from executing UML models that are annotated with the MARTE profile [OMG11b], attach timing information to these execution traces according to a *workload* specification, compute the *temporal relation* of these execution traces, and calculate performance indices that can so far be only obtained through translating UML models and performing the analysis based on other notations and formalisms, such as QNs.

As no transformation and no notation other than UML is involved, the presented framework is easily extensible with respect to the integration of additional analysis aspects. Thus, we further incorporated the analysis of *load balancing and scaling strategies* into our framework. Thereby, developers are equipped with methods for reasoning about optimal resource management configurations of the modeled components.

3.3.2 Trace-Based Analysis Framework

In order to realize our approach, we use the fUML analysis framework introduced by Berardinelli et al. [BLM13] as basis. Their analysis framework uses the open-source fUML implementation of the extended fUML virtual machine by Mayerhofer et al. [MLK12, May12], which enhances the standard virtual machine from the OMG with capabilities to not only produce outputs but also capture *execution traces* as additional output of a performed model execution. An execution trace provides the information necessary for analyzing the runtime behavior of the executed model. It captures information about the call hierarchy among executed activities, the chronological execution order of activities and contained activity nodes, the input provided to and the output produced by the activities and activity nodes, as well the token flow. These execution traces are then used for further analysis, an idea also outlined in the Software Performance Engineering approach [SW02] by using UML models annotated with UML SPT [OMG05] stereotypes and execution graphs to aggregate demands of computing, storage and communication resources. An overview of the framework is depicted in Figure 3.7.

The framework takes common UML models and applied profiles as input, seamlessly adapts those models to fUML for executing them, and transparently maps the resulting execution traces back to the level of UML, where the information on profile applications is again accessible. Thus, the framework enables the development of analysis components that leverage the well-defined semantics of fUML for capturing the runtime behavior of UML models in combination with the additional information from UML profiles.

UML Profile Application. As described in the previous section UML profiles are lightweight extensions to existing UML languages using stereotypes. These stereotypes can be used to provide additional information needed for the evaluation of the non-functional properties, e.g., time needed to add two integer values. From this low level information, it is possible to calculate information on a higher abstraction level, e.g. time one method takes to execute. Often the low level data depends on the specific platform the application is executed on, e.g. the power of the CPU or the read/write speed of the hard drive. Since there is no existing profile that covers all non-functional

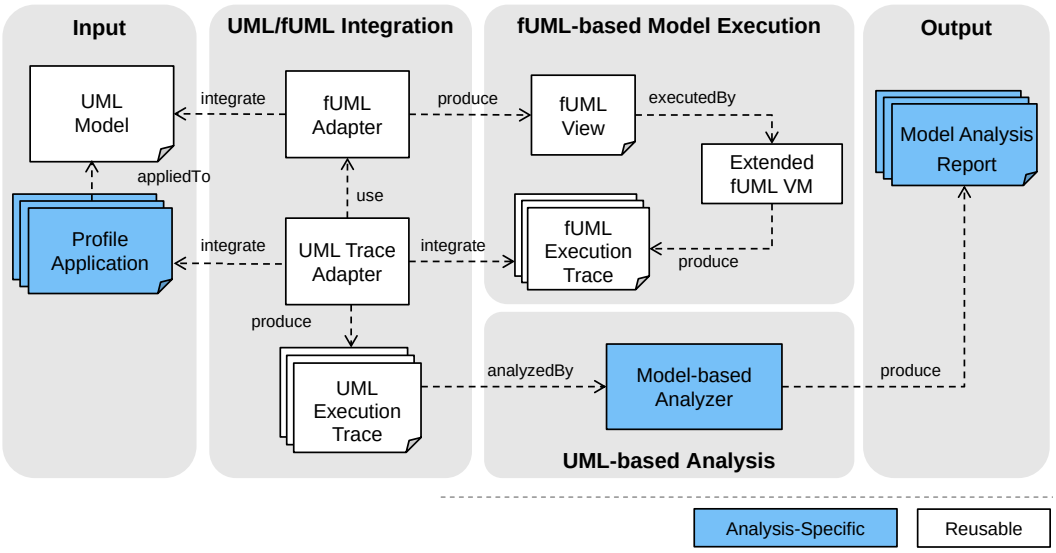


Figure 3.7: fUML-Based Analysis Framework For Non-functional Properties [BLM13]

properties out of the box, multiple profiles can be used, e.g., MARTE for performance properties or the MARTE-extension for dependability properties Dependability Analysis Modeling (DAM) [BMP09]. The annotated UML model can then serve as input to the fUML Adapter.

UML/fUML Integration: fUML Adapter. The framework uses *fUML Adapter* to convert UML models into fUML models and produce the in-memory representation of the model necessary to execute it in the virtual machine. In its current implementation, the Adapter truncates all UML model elements that are not conform to the supported subset of fUML, maps the conforming elements with their fUML counterpart and then generates the *fUML View*. This includes also any information annotated with the UML profile applications, which need to be introduced again by the UML Trace Adapter. While the additional meta-information from the profile applications can be introduced again, non-compliant UML model elements are lost and cannot be included in the model-based analysis. This is considered in our approach and only compliant elements should be used.

fUML-based Model Execution. Using the fUML View created by the fUML Adapter, i.e., instantiated Java classes, the *extended fUML VM* can execute the modeled behavior. As mentioned previously, the fUML standard is accompanied by a Java-based reference implementation of a *virtual machine*, which allows computing outputs from executing fUML activities with specified input parameters. However, while this enables executing fUML-compliant models and validating their execution output, a thorough analysis of a performed model execution is not possible. This prevents the model-based analysis of functional and non-functional properties of the modeled system. From this VM we are able

to retrieve a detailed *fUML Execution Trace*. The execution trace includes the runtime information of the model execution, i.e., the call hierarchy of executed activities, the chronological execution order of the activity nodes contained in the executed activities, the input provided to and the output produced by the executed activity nodes and activities, as well as the token flow between activity nodes and activities. This information can be used to reason about the model execution and analyze the details.

UML/fUML Integration: UML Trace Adapter. As mentioned previously, the extended fUML virtual machine only handles fUML-compliant model elements and therefore the fUML Adapter truncates the data provided by the UML profile applications. It is the task of the *UML Trace Adapter* to integrate this missing data back into the fUML execution trace and produce the enriched UML execution trace. To allow the re-introduction of the profile information, the UML Trace Adapter uses the mapping information generated by the fUML Adapter, i.e., which fUML element corresponds to which UML model element.

Model-Based Analyzer. As a final step, the enriched UML execution trace can then be used to analyze the model runtime information together with the non-functional properties annotated in the model, e.g., the runtime of a method based on how often the method was called and how much time the contained activities need. This analysis is done by a dedicated *Analyzer* which is also responsible to produce the desired output, e.g., property values, graphs, etc.

3.3.3 Resource Contention Analyzer

Based on their framework, Berardinelli et al. [BLM13] showed how performance analysis methodologies that are based on execution graphs [SW02] can be conducted directly on UML models and execution traces to aggregate demands of computing, storage, and communication resources. However, in order to carry out a performance analysis that considers the *contention of resources*, we have to deal not only with single independent executions but multiple overlapping executions. This, however, is not possible yet as neither plain fUML nor our existing analysis framework allows running competing executions of models concurrently within a shared runtime environment. Only sequential executions are supported so far, resulting in execution traces that are independent of each other. However, when analyzing software systems, performance indices concerning the contention of resources, such as *response time, utilization, and throughput*, are of utmost importance.

In this section, we show how we address this limitation to enable the analysis of resource contention based solely on UML models, profile application, and model execution, without the need to translate the involved models into different notations and formalisms, as it is done in existing approaches. Please note, that in our approach we consider software components as shared resources, whereas this notion could be extended also to platform

resources. We provide in the following a brief overview on the basic idea behind our approach and discuss the input of the analysis process and the individual components.

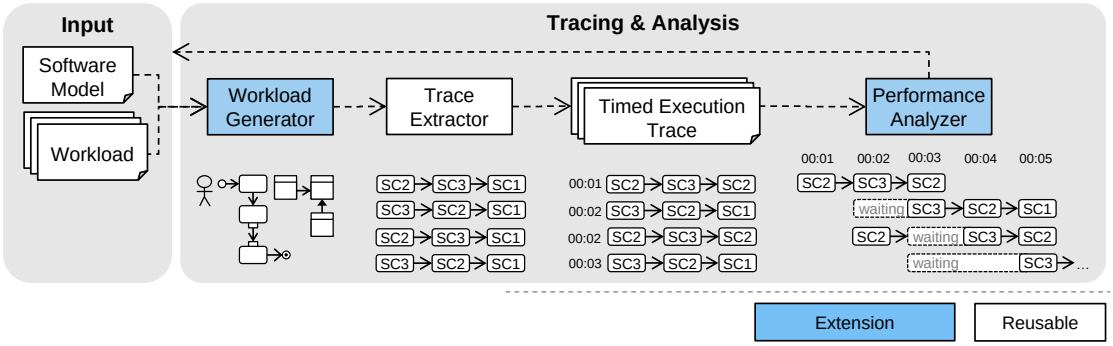


Figure 3.8: Model-Based Performance Analysis Framework

As proposed by Di Marco [DM05], we adopt the idea of considering software components as *service centers* [LZGS84]. A service center represents a software component that provides some sort of functionality as services to other classes. For instance, in the *PetStore* the *ApplicationController*, *CustomerService*, *CatalogService*, *OrderService*, and *EntityManager* may be considered service centers. A *job* is a request for the *services* of different service centers and has an arrival time specifying the point in time at which it enters the system. As long as a request is processed by a service center, further requests to this service center are stored in a queue until the service center is available. After the request has been processed, the next request from the queue is chosen following a first-come, first-serve (FCFS) principle. Based on these concepts, mature algorithms are available to compute performance indices under resource contention.

In order to apply these concepts to UML models directly, without translating them into dedicated performance models, we propose to: (*i*) trigger executions of UML models according to specific *workloads* for obtaining execution traces that reflect the runtime behavior of jobs (i.e., which services are requested in which order), (*ii*) annotate the arrival time to each of the resulting execution traces, and (*iii*) compute, based on known service times of consumed services, the temporal relation of concurrently running jobs (cf. Figure 3.8). Based on the temporal relation of executed jobs, we can step-wise derive their temporal overlaps and compute waiting times in each queue and, in further consequence, the overall response time, throughput, and utilization indices. In addition, we introduce dedicated types of service centers that support *balancing and scaling strategies* to allow users of our framework to reason about optimal resource management configurations. In particular, a single service center may distribute incoming jobs to multiple instances of this service center according to certain strategies, as well as dynamically allocate and deallocate instances (*horizontal scaling*).

Input

As input of our proposed approach, we use a UML model annotated with MARTE stereotypes representing the system. A summary of the stereotypes used for the software model (SW), hardware platform (HW), workload model (WL), and property analysis results (PA) can be found in Table 3.1. The UML model contains the specification of the software structure and behavior, whereas MARTE is used to specify the system workload(s) as well as the performance characteristics of its structural and behavioral elements.

Table 3.1: MARTE Stereotypes in the Performance Evaluation Framework

View	Stereotype Name	Applied To	Used Tagged Values
HW	HwProcessor	Class	mips
SW	RtScalableUnit	Class	srPoolSize, queueSchedPolicy, scalingStrategy, timeToScale, scalingRange, scaleInThreshold, scaleOutThreshold, balancingStrategy
SW	GaStep	Activity	execTime
WL	GaScenario	Activity Diagram	cause, root
WL	GaWorkloadEvent	Activity	effect, pattern
PA	GaAnalysisContext	*	contextParam

The *software specification* consists of one or more class diagrams defining the structure and activity diagrams representing its behavior, as shown for the *PetStore* in Section 3.2. Classes that should act as software service centers during the model execution (*RtScalableUnit*) have to be extended with information regarding: (*i*) the initial number of instances (*srPoolSize*), (*ii*) the scheduling policy for the incoming operation calls (*queueSchedPolicy*), (*iii*) the balancing strategy for selecting the instance that receives the next request (*balancingStrategy*), and, optionally, (*iv*) the rules for horizontal scaling (*scalingStrategy, timeToScale, scalingRange, scaleInThreshold, scaleOutThreshold*). Currently, no stereotypes in MARTE can represent both balancing strategies and scaling rules. Thus, we extended the existing stereotype *RtUnit* in this initial version of the framework to provide a set of pre-defined rules from which the model engineer can choose, such as round robin or random balancing, and scaling based on the queue length.

In addition, the UML activities representing the software behavior (*GaStep*) have to be annotated with their respective execution times (*execTime*). These values may be either computed by estimating the complexity of behavioral units (e.g., number of executed instructions) in combination to the underlying platform characteristics (e.g., millions

of instructions per second of *HwProcessor*), as proposed in [BLM13] and shown in Section 3.4.

Alongside the structure and behavior of the software, the model engineer has to specify the *workloads* in terms of UML activities that represent the expected interactions with the software. Such interactions start with a workload event (*GaWorkloadEvent*), e.g., a user interaction with the system, and a behavior scenario (*effect*) that is triggered by that event (*cause*). A behavior scenario (*GaScenario*) is a sequence of execution steps, i.e., calls of activities in the software models, that require the operations associated to service centers, i.e., the *RtScalableUnit* classes. To specify how often a scenario is triggered, the model engineer provides an arrival *pattern* for different types of workloads, such as periodic, open, or closed workloads.

After the evaluation has been performed, the results are attached to the respective model elements, e.g., models, classes, or attributes, through the *GaAnalysisContext* stereotype. All results pertaining to a particular model element are provided as key-value pair using the context parameters (*contextParam*) of that stereotype. This allows the user to view the results directly in the UML modeling editor and avoids a translation between different formalisms.

Workload Generator

Once the UML model is provided, the analysis can be started. In a first step, a *workload generator* component reads the scenarios defining the software workload and automatically runs each of them once by executing the associated activities on top of the fUML virtual machine. From these executions, we obtain one execution trace for each scenario that captures the order in which services are requested as well as the execution time for each of the requests. In a next step, the traces are annotated with their arrival times as obtained from the inter-arrival times randomly generated from a probability distribution (e.g., exponential) according to the specified arrival pattern. This step results in a set of *timed execution traces*.

Performance Analyzer

The Performance Analyzer takes the timed execution traces with their execution time as input and performs common operational analysis as defined in [LZGS84]. Special consideration is given to service centers having multiple instances, because they require balancing strategies to determine which instance will get the next request from the queue. We support balancing and scaling strategies, such as round robin, random balancing, and scaling based on queue length. However, additional strategies can be easily added by extending the interpretation of the respective MARTE stereotypes. The obtained performance values include, among others, the waiting time and service time for the scenarios and utilization and throughput for the service centers, i.e., instances of the corresponding UML classes.

Using our performance analyzer, it is possible to calculate the performance values for the overall simulation time, for a specific point in time or for a given time frame. As a result, we can generate detailed graphs that show the evolution of individual performance values in order to provide additional insight into the systems behavior.

In the next section, we evaluate our resource contention analysis approach by applying it on the *PetStore* system and validating the results of our performance analyzer.

3.4 Evaluation

Using our approach, we can compute the utilization, throughput, and response time of all jobs for the overall workload, as well as the minimum, average, and maximum waiting time and service time of the jobs for each scenario. Additionally, we calculate the idle time, busy time, utilization, throughput, as well as the minimum, average, and maximum queue length for each service center. The computed results are annotated in the UML model.

In order to evaluate our approach, we are interested in the following two research questions:

RQ1 *Applicability:* Is our approach applicable to software systems modeled in UML?

RQ2 *Result Analysis:* Does our approach produce results comparable to results of established property analyzers?

To answer RQ1, we apply our approach on the *PetStore* case study, already partially introduced in Section 3.2. We demonstrate what additional information is needed and what output is produced.

In order to validate the analysis results obtained by our approach for RQ2, we compare its results for the *PetStore* case study with the results retrieved from an existing QN solver. Among them, our first choice is JSIMgraph (JSIM), a QN models simulator part of the Java Modelling Tools (JMT) suite[2] [BCS09]. JMT is a standalone Java application that allows performance evaluation using queuing networks with Mean Value Analysis and similar algorithms for exact and approximate solutions to retrieve the residence time, utilization, and throughput for each service center and overall. The necessary model can be provided via a textual wizard or via a graphical interface. Extensions have been made to also allow non-product-form models containing blocking, parallelism, priority classes or class switching. Besides bottleneck identification and a graphical simulation of Markov chains, JMT also allows workload analysis, e.g., checking for univariate or multivariate data or clustering.

[2] `http://jmt.sourceforge.net/`

3.4.1 RQ1: Application on the PetStore

The software architecture and the hardware platform of the *PetStore* have been modeled in UML[3] as described in Section 3.2. Additionally, to apply our analysis approach we need to provide additional information through a subset of MARTE stereotypes and properties. Specifically, we need to provide a scenario which annotates the expected runtime for each operation, declare what parts of the software system should be used as service centers and define a hardware specification as context of the execution.

Scenario In order to analyze the resource contention, the functionality of the *PetStore* needs to be modeled using UML activity diagrams while the structure is defined using class diagrams. Based on this functionality we can define a scenario that is executed for our dynamic analysis. In this case study, we want to analyze a typical online shopping workload consisting of the *Single Buy Scenario*, which represents a user that logs into the *PetStore*, searches for a specific item, adds this item to the shopping cart and confirms the order. We assume this to occur on average every two seconds, exponentially distributed, which is annotated as pattern in the triggering event of the scenario. Figure 3.9 depicts the *Single Buy Scenario* including the involved classes, the called operations and the applied stereotypes as sequence diagram. Each lifeline corresponds to a service center instance while each asynchronous message corresponds to the invocation of an operation on the receiving lifeline. Note that in our model the service center annotations are applied to a class level and not to an instance level, so all instances of the same class share the same performance characteristics. The scenario as well as its setup, e.g., the available pets or existing user accounts, are modeled using fUML activity diagrams.

Hardware Specification Besides the software architecture and the behavior, we need information about the underlying hardware in order to estimate the times needed to execute the respective operations provided by the service classes. In this case study, we therefore assume a reference hardware platform that is capable of executing 200,000 millions of instructions per second (MIPS). Since all services use the same hardware, the contention of resources is limited to the software components. The hardware model is depicted in Figure 3.10 and shows the allocation of the *PetStore* software components on the reference hardware platform using the *allocate* relationship.

Estimating Execution Times Based on the hardware and the instructions it is capable to execute, we can obtain an early estimation of the execution time of the operations provided by the services using an *overhead matrix* as defined by Smith and Williams [SW02]. An overhead matrix contains the software-to-hardware unit conversion factors that are used to combine an estimated number of high-level instructions executed for each operation call with the capability of the CPU. For our conversion, we use the number of instructions on three distinct levels of abstraction: assembly-level instructions as the lowest level of instructions closest to the CPU (L1), bytecode-level instructions

[3]The *PetStore* UML model can be downloaded at `http://www.modelexecution.org/`.

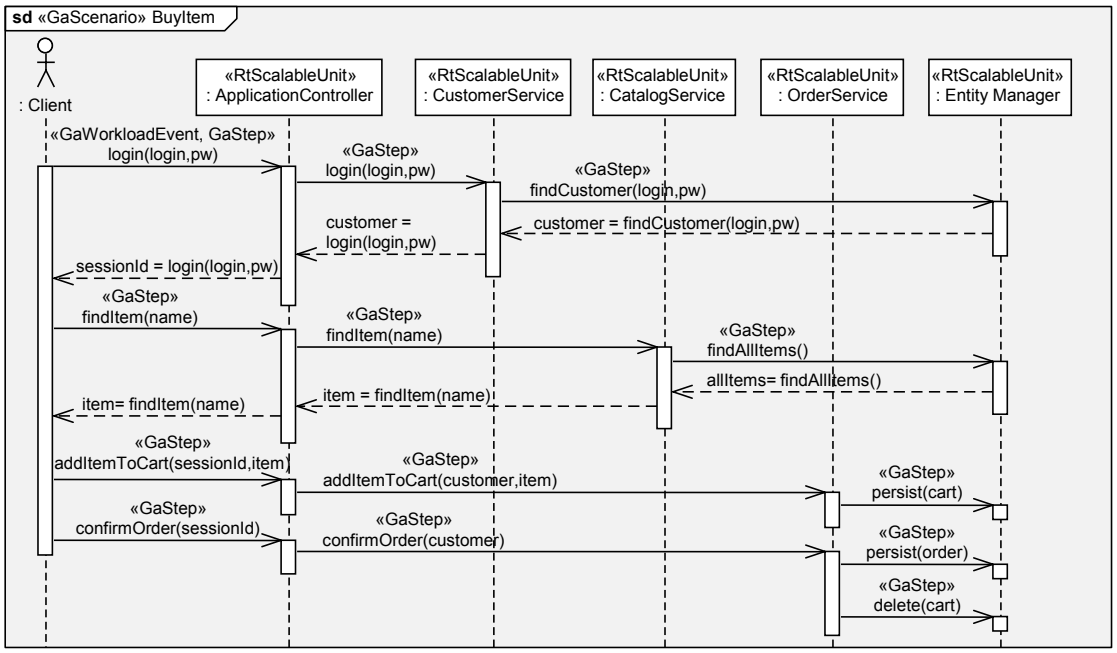


Figure 3.9: *PetStore Single Buy Scenario*

as executed in the Java virtual machine (L2), and finally high-level Java instructions as written by the developer (L3). To perform our conversion, we first measure the complexity of the operations provided by the *PetStore* software services as a number of high-level instructions executed for their invocation. Then, we set a conversion factor from assembly-level to bytecode-level to $\times 20$ and a conversion factor from bytecode-level to high-level to $\times 25$, as depicted in Table 3.2. The accuracy of this kind of estimation depends on the accuracy of such factors.

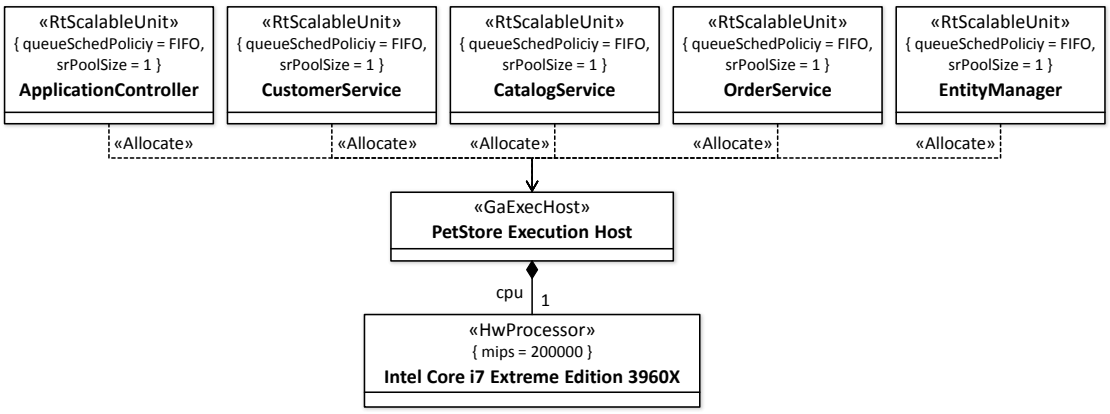


Figure 3.10: *PetStore* Hardware

	MIPS L1	MIPS L2	MIPS L3
MIPS L1	1.0000	0.0500	0.0020
MIPS L2	20.0000	1.0000	0.0400
MIPS L3	500.0000	25.0000	1.0000

Table 3.2: Overhead Matrix for the *PetStore*

	Number of executed instructions			
Operation	**MIPS L3**	**MIPS L2**	**MIPS L1**	**Time**
AppCon::login	10.00	250.00	5000.00	25.00
AppCon::findItem	5.00	125.00	2500.00	12.50
AppCon::addItemToCart	50.00	1250.00	25000.00	125.00
AppCon::confirmOrder	50.00	1250.00	25000.00	125.00
CusSer::login	100.00	2500.00	50000.00	250.00
CatSer::findItem	10.00	250.00	5000.00	25.00
OrdSer::addItemToCart	10.00	250.00	5000.00	25.00
OrdSer::confirmOrder	500.00	12500.00	250000.00	1250.00
EntMan::findCustomer	2000.00	50000.00	1000000.00	5000.00
EntMan::findAllItems	2000.00	50000.00	1000000.00	5000.00
EntMan::persist	1000.00	25000.00	500000.00	2500.00

Table 3.3: Execution Times for Operations in the *PetStore Single Buy Scenario*

Based on the capabilities of the CPU and our overhead matrix, we can estimate the execution times for our *PetStore Single Buy Scenario*. The concrete numbers and execution times are depicted in Table 3.3.

Finally, the operations are annotated with their respective execution times using the *GaStep* stereotypes. An example is shown in Figure 3.11 for the `login` operation of the `CustomerService` class, where we have annotated an estimated execution time of 250ms.

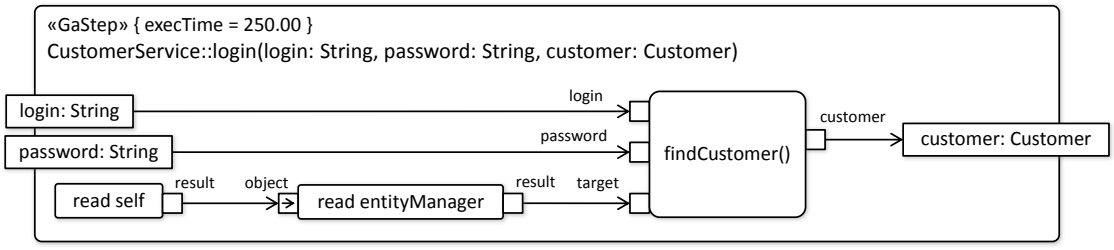


Figure 3.11: *PetStore* CustomerService Login Behavior

Output. Using the defined UML model as input, we can reason about different configurations on software service level and explore the effect of different balancing and scaling strategies. For simplicity, we consider four configurations in our example and focus on the utilization of service centers within the system, as well as the overall execution time. The utilization represents the busy time of a system or component in relation to its overall execution time. Furthermore, we focus on the provided reasoning based on the auto-generated graphs while for each configuration we also obtain an output model which is equivalent to the input model with the analysis results annotated as stereotype applications.

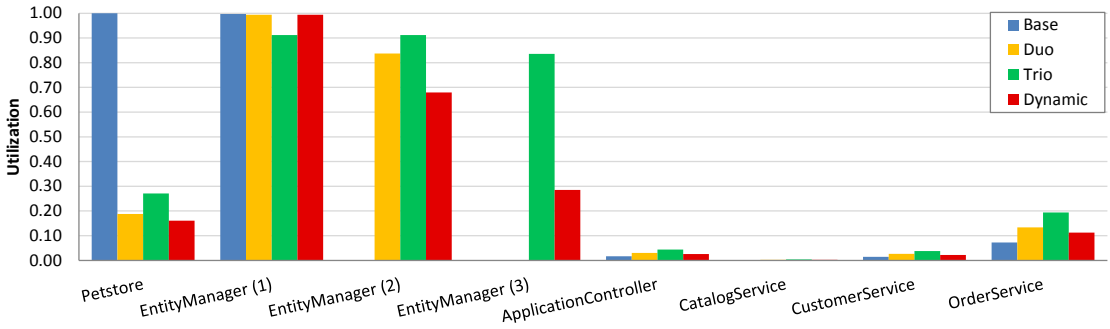


Figure 3.12: Utilization of the *PetStore Single Buy Scenario*

For all four configurations we apply the same workload within the same time frame of 6 seconds. The result for each single configuration is depicted in Figure 3.12. As baseline, the first configuration (*Base*) only considers one instance per service center and uses neither balancing nor scaling, resulting in an average execution time of about 87 seconds. From these results we can identify the *EntityManager* as the bottleneck of the application: the *EntityManager* has a very high utilization and blocks an optimal utilization of the other components. This is not surprising considering that the *EntityManager* is needed for almost every operation.

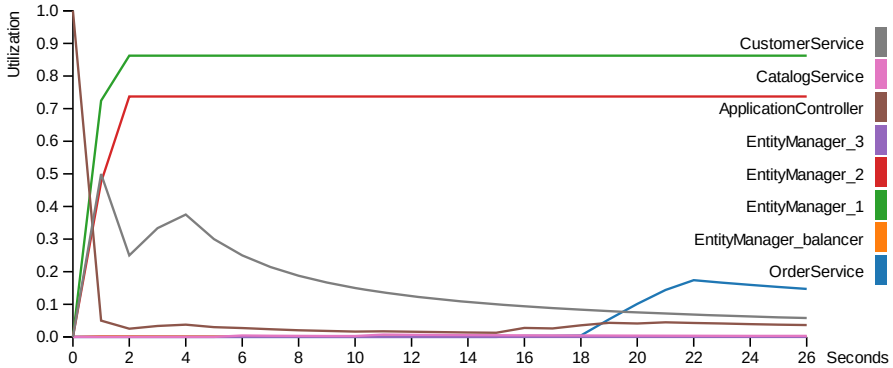


Figure 3.13: Evolution of the Utilization Property Over Time

Trying to improve this result, we introduce one (*Duo*) and two (*Trio*) additional *Entity-*

Manager instances and balance the requests between these instances using a round robin strategy. Figure 3.12 depicts the utilization of these additional instances, identified by the number in the parenthesis. A more detailed investigation of the results is possible by considering how the utilization of each instance behaves over time. Figure 3.13 depicts the evolution of the utilization for each instance for the *Dynamic* configuration. We can see that, in our example, multiple instances of the *EntityManager* center can reduce the execution time to almost a half or third and, hence, increase the utilization of the other service centers. However, more instances usually also imply more costs, thus making the number of instances a tradeoff between cost and performance.

In the fourth configuration (*Dynamic*), we vary the number of *EntityManager* instances dynamically instead of choosing a fixed number. We introduce the following horizontal scaling strategy: whenever the average queue length of the *EntityManager* is larger than 1.2, a new *EntityManager* instance should be allocated, and whenever the average queue length is lower than 0.6, an instance should be removed. For the sake of experimentation, the time needed for adding and removing instances is set to 100ms and we allow the number of instances to range from one to three, starting with one instance. The results for this configuration show that two additional instances are created during the run time, but neither of them can reach a high utilization, indicating that they might have been allocated too late. In comparison with the previous two configurations, we can further see that horizontal scaling yields no real benefit in our example. Possible reasons for this could be that the specified workload has little time between two jobs and that the average queue length does not reflect the changes made through scaling fast enough, resulting in a quick allocation and deallocation of many instances in a short period of time.

In conclusion, this case study illustrates that we can use our approach to analyze NFPs considering the resource contention solely based on UML models and execution traces. Moreover, it shows that our approach is extensible as it allows considering further concerns, such as the analysis of optimal configuration of balancing and scaling strategies.

3.4.2 RQ2: Comparison with JMT

In order to answer RQ2 and validate the results of our analysis approach, we compare them to the results retrieved by JSIMgraph (JSIM), an established QN models simulator. Our aim in this evaluation is not to re-implement the complete functionality provided by JSIM, but use it as a reference implementation from which we select a very small set of basic functionalities that are indispensable to calculate a performance metrics, such as the utilization of each service centers. For our comparison, we use a simple scenario (Scenario 1) and the *PetStore Single Buy Scenario* (Scenario 2).

The simple scenario is depicted in Figure 3.14 as queueing network modeled in JSIM. The QN is composed of a `source` node, a `sink` node and four service centers (`SCs`). Within the QN, we define two different types of customers (*TypeA* and *TypeB*) performing different requests on the queueing network. The interarrival times among jobs, expressed

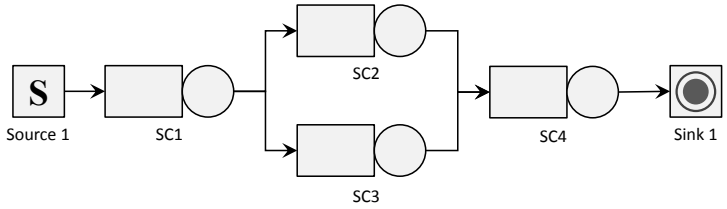


Figure 3.14: Scenario 1 QN Specified in JSIM

in milliseconds, are generated by two distinct exponential probability distributions $exp(\lambda)$. Each job from both customer classes generates three service requests before leaving the QN from the same *sink* node. Specifically, jobs of *TypeA* request the service centers in the following order `SC1`, `SC2`, and `SC4`, whereas customers of TypeB request the service centers as `SC1`, `SC3`, and `SC4`. At each service center, the service time (in milliseconds) required to complete the processing of a single job, is obtained by two probability distribution functions: $Deterministic(k)$ or $Exponential(\lambda)$. The former returns always k, i.e., the service time is deterministic and always equal to k milliseconds, while the latter generates a positive random natural number $0 < \lambda < 1$ according to an exponential distribution. Next, we create the same model (i.e., the same SCs, the same connections and parameters) programmatically using the API provided by our approach.

The second scenario, i.e., the *PetStore Single Buy Scenario*, has been discussed in the previous section. It is also modeled using four service centers.

Finally, in order to evaluate our approach, we execute the same scenarios with the same configuration with our approach modeled in UML and the queueing networks modeled in JSIM. This execution is repeated several times in order to raise the confidence of our evaluation and mitigate the risk of false results due to the randomness given through the exponential distribution in the service times. The results for our two scenarios for the average utilization property are depicted in Table 3.4.

Table 3.4: Average Utilization per ServiceCenter in JSIM and our Approach

	Scenario 1		**Scenario 2**	
	JSIM	**Our Approach**	**JSIM**	**Our Approach**
Service Center 1	1.000	1.000	1.000	1.000
Service Center 2	0.500	0.510	0.503	0.445
Service Center 3	0.505	0.510	0.248	0.261
Service Center 4	1.000	0.994	1.000	0.994

As we can see, the results are very close to each other and there is only a slight variation between the values. These variations are most likely the result of the different number of jobs created during the sampling process. Therefore, we conclude that our approach produces values very close to an established queueing network solver and as a result its values can be considered valid.

Nevertheless, there are to the validity of our evaluation. First, we only performed the analysis for one case study and second, we only compared the results to one queueing network solver. However, regarding the applicability of our approach, we do not expect any difference for other case studies as the model formalisms and our approach components stay the same. In order to mitigate the threat regarding the queuing network solver, we selected a well-known solver in order to improve the trust into its results.

3.5 Related Work

Due to the previous lack of semantics in UML, many UML model-based analysis approaches have implemented dedicated model transformations to specific performance models that can be used as input for existing analysis tools. In this section, we only briefly outline a few of these approaches. For more details, we refer to the respective approach or surveys such as the one performed by Balsamo et al. [BMIS04].

Performance from Unified Model Analysis [WPP+05] (PUMA) is an approach that uses transformation from UML software models into dedicated performance models. They use a transformation from UML activity, deployment and interaction diagrams annotated with the SPT profile into the common Core Scenario Model (CSM) format to avoid the N-by-M problem of transformations. The common format can then be translated into Layered Queueing Networks, Petri Nets, and Queueing Networks that can be fed into existing performance analysis tools. UML-Ψ [MB04] is a performance evaluation tool using process-oriented simulation to calculate the resource utilization, throughput and mean execution time of actions and use cases. UML Use Case, Activity, and Deployment diagrams are enriched with SPT profile annotations and combined with a configuration file translated into a C++ simulation model. All results are afterwards annotated on the respective UML elements using the Tag Value Language (TVL).

This eventually led to the introduction of common performance model interchange formats, such as PMIF [SLP10a] or CSM [PW07], to reduce the effort for transforming UML models to performance models and for integrating new methodologies with existing tools. However, due to the fact that many analysis tools existed before the introduction of the interchange formats, there is still limited support for these intermediate formats in analysis tools [SLP10b].

Other methodologies overcome this problem by introducing their own proprietary modeling notation and their own integrated analysis tools. For instance, Rivera et al. [RDV09] added time-related attributes directly to model transformation rules of their e-Motions tool based on Maude [CDE+07], a framework capable of simulating models using rewriting logic. Based on these attributes and the simulation engine, special observer objects can then be integrated to monitor the system and analyze non-functional properties such as throughput or idle-time [TRV10]. Another tool is Palladio [BKR09, BKR07], a software architecture simulation tool based on the Eclipse Modeling Framework. It allows the simulation and analysis of component-based software architectures that can be described with the so called Palladio Component Model (PCM). The model combines the component specification, the assembly model, the allocation model and the usage model. The PCM is an Ecore-based domain-specific language that bears a lot of resemblance with many UML2 models, e.g., specifying the behavior and usage in an activity diagram-like notation. Random variables are used to express resource demands, number of loop iterations and in QoS-related specifications. The PCM is also used by other simulators, e.g., SLAstic.SIM [vMvHH11] which changes component deployment and server allocation at runtime to control the QoS of distributed systems. The main

difference of our approach to Palladio besides the functionality is the usage of standards, which promotes interoperability and tool-independence.

Recently, also approaches that directly execute models for analysis without translating them into dedicated analysis models or code have emerged. Especially, since the introduction of the fUML standard [OMG16], more and more tools utilize the standard-conform execution of UML models. For instance, Moliz [MLMK13] is a testing and debugging framework for fUML models based on the extended virtual machine by Mayerhofer et al. [MLK12, May12]. Another example is Moka [Ecl16c], an extension to the Papyrus UML editor that is capable of simulating UML activity diagrams using the standard execution engine or a user-provided execution engine, and provides basic debugging support based on the graphical notation used in Papyrus.

CHAPTER 4

Marrying Optimization and Model Transformations

4.1 Overview

In this chapter, we introduce a model-level optimization approach that combines search-based optimization (SBO) methods with model transformations in order to facilitate the tackling of highly complex problems using MDE techniques. The related contributions of this approach are highlighted in Figure 4.1 and explained in the following paragraphs.

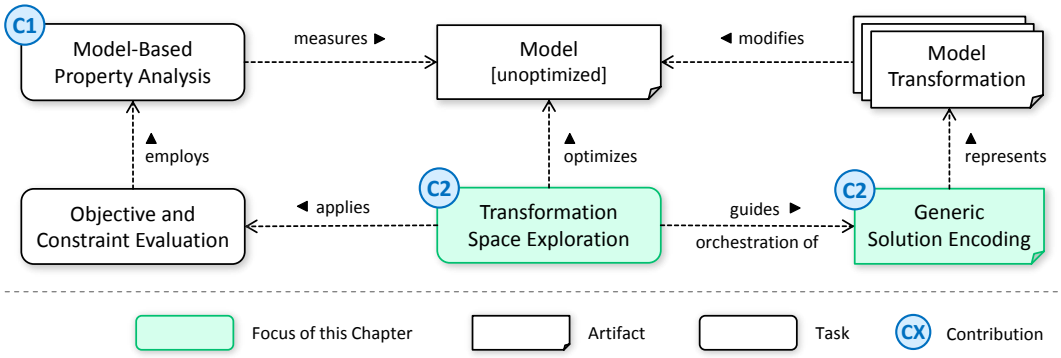


Figure 4.1: Marrying Optimization and Model Transformations Contribution

Model transformations are an important cornerstone of model-driven engineering, a discipline which facilitates the abstraction of relevant information of a problem as models. The success of the solutions heavily depends on the optimization of these models through model transformations. Currently, the application of transformations is realized either by following the apply-as-long-as-possible strategy or explicit rule orchestrations

have to be provided, which suffers from several drawbacks as outlined in Section 1.2. Summarized, these drawbacks include the implicitly hidden effect a transformation has on the characteristics of the model, the required knowledge to understand the relationships among transformation rules, i.e., whether they are conflicting or enabling, the large or even infinite number of rule combinations, and the consideration of multiple, potentially conflicting objectives.

To overcome these drawbacks, we present a novel approach which builds on the non-intrusive integration of search-based optimization and model transformations to solve complex problems on model level. In particular, we formulate the transformation orchestration problem as a search problem by providing a generic solution encoding based on transformations, which allows for search-based exploration of the transformation space and explicating the transformation objectives. The idea to tackle this problem using search-based techniques is also reflected in the recent approaches by Abdeen et al. [AVS+14] who integrate multi-objective optimization techniques to drive a rule-based design exploration and Denil et al. [DJVV14] who integrate single-state search-based optimization techniques such as Hill Climbing and Simulated Annealing directly into a model transformation approach. Based on this trend and the ideas that we have initially outlined in [KLW13] on combining metaheuristic optimization and MDE, similar to Search-Based Software Engineering (SBSE) [Har07], we introduce a problem- and algorithm-agnostic approach called *MOMoT* (Marrying Optimization and Model Transformations), also published in [FTW15, FTW16b]. Our approach loosely couples the MDE and SBO worlds to allow model engineers to benefit from search-based techniques while staying in the model-engineering technical space [KAB02], i.e., the input, the search configuration and the computed solutions are provided at model level. In particular, we are focusing on how different metaheuristic methods can be used to solve an optimization problem, on how we can support the model engineer in configuring these methods, and on the separation of the objectives of the transformation from the transformation itself. This separation enables the reuse of the same set of transformation rules and objective specifications for several problem scenarios.

The remainder of this chapter is structured as follows. Section 4.2 introduces a running example which we use throughout this chapter to demonstrate the different aspects of our approach. Section 4.3 provides an overview our approach, followed by Section 4.4 which details the generic encoding on which the features of our approach are based. Section 4.5 to Section 4.7 then describe these features in detail, from the specification of the objectives and constraints to the exploration configuration of the optimization method and the returned analysis results. In Section 4.8, we describe how, through the use of a dedicated configuration language, we support model engineers in selecting and configuring the respective optimization algorithms and experiment parameters. Section 4.9 outlines how the approach has been implemented, in particular, which techniques we build upon and how the framework can be used and extended. Finally, Section 4.10 describes how the approach fits into the current state of the art. The evaluation of our approach can be found in Chapter 5.

4.2 Running Example

A well-known problem in software architecture design that can often not be solved through exhaustive approaches is the modularization problem [PHY11, SSB06, HT07]. The goal is to create high-quality object-oriented models by grouping classes into modules. Solving this problem implies to decide where classes belong and how objects should interact. Producing a class diagram where the right number of modules is chosen and a proper assignment of classes is realized is a non-trivial task, often requiring human judgement and decision-making [MJ14]. In fact, the problem has an exponentially growing search space of potential class partitions given by the Bell number $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$. The nth of these numbers, B_n, counts the number of different ways a given set of n classes can be divided into modules. If there are no classes given (B_0), we can in theory produce exactly one partition (the empty set, $\emptyset$). The order of the modules as well as the order of the classes within a module do not need to be considered as the semantics of a class diagram does not depend on that order. Already starting from a low number of classes, the number of possible partitions is unsuitable for exhaustive search, e.g., 15 classes, which is rather small in a real-world example, yields $1,382,958,545$ possible ways to create modules. Even when dismissing special cases where we group all classes into one module or every class into a separate module, the search space is still enormous.

4.2.1 Metamodeling

In MDE, the first step to solve such a problem is to model the problem domain in terms of a metamodel, cf. Section 2.1.1. UML already considers this problem as part of its class diagram structure. However, for simplicity we will work on a smaller version of this metamodel, as depicted in Figure 4.2. In our modularization metamodel, a system consists of modules and classes. Classes may depend on an arbitrary number of other classes (*dependsOn-provider* relationship), i.e., if class A depends on class B, B is a provider for A. Modules are used to group classes into meaningful clusters. As we can see, each class may be in one module, while modules may contain many classes. An instance of our modularization metamodel is depicted in Figure 4.3. In the figure, the relationships between classes are shown in different colors to foster the figures readability. The depicted model is a representation of *mtunis* [Hol83], an operating system for educational purposes written in the Turing Language. This system contains 20 classes and 57 dependencies among them. As a result, there are, according to the Bell number described above, $51,724,158,235,372$ possibilities for grouping the classes into modules. Please note that the figure depicts the abstract model syntax, i.e., the graph-based representation of the model information modulo its notation.

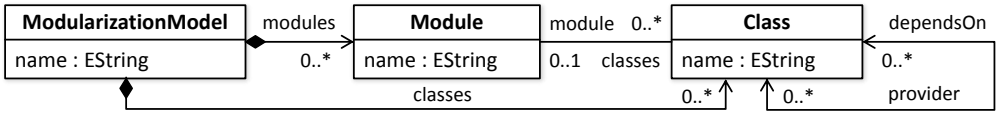


Figure 4.2: Modularization Metamodel

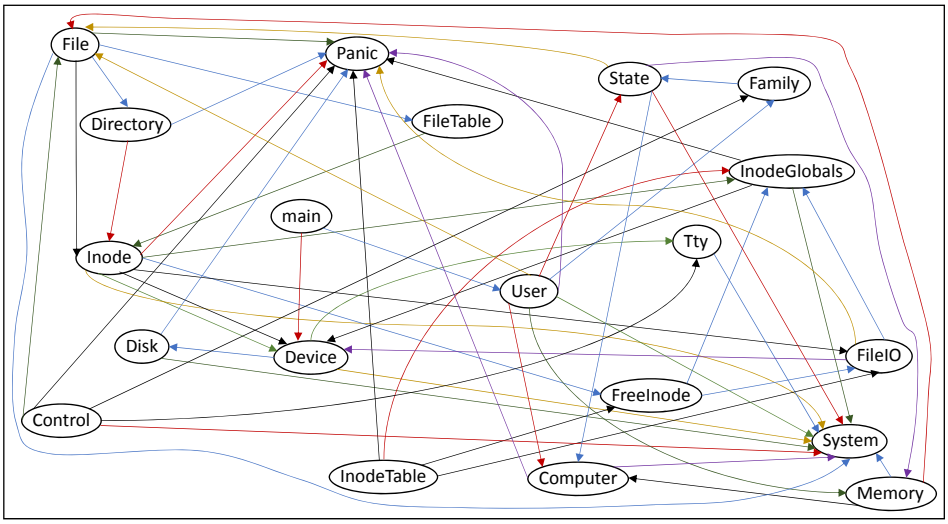


Figure 4.3: Modularization Model Instance (*mtunis* System)

4.2.2 Model Transformations

To manipulate model instances in order to group classes into modules, we propose the two rules depicted in Figure 4.4. In this figure, we use the Henshin notation, as described in Section 2.1.5. Since at the beginning there are no modules in the input model (it only contains classes and the dependencies among them), we define a rule to create a module (`createModule`). This rule creates a module with the provided name, only if a module with such a name does not already exist. Rule *assignClass* then enables the system to group a previously unassigned class into a module. All rules have parameters, namely `moduleName`, `module` and `class`. While producing a match for these rules, all these input parameters acquire a value, i.e., they are instantiated, and the rules can be applied. For retrieving such values, the graph transformation engine matches the pattern in the rules consisting of nodes and edges with the model graph. Since the `moduleName` parameter provides a value for a newly created class instance, it cannot be matched automatically and requires input from the user before it can be applied.

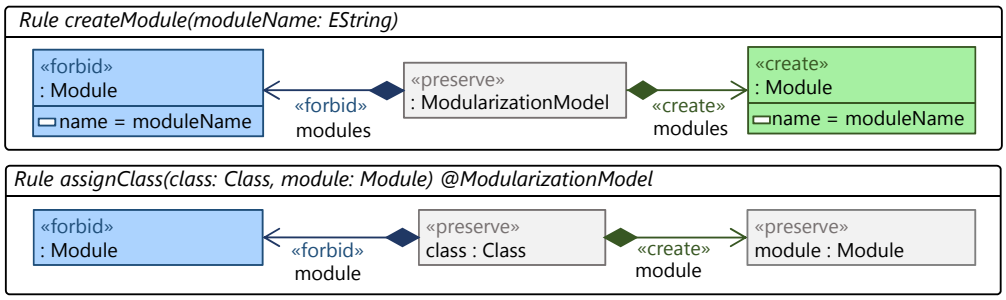


Figure 4.4: Rules to Manipulate a Modularization Model

4.2.3 Quality

For determining the quality of the obtained modularized model, we follow an *Equal-Size Cluster Approach* (ECA) [PHY11]. The ECA attempts to produce a modularization that contains modules of roughly equal size. Specifically, we use the following metrics for measuring the quality of the modularization: (1) coupling, (2) cohesion, (3) modularization quality, (4) number of modules, and (5) the difference between the maximum and minimum number of classes in a module. Please note that in general several metrics may exist to measure the same aspect of a system, but the results retrieved from these metrics do not necessarily compare to each other [ÓCTH+12]. For instance, the coupling and cohesion of a system can be calculated using several different metrics, as studied by Abdeen et al. [ADS11].

In our example, coupling refers to the number of external dependencies a specific module has, i.e., the sum of inter-relationships with other modules. This translates to the number of class dependencies (`dependsOn`) that origin in one module but end in another. Cohesion refers to the dependencies within a module, i.e., the sum of intra-relationships in the module. In our example, this reflects the number of class dependencies that origin and end in the same module. Typically, low coupling is preferred as this indicates that a module covers separate functionality aspects of a system, improving the maintainability, readability and changeability of the overall system [YC79, ADS11]. On the contrary, the cohesion within one module should be maximized to ensure that a module does not contain parts that are not part of its functionality. Ideally, a module should be a provider of one functionality [ADS11]. The calculation of coupling and cohesion used for our example is depicted in Equation 4.1 and Equation 4.2 respectively. In these equations M refers to the set of all modules and $C(m)$ refers to the set of classes contained in the module m.

$$\mathrm{COP} = \sum_{\substack{m_i, m_j \in M \\ m_i \neq m_j}} \sum_{\substack{c_i \in C(m_i) \\ c_j \in C(m_j)}} \begin{cases} 1 & \text{if } c_i \text{ depends on } c_j \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$\mathrm{COH} = \sum_{m_i \in M} \sum_{c_i, c_j \in C(m_i)} \begin{cases} 1 & \text{if } c_i \text{ depends on } c_j \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

The modularization quality (MQ) [PHY11] evaluates the balance between coupling and cohesion by combining them into a single measurement. It has been proven that the higher the value of MQ, the better the quality of the modularization [dOB11]. The aim is to reward increased cohesion with a higher MQ score and to punish increased coupling with a lower MQ score. The modularization quality is defined in Equation 4.3, where MF_m is the modularization factor for module m. The modularization factor is the ratio of intra-edges to edges in each cluster, as defined in Equation 4.3, where i is the number of intra-edges, i.e., cohesion, and j is the number of all edges that originate or terminate in

module m. The reason for the occurrence of the term $1/2$ rather than merely j is to split the penalty of the inter-edge across the two modules connected by that edge [PHY11].

$$\text{MQ} = \sum_{m_i \in M} MF_m, \text{ where } MF_m = \begin{cases} 0 & \text{if } i = 0 \\ \frac{i}{i + \frac{1}{2}j} & \text{if } i > 0 \end{cases} \tag{4.3}$$

The MQ value tends to increase if there are more modules in the system, so it also makes sense to include the number of modules in the model as an objective. Another reason to include this metric is to avoid having all classes in a single large module which would yield no coupling and a maximum cohesion. Therefore, as part of our fitness function, we also maximize the number of modules. At the same time, we minimize the difference between the maximum and minimum number of classes in a module to aim at equal-sized modules.

With the rules and quality objectives presented, it is not clear how to orchestrate the two rules depicted in Figure 4.4 in order to obtain a desired model where all measures are optimized. In fact, model transformation languages do not offer any mechanism to specify the objectives of a transformation explicitly. Besides, if objectives were encoded somehow within the transformation, e.g., by using a scheduling language for transformation rules or encoding the schedule as additional positive or negative application conditions, the transformation could only be used to solve this particular problem and only for the given objectives. On the other hand, if they are not encoded in the transformations, it is less clear how the rules affect the objectives and how they should be applied. Therefore, in next section we introduce our approach to solve this problem and in the process facilitate the usage of search-based optimization methods on model level.

4.3 MOMoT Approach

In order to realize an approach that solves this problem, we combine MDE techniques with search-based optimization methods. Instead of manually deriving an orchestration of transformations for a given scenario in a specific problem domain, we deploy dedicated optimization methods to calculate the transformation orchestration based on a given set of objectives and constraints. In fact, metaheuristic optimization methods allow us to address multi-modality problems as they aim to find the Pareto-optimal set of solutions, as opposed to trying to obtain a single optimal solution. For the modularization problem this would mean that we are interested in a set of solutions where all objectives are compensated and optimized instead of being combined into a single metric, which may not achieve optimality [PHY11].

An overview of our approach with the necessary inputs and the provided outputs is depicted in Figure 4.5. Specifically, we design our approach to meet the requirements stated in Section 1.3, i.e., the approach needs to be problem- and optimization method-agnostic, transparent to the model engineer, declarative in the specification of the objectives, and supportive by providing additional information to the model engineer.

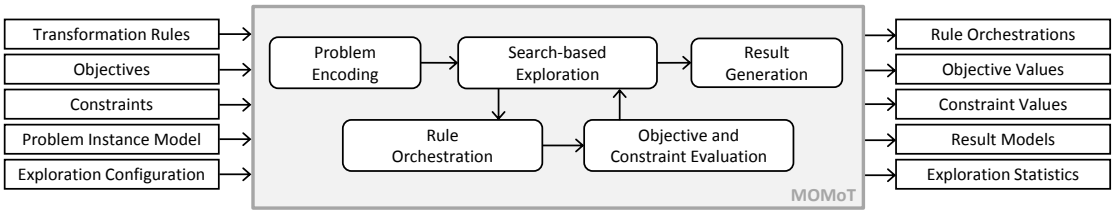


Figure 4.5: Overview of the MOMoT Approach

To realize our approach, we need the following ingredients: (*i*) a generic way to describe the problem domain and the concrete problem instance, (*ii*) an encoding for the solution of the concrete problem instance based on model transformation solutions, (*iii*) a random solution generator that is used for the generation of an initial, random individual or random population, and (*iv*) a set of search-based algorithms to execute the search. To further support the use of multi-objective evolutionary algorithms, we additionally provide (*v*) generic objectives and constraints for our solution encoding, (*vi*) generic mutation operators that can modify the respective solutions, and (*vii*) a configuration language that provides feedback about the specified search configuration. Since our approach combines MDE techniques with metaheuristics, the key building blocks are an environment to enable the creation of metamodels and models, a model transformation engine and language to manipulate those models and a set of optimization algorithms that perform a search to find transformation orchestrations that optimize the given objectives and fulfil the specified constraints. In the following, we describe the different parts of our approach on the basis of the modularization problem introduced in the previous section.

4.4 Generic Solution Encoding

As it is typical in MDE, the problem domain itself is defined as a metamodel (e.g., cf. the metamodel of the modularization problem depicted in Figure 4.2). Based on the specific problem domain, a user can define both concrete problem instances, i.e., models (e.g., cf. Figure 4.3) and transformation rules that specify how problem instances can be modified (e.g., cf. Figure 4.4). In this section, we therefore provide a generic encoding for search-based optimization methods that considers these concepts.

A solution to an optimization problem consists of a set of decision variables that are optimized by the respective search-based optimization algorithm, as described in Section 2.2. As we deal with the problem of orchestration transformation rules, there are two common ways in representing a solution. Either a solution is an ordered sequence of rule applications or it is the model resulting from the application of that sequence. We choose the first encoding as we consider it more flexible, because the resulting model can always be calculated from the sequence of configured rules and may be stored in a solution as attribute to avoid re-execution. Furthermore, in our understanding, using a rule application sequence as first-class citizen in the encoding has several advantages. First, we are in line with the general optimization encoding, where a solution consists of separate

decision variables that are optimized by an algorithm. This increases the understanding for users who are knowledgeable in optimization methods. Second, we are on the level of abstraction on which the user has provided information, i.e., transformation rules. Therefore, we give also novices in metaheuristic optimization insight into the solutions. And third, we think that having the rule sequence shown explicitly also makes the output models more comprehensible as the user can compare solutions on the level they are computed and not only based on the output model which may increase the acceptance of our approach. Therefore, a decision variable in our solution is one transformation unit. A description of all supported transformation units based on [ABJ+10] can be found in Table 4.1.

Table 4.1: Transformation Units That Serve as Decision Variables in the Encoding

Unit	**Description**
Transformation Rule	Rules are the main transformation units and the only ones that are ultimately executed on the input model. A rule consists of a left-hand side and a right-hand side graphs which describe the pattern to be matched and the changes to be made, respectively. Rules may have positive and negative application conditions and can be nested to execute inner rules as long as possible if the outer rule matches.
Sequential Unit	A sequential unit executes all sub-units in the given order once and may be configured to stop the execution if any of the sub-units can not be executed and to rollback any changes that have been made by the sequential unit.
Priority Unit	A priority unit executes the first given sub-unit that can be successfully applied on the given input model. Subsequent sub-units are not executed.
Independent Unit	An independent unit executes the first randomly selected sub-unit that can be executed on the given input model. Other sub-units are not executed.
Loop Unit	A loop unit consists of one sub-unit. The loop unit is executed as long as the sub-unit can be applied in the underlying graph.
Iterated Unit	An iterated unit is composed of one sub-unit that is executed as often as explicitly specified.
Conditional Unit	This unit enables the expression of an if-then-else condition. It consists of at least two sub-units (*if*, *then*) and an optional third sub-unit (*else*). The *then*-unit is only executed if the *if*-unit can be successfully matched, otherwise the *else*-unit is executed if possible.
Placeholder Unit	This unit acts as a placeholder within the transformation without modifying the given input model.

The most basic *transformation unit* is a transformation rule which directly manipulates the model being transformed. Other transformation units can be used to combine one or more transformation units to provide simple control-flow semantics. It is important to note that a loop unit may create an infinite search space in which our approach would not stop until it runs out of memory, e.g., when the provided sub-unit manipulates the graph in a way that produces additional matches for that sub-unit. Currently, we have no way to automatically detect whether a loop unit creates an infinite search space, but we inform the user about the possibility using our support system (cf. Section 4.8).

Each transformation unit can have an arbitrary number of parameters that need to be instantiated, i.e., values for these parameters must be found. Additionally, application conditions can be specified that need to be fulfilled in order for the unit to be applicable. As typical for graph transformation systems, there are two kinds of application conditions, positive application conditions (PACs) and negative application conditions (NACs). The former require the presence of certain elements or relationships in the model, whereas the latter forbid their presence. Besides using model elements and relationships in the PACs and NACs, it is also possible to formulate these conditions using parameter values or attributes of elements. In such a case, we allow the usage of JavaScript or OCL. As a special transformation unit, we provide transformation unit *placeholders*, i.e., units that are not actually executed and do not have any effect on the output model. This enables the actual solution length to vary in cases where the solution length must be fixed in advance. Besides the decision variables, we also provide an attribute storage for a solution where key-value pairs can hold additional information relating to the evaluation or the search process.

Decision Variables (Transformation Units)				Objectives	Constraints
unit = createModule moduleName = 'ModuleZrldp'	unit = assignClass module = ModuleZrldp class = FreeInode	*Placeholder*	...	Coupling = 66.0 Cohesion = -24.0 MQ = -1.964 MinMaxDiff = 4.0 NrModules = -6.0 Length = 33.0	UnassignedClasses = 0 EmptyModules = 0

Figure 4.6: Solution with the First Two Transformation Units and One Placeholder

In summary, one solution consists of an ordered sequence of applicable transformation units (decision variables) that creates a valid output model when executed on the specified input model. An example of a solution for the modularization problem with the first two transformation units and one placeholder is depicted in Figure 4.6. The figure also visualizes that for each solution, we store the respective objective and constraint values.

Transformation Unit Parameters. Parameters allow to change the behavior of transformation units with variable information that is typically not present before execution time. When dealing with model transformations, we can distinguish between two kinds of parameters: those that are matched by the graph transformation engine (*matched parameters*), and those that need to be set by the user (*user parameters*). The

former are often nodes within the graph, whereas the latter are typically values of newly created or modified properties. In the rules provided for the modularization problem (cf. Figure 4.4), the `class` and `module` of the `assignClass` rule are matched parameters, whereas the `moduleName` parameter of the `createModule` rule is a user parameter.

Values for matched parameters can be retrieved from the respective transformation engine through a matching procedure. In our approach, we use non-deterministic matching, i.e., for a given model a different transformation unit with different parameters may be selected each time the matching is executed. This leads to a non-deterministic optimization, where we produce different results every time the search is executed. However, during the search, the matching parameters within a solution are fixed, unless otherwise modified, i.e., a solution always produces the same output model for the given input model. Values for user parameters need to be handled differently, as a user usually can provide a value for these programmatically or via a dedicated user interface when the unit is applied manually. In an automated approach, however, we need a way to generate those values when needed. This is also necessary in order to facilitate the creation of random solutions and populations, what is needed by most search algorithms. Usually, a high variance of parameter values is preferred to cover as much area of the search space as possible. By default, we use random parameter value generators for most primitive values. The range of these value generators is the range of the data type of the respective parameter, e.g., for Integer in Java the value can range from -2^{31} to $2^{31} - 1$. Although an efficient search algorithm should quickly remove values that are not beneficial in a specific scenario, the user may restrict this range as part of the configuration to prune such unfruitful areas of the search space in advance. Additionally, we provide a hook where a user can integrate how values should be generated for specific parameters. Furthermore, the user may define which parameters should be retained as part of the solution. By default all parameters are kept, any other parameters are re-matched by the graph transformation engine when the respective unit is executed again.

Solution Repair. Even though constraints can be used to specify the validity or feasibility of solutions, a solution that is the product of re-combining two other solutions might have unit applications that are not actually executable. By default, units that can not be executed are ignored. However, this behavior might not be satisfactory in some cases as the process of establishing that a transformation unit can not be executed is quite expensive due to unnecessary match finding done by the transformation engine. Therefore we consider two repair strategies in our approach.

The first strategy replaces all non-executable transformation units with transformation placeholders and the second strategy replaces each non-executable transformation unit with a random, executable transformation unit. Which strategy is the best depends on the problem at hand. Replacing non-executable unit applications with a placeholder effectively shortens the solution length and has the risk of removing useful applications from the search-space which may cause the search to converge too early. On the other hand, replacing a non-executable unit application with a random executable unit application

may have an impact on the overall solution quality as other transformation units might become non-executable due to the new transformation unit and may also need to be replaced. The resulting solution may be a solution that is quite different from the original solution, which may negate the positive effect of selection and recombination. Therefore, this strategy should be used with caution, especially when many different transformation rules are given and the risk of producing a very different solution is higher.

In the ideal case, no solution repair strategy is necessary as no non-executable unit applications are produced. Of course this depends on the chosen algorithm and the actual constraints of the solutions. A user can also select a dedicated re-combination operator that is able to consider some constraints, e.g., the partially matched crossover (PMX) [GL85] can preserve the order of variables within a solution.

4.5 Solution Fitness

As described in Section 2.2, the quality of each solution candidate is defined by a fitness function that evaluates multiple objective and constraint dimensions. Each objective dimension refers to a specific value that should be either minimized or maximized for a solution to be considered "better" than another solution. In our approach, we can distinguish between objectives that are problem domain-specific, e.g., minimizing the coupling of modularization models, and objectives that relate to the solution encoding, e.g., minimizing the number of transformations that should be applied to reach a solution. Additionally, a solution candidate may be subjected to a number of constraints in order for the solution to be valid. Depending on the algorithm, invalid solutions may be filtered out completely or may receive a low ranking in relation to the magnitude of the constraint violation. As with objectives, we distinguish between domain-specific constraints, e.g., all classes must be assigned to a module, and solution-specific constraints, e.g., a specific transformation rule needs to be applied at least once.

Objectives and constraints can be defined either by providing a direct implementation of the respective objective value or by specifying model queries in OCL. OCL is a standardized and formal language to describe expressions, constraints and queries on models. In our approach, the support for OCL is crucial as model engineers are typically proficient in OCL and therefore can provide the necessary input in a language they are familiar with. Alternatively, users may also provide objectives and constraints in a Java-like expression language in case they are not familiar with OCL. Furthermore, objectives and constraints may also be calculated using external mechanisms such as model-based analysis techniques described in Chapter 3. Constraints may also be defined as NACs directly in the transformation units as we are building upon the expressive power of a graph transformation system. By doing so, we can effectively avoid the generation of invalid solution candidates, resulting in a potentially smaller search space. However, due to the cost of graph pattern matching, the application of NACs may also introduce an additional overhead, depending on the NACs complexity and the number of pruned solutions.

In summary, the problem domain is represented as a metamodel, from which a concrete problem instance model can be created. Transformation units defined upon concepts of the metamodel are then used to modify the model instances. The objectives that should be optimized in order to create the desired output model may be defined directly or via model queries in OCL. The constraints that a possible solution must fulfil in order to be valid can also be specified in OCL, but may also be encoded as NACs directly in the rules.

4.6 Exploration Configuration

In order to search for good solutions to the given search problem, the user must select and configure at least one optimization method and specify the parameters of the experiment configuration.

4.6.1 Optimization Method Selection

As mentioned previously, our proposed approach is optimization method-agnostic. Therefore additional method-specific exploration options need to be configured by the user. These options depend on whether it is an evolutionary algorithm or a local search algorithm. The parameters for these algorithm families are summarized in the following paragraphs. Further details on these families can be found in Section 2.2.2.

Evolutionary search algorithms are a subset of population-based search algorithms that deploy selection, crossover, and mutation operators to improve the fitness of the solutions in the population in each iteration (the first population is usually generated randomly). The *selection operators* can be defined generically and choose which solutions of the population should be considered for re-combination. An example for a selection operator would be deterministic tournament selection, which takes k random candidate solutions from the population and allows the best one to be considered for re-combination. The *crossover operator* is responsible for creating new solutions based on already existing ones, i.e., re-combining solutions into new ones. Presumably, traits which make the selected solutions fitter than other solutions will be inherited by the newly created solutions. In our case, each solution is represented as a sequence of transformation application units for which many generic operators already exist, e.g., the one-point crossover operator that splits two solutions at a random point and merges them crosswise. The *mutation operators* are used to introduce slight, random changes into solution candidates. This guides the algorithm into areas of the search space that would not be reachable through recombination alone and avoids the convergence of the population towards a few elite solutions. To take the semantics of transformation units into account, we have introduced three dedicated mutation operators. The first operator replaces random transformation units by placeholders, reducing the actual solution length. The second operator varies the user parameters of a transformation unit based on the parameters values (cf. Section 4.4). The third operator selects a random position within a solution

and replaces all transformation units after that position with a random, executable transformation unit.

Local search algorithms maintain one solution at a time and try to improve that solution in each iteration. Improvement depends on the *solution comparison method* the user selects, e.g., comparison based on objective, constraint or attribute values. The initial solution may be given by the user or can be generated randomly. In each iteration, the algorithm may take a step to a neighbor solution, i.e., a solution that is a slight variation of the current solution. The calculation of neighbors from the current solution can be done generically using a *neighborhood function*. How many neighbors are evaluated and whether only fitter neighbors are accepted as the next solution depends on the respective algorithm. In our approach, we support two neighborhood functions. Following the principle of reuse, the first function uses one of the previously defined mutation operators to introduce slight changes into the current solution. Depending on the operator, this function may produce an infinite number of neighbors, e.g., when varying floating point rule parameter values. Therefore, an upper bound on the number of calculated neighbors can be specified. The second neighborhood function adds an additional, random transformation unit to the current solution, increasing its solution length. Here, an upper bound on the solution length can be specified.

If multiple algorithms are configured by the user, each algorithm is executed individually and their results can be used separately or in combination (see Section 4.7).

4.6.2 Experiment

In order to execute the search for the selected and configured optimization methods, the user needs to provide additional experiment parameters. First, there is the population size, i.e., the number of solutions in each iteration (*population size*) for evolutionary algorithms. Second, the user must specify a stopping criterion for the algorithms, i.e., the maximum number of fitness evaluations (*maximum evaluations*) that are performed by each algorithm. The number of iterations of an algorithm is implicitly calculated by the population size divided by the maximum number of fitness evaluations. Finally, the user needs to specify the number of times each algorithm should be executed (*number of runs*). As explained by Harman et al. [HMTY10], experiments should be repeated 30-50 times in order to draw statistically valid conclusions from the results.

As part of the optional configuration parameters, a user may specify a reference Pareto front, i.e., a set of known, good objective values, to enable the analysis the search process (cf. Section 4.7). Additionally, a user may configure our approach to print detailed information about the on-going search process on the console or terminate a single run of an algorithm based on a different criteria, e.g., after a certain time limit has been reached or a given solution has been found.

As our target audience of our approach is model engineers, we aim to make the configuration of the exploration easy and provide support with respect to their configuration options (cf. Section 4.8).

4.7 Result Analysis

4.7.1 Output

Finally, after the experiment with the configured search algorithms has been executed, the user can process the results. As results we have (*i*) the set of orchestrated transformation sequences leading to (*ii*) the set of Pareto-optimal output models with (*iii*) their respective objective values, i.e., the Pareto front. Since the results are on the level of the provided input, i.e., transformation rules, the set of orchestrated transformation sequences can be inspected by the model engineer to get an understanding of the produced results. The set of output models produced by these transformations conform to the same metamodel as the given input model and can be inspected or processed further by MDE tools. The objective values may give an overview of how well the objectives are optimized by the different algorithms. All results of the experiment can be retrieved for each algorithm individually or combined into one set of Pareto-optimal solutions. Especially, the Pareto front of the combined solutions may be of interest, as they can be used as a reference set for future experiments if a Pareto front is not known a priori, which is the case for many real-world problems. Individual algorithm results may be of interest if different algorithms or different configurations of one algorithm need to be compared.

4.7.2 Statistical Analysis

This data can be used to plot graphs and give a better overview about the algorithm executions. For example, Figure 4.7 depicts the convergence of the modularization quality objective over time for multiple runs of two local search algorithms, Hill Climbing and Random Descent. Each line corresponds to one run of the respective algorithm.

If the user provides a reference set, we can also calculate several *indicators* during the search. These indicators are used to gain insight into the search performance of the different algorithms. As an example, we describe two common indicators, Hypervolume and inverted generational distance. Hypervolume corresponds to the proportion of the

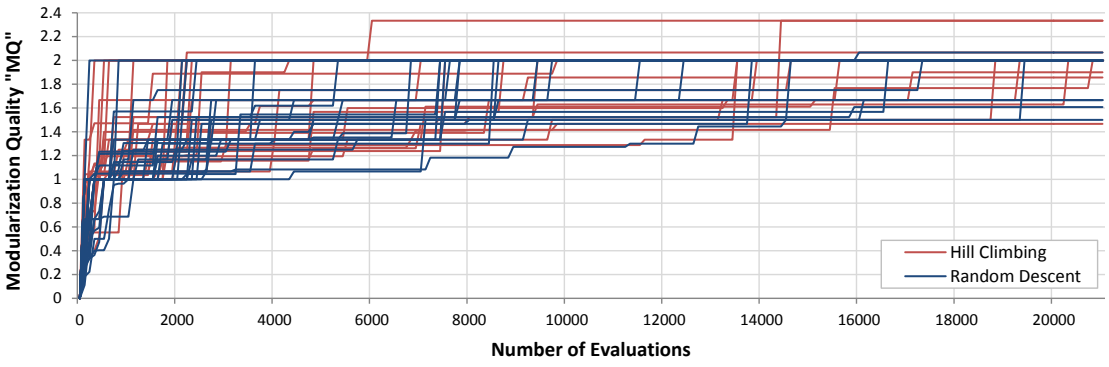


Figure 4.7: Modularization Quality Convergence Graph For Local Search

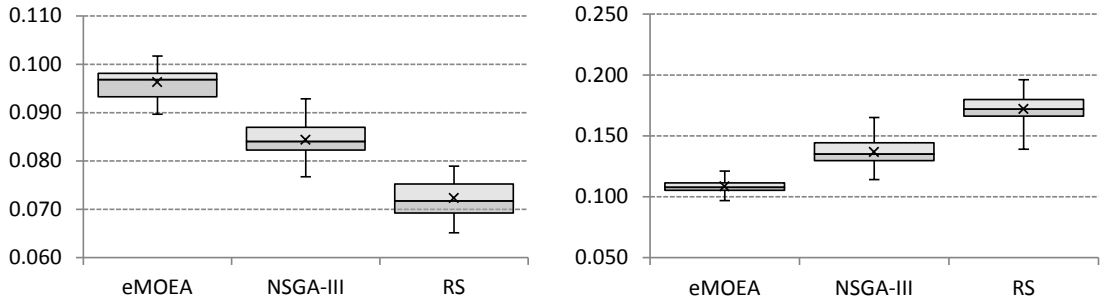
objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. The larger the proportion, the better the algorithm performs. Hypervolume can capture both the convergence and the diversity of the solutions. Inverted generational distance is a convergence measure that corresponds to the average Euclidean distance between the Pareto set produced by the algorithm and the given reference set. We can calculate the distance between these two fronts in an M-objective space as the average M-dimensional Euclidean distance between each solution in the approximation and its nearest neighbor in the reference front. Better convergence is indicated by lower values. For each run of an algorithm, we can calculate one value per indicator.

In order to draw valid conclusions from these values, we need to perform a statistical analysis, e.g., using the Mann-Whitney U test [MW47]. The Mann-Whitney U test, equivalent to the Wilcoxon rank-sum test, is a nonparametric test that allows two solution sets to be compared without making the assumption that values are normally distributed. Specifically, we test under a given significance level α the null hypothesis (H_0) that two sets have the same median against the alternative hypothesis (H_1) that they have different medians. If the resulting p-value is less than or equal to α, we accept H_1 and we reject H_0; if the p-value is strictly greater than α we do the opposite. Usually, a significance level of 95% ($\alpha = 0.05$) or 99% ($\alpha = 0.01$) is used. Alternatively, in our approach, the user may choose another test like Kruskal-Wallis One-Way Analysis of Variance by Ranks test [KW52, She03]. To also allow for a visual evaluation, we generate box plot diagrams for each indicator and algorithm.

Using these statistical tests, we can determine whether two algorithms are interchangeable with respect to a certain indicator. However, it is not possible to show the magnitude of how much one algorithm is better than another, i.e., the *effect size*. In our approach, we support the calculation of the effect size using Cohen's d statistic [Coh88] for equal sized groups. Cohen's d is defined as the difference between the two means $\overline{x_1}$ and $\overline{x_2}$ divided by the mean squared standard deviation s_i, as depicted in Equation 4.4. The effect size is considered: (1) small if $0.2 \leq d < 0.5$, (2) medium if $0.5 \leq d < 0.8$, or (3) large if $d \geq 0.8$.

$$d = \frac{\overline{x_1} - \overline{x_2}}{s}, \text{ where } s = \sqrt{\frac{s_1^2 + s_2^2}{2}} \tag{4.4}$$

For an example experiment, we use three algorithms, ε-MOEA algorithm [DMM03], NSGA-III [DJ14] and Random Search (RS), and execute each algorithm 30 times on the mtunis system with a population size of 100 and for 200 iterations. We use tournament selection with $k = 2$, a one-point crossover operator and a placeholder mutation operator with $p = 0.1$. The result of the analysis are shown in Table 4.2 and depicted in Figure 4.8. Each box plot in the figure shows the minimum value of the indicator (shown by the lower whisker), the maximum value of the indicator (shown by the upper whisker), the second quantile (lower box), the third quantile (upper box), the median value (horizontal line separating the boxes) and the mean value of the indicator (marked by an 'x') for each



(a) Hypervolume (b) Inverted Generational Distance

Figure 4.8: Hypervolume and Inverted Generational Box Plots

algorithm. We can clearly see that for the Hypervolume indicator, Random Search has lowest and therefore worst value while ε-MOEA has the highest value. A similar result is produced for the inverted generational distance where lower values are considered better. In order to investigate the results further, we deploy the Mann-Whitney U test with a significance level of 99% as described above. Based on this statistical analysis we determine that the results of all algorithms are statistically different from each other, i.e., no two algorithms perform equally well. This is also indicated in Table 4.2 (row *indifferent*) as empty set. Finally, we calculate the effect size and find the magnitude of the differences between the algorithms using Cohen's d statistic. From the results, we can see that all differences are considered large. The effect size for ε-MOEA and NSGA-III is $d = 3.44$, for NSGA-III and RandomSearch is $d = 3.21$, and for ε-MOEA to Random Search is $d = 6.96$ for Hypervolume. A similar effect size is calculated for the inverted generational distance. As a conclusion, we can state that for the settings we have chosen and based on our selected performance indicators, ε-MOEA performs the best, NSGA-III performs second best, and Random Search has the worst performance. The fact that a sophisticated metaheuristic search outperforms Random Search is also a good indicator that the problem is suitable for search-based optimization.

Table 4.2: Excerpt of Indicator Statistic for Different Multi-Objective Algorithms

	Hypervolume			*Inverted Generational Distance*		
	ε-MOEA	NSGA-III	RS	ε-MOEA	NSGA-III	RS
Min	0.089	0.078	0.065	0.097	0.114	0.139
Median	0.097	0.084	0.072	0.108	0.135	0.172
Max	0.102	0.093	0.079	0.121	0.165	0.196
Mean	0.097	0.084	0.072	0.108	0.136	0.172
StdDev	0.003	0.004	0.004	0.006	0.011	0.011
Count	30	30	30	30	30	30
Indifferent	{}	{}	{}	{}	{}	{}

4.8 Support System

4.8.1 Configuration Language

To support the model engineer in specifying the search problem, the necessary input and the experiment configurations, we provide a dedicated configuration language to use MOMoT. Conceptually, this domain-specific language is defined platform-independently using a metamodel and an expression language, making the specified configuration of a model. In fact, all configurations can be made in a single file with a few lines of mandatory configuration parameters and some optional configuration parameters. In general, the textual notation of the language uses '=' to assign values, '{}' to define objects, '[]' to denote lists and ':' to separate key-value pairs. Content-Assist provides the next allowed configuration parameters and guides the users when they need to provide their own input. Furthermore, when the user is required to implement specific behavior, we provide helper objects with additional convenience methods. An excerpt of a configuration is depicted in Listing 4.1. Using such configuration, we can either generate code or interpret the configuration directly to execute the search.

Listing 4.1 Experiment Configuration Excerpt in the Textual Notation of Our DSL

```
1: model = "modularization_model.xmi" // has a reference to the metamodel
2: transformations = { modules = [ "modularization.henshin" ] }
3: fitness = {
4:    objectives = { ...
5:       NrModules : maximize "self.modules->size()" // OCL-specification
6:       Length : minimize new TransformationLengthDimension // generic objective
7:    }
8:    constraints = { ...
9:       UnassignedFeatures : minimize { // Java-like syntax, direct calculation
10:          (root as ModularizationModel).classes.filter[c|c.module == null].size }
11:   }
12: }
13: algorithms = { ... // moea is an auto-injected objects
14:    Random  : moea.createRandomSearch()
15:    NSGAIII : moea.createNSGAIII(
16:       new TournamentSelection(2), // k == 2
17:       new OnePointCrossover(1.0), // 1.0 == 100%, always do crossover
18:       new TransformationPlaceholderMutation(0.15)) // 15% mutation
19: }
20: experiment = {
21:    populationSize = 300
22:    maxEvaluations = 21000
23:    nrRuns = 30
24:    referenceSet = "reference_objectives.pf" // if available (optional)
25:    collectors = [ hypervolume ] // collect during search, needs referenceSet
26: }
27: results = {
28:    objectives = {
29:       outputFile = "data/output/all/modularization_model.pf"
30:    }
31:    models {
32:       algorithms = [ NSGAIII ]
33:       outputDirectory = "data/output/nsgaiii/modularization_model/"
34:    }
35: }
```

4.8.2 Static Analysis

One major advantage of providing a DSL over providing an API is that we can statically check the given configurations before we compile and execute them. These static analysis checks can be used to *inform* the user about certain aspects of the search, *warn* about uncommon configurations and prohibit the execution in case any *errors* can be detected. These checks are integrated in the editor of the DSL providing the details on the respective positions in the text. Additionally, an overview is used to show all information in a list. A summary of all provided checks is shown in Table 4.3. The upper part of the table shows checks related to the search, while the lower part shows consistency checks. Dependencies between the individual configurations are considered by the editor and invalid configurations are marked and prevent execution. Please note that some of these checks are more complex than others, e.g., checking if any of the given transformation units can be applied can only be answered if we actually load the model and transformations and try it out. Therefore, not all checks are done on the fly and the user needs to explicitly trigger the more complex validation process.

Table 4.3: List of Static Checks in the Configuration DSL

Name	**Type**	**Description**
Unit Applicability	Error	Check if any of the given transformation units can be applied.
User Parameters	Error	Check if values are provided for user parameters.
Algorithm Runs	Warning	Warn if the number of runs less than the recommended minimum (< 30) [HMTY10].
Number of Iterations	Warning	Warn if the resulting number of iterations is very small (≤ 10).
Many-Objective	Warning	Warn if we have many-objective search (> 3 objectives) [DJ14] but a multi-objective or local search algorithm has been chosen.
Object Identity	Warning	Warn about missing `equals` implementation for parameters.
Population Size	Warning	Warn if the population size is very small (≤ 10).
Algorithm Parameters	Info	Inform about the relevance of different algorithm parameters, e.g., about specific evolutionary operators.
Loop Unit	Info	Inform the user about the possibility of a potential infinite search space created through a loop unit.
Single-Objective	Info	Inform about local search algorithms if only a single objective is used.
Algorithm Name	Error	Check if every algorithm name is unique.
Input Model	Error	Check if the specified input model exists.

OCL Dimension	Error	Check if the given dimension are valid OCL queries.
Parameter Value	Error	Check if parameter values are defined for existing parameters.
Parameter Value Name	Error	Check if parameter values are not defined twice for the same paramter.
Reference Set	Error	Check if the given reference set file exists.
Transformations	Error	Check if the specified transformations exist.
Save Analysis	Warning	Warn if analysis should be saved but has not been defined.
Result Analysis	Info	Inform when an existing analysis file will be overridden.
Result Objectives	Info	Inform when an existing objectives file will be overridden.

4.9 Implementation

In order to show the feasibility of our approach, we provide an implementation as Java framework. In this section, we provide an overview of the technology stack, the core classes, and the implementation of the configuration DSL. The complete code of MOMoT with further explanations as well as the case studies presented in this thesis can be found on our project website [FTW16a].

4.9.1 Technology Stack

While it is possible to implement our MOMoT approach from scratch, reusing the functionality of existing frameworks as much as possible is a central principle of our implementation. Reuse avoids the necessity for users to learn new formalisms and reduces the risk of introducing additional errors through re-implementation. Furthermore, there is little to no delay in receiving updates for bug-fixes, new functions, algorithms, or optimizations from the existing frameworks. Specifically, to unify the MDE and search-based optimization worlds in a single framework, we bridge the Eclise Modeling Framework (EMF), the Henshin graph transformation system, and the MOEA Framework. For realizing the MOMoT configuration language, we build on the functionality of XBase. The resulting technology stack is depicted in Figure 4.9.

EMF. The Eclipse Modeling Framework (EMF) [Ecl16b] is an open-source, Eclipse-based framework that supports the creation of modeling and metamodeling tools. At the core of EMF is Ecore, a meta-metamodel to define metamodels and DSLs. Ecore is the de facto reference implementation of the Essential MOF standard in Java. By basing our implementation on EMF, we can reuse many existing frameworks. Particularly, we use Eclipse OCL to evaluate OCL queries and constraints on Ecore-based models, and the EMF Validation Framework to implement the support checks of our configuration DSL.

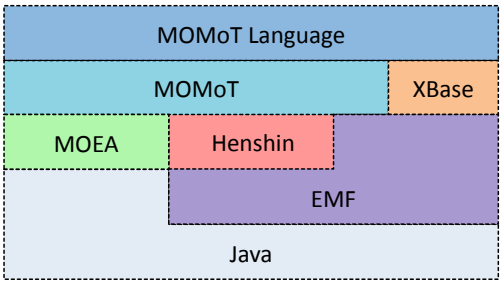


Figure 4.9: Technology Stack of MOMoT

Henshin. In our implementation, we use Henshin [ABJ+10] as model transformation language and the accompanying transformation engine. Henshin has already been described and applied in Section 2.1.5. An example of the visual notation of a Henshin transformation rule has been shown in Figure 4.4. Besides rules, Henshin provides units to orchestrate these rules, e.g., sequential units, priority units or amalgamation units [BET10]. These units cover all transformation units proposed for our generic solution encoding in Section 4.4

MOEA Framework. The MOEA framework is an open-source Java library which provides a set of multi-objective evolutionary algorithms with additional analytical performance measures and which can be easily extended with new algorithms. By reusing the MOEA framework, we can use the following evolutionary algorithms out of the box: NSGA-II, eNSGA-II, NSGA-III, eMOEA, and Random Search. Furthermore a set of selection and crossover operators are provided, which can also be reused. A user may choose to develop further algorithms or integrate existing ones from the jMetal library[1], the PISA library[2] and the BORG MOEA Framework[3], for which adapters or specific plug-ins are already provided by MOEA.

In order to also support local search algorithms, we provide a base interface and implementation as well as the described neighborhood functions within MOEA. More precisely, we demonstrate the use of our local search hooks by implementing Random Descent and Hill Climbing as proof-of-concept algorithms.

Xbase. Xbase [Ecl16d] is a statically typed expression language similar to Java. Among the main differences to Java are the lack of checked exceptions, the automatic type inference, lambda expressions, and the fact that everything is an expressions, i.e., there are no statements like in the Java language. Xbase itself is implemented in Xtext [Ecl16f], a language development framework for defining the abstract and the textual concrete syntax of a modeling language that also covers a languages infrastructure such as parsers,

[1] `http://jmetal.sourceforge.net`
[2] `http://www.tik.ee.ethz.ch/pisa/`
[3] `http://borgmoea.org/`

linkers, compilers and interpreters. Xbase is described as a partial programming language and is intended to be embedded and extended in other DSLs written in Xtext. In our implementation we embed Xbase in our MOMoT configuration language to enable the user to not only specify the configuration parameters but also implement behavior directly in the language if necessary, e.g., the calculation of objective values.

4.9.2 Core Classes

In this section, we present the core classes that are needed to connect the different frameworks in order to encode the model transformation problem, and the central classes that a user needs to use in order to solve such a problem through search. Please note that for most of the discussed classes, only their core functionality is presented. For more details on a specific class or on other involved classes, we kindly refer the reader to our project website [FTW16a].

Solution Definition

In order to encode a solution, we need to define decision variables that are optimized by the search algorithm (cf. Section 2.2.1). The solution and decision variable classes are implemented as depicted in Figure 4.10.

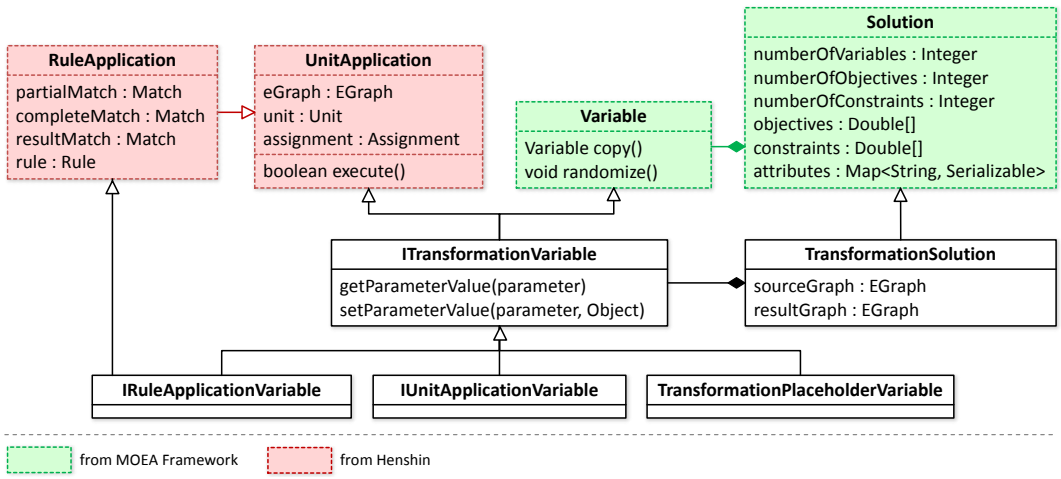


Figure 4.10: Solution Definition in MOMoT

Transformation Variable. The main components of each solution are the decision variables which are optimized by the search algorithm. For the model transformation problem, we distinguish between three decision variables. First, we have unit application variables which represent a configured transformation unit, i.e., a unit where all parameters have a dedicated value. Second, we have a rule application variable which represents a configured transformation rule. While in general, we consider a rule to be the most basic form of a transformation unit (cf. Section 4.4), in the implementation we distinguish

these two units as it is done in Henshin. This distinction comes from the way the unit parameters are handled. In order to create a rule application, the transformation engine needs to find a match in the underlying graph to lock values for the *matched parameters*. This is not necessary for the other transformation units, where the parameters are either user parameters or come from a rule application within the unit. In general, before a unit application can be created, the *user parameters* need to be assigned; `assignment` in the unit application, and `partialMatch` in the rule application. Only if all parameters have values, is a unit application considered to be complete and applicable. The final decision variable type we consider is placeholder variables. These variables do not have an impact when applied on any graph and may be used to vary the solution length.

Transformation Solution. A solution to the model transformation problem consists of a vector of decision variables, i.e., configured model transformations units, a vector of objectives values that reflect the quality of the solution, a vector of constraint values that indicate whether a solution is feasible or not, and a map of key-value pairs (`attributes`) to store additional information about a solution. The size of the variable, objective, and constraint vector needs to be fixed when the solution is created. Therefore, in order to vary the length of the solution, placeholder variables can be used which do not towards the calculated solution length but still take a place in the variable vector. Furthermore, each solution holds a reference to the graph that represents the problem instance (`sourceGraph`) and a solution graph that is produced when the vector of decision variables is executed on the problem instance (`resultGraph`).

Problem Definition

In order to represent the model transformation problem and use the MOEA framework, we need the following three main classes, as depicted in Figure 4.11.

Search Problem. The main class in the MOEA framework is the `Problem` class. It specifies how many variables a dedicated solution has, the number of constraints that need to be satisfied in order for a solution to be feasible, and the number of objectives that a search algorithm needs to optimize. Furthermore, it is also responsible for evaluating the solution and generating new solutions. As in the model transformation problem, these two tasks are more heavy-weight as it involves the derivation of matches and the evaluation on model level, we delegate these tasks to two dedicated classes. Namely, a fitness function for the evaluation of the objectives and constraints and a solution generator to create new solutions. Separating these two concerns enables us to quickly switch between different fitness functions within a dedicated problem.

Fitness Function. The fitness function interface is responsible for evaluating a given solution based on the user-defined objectives and constraints. The central implementation of this interface in the MOMoT framework is the `EGraphMultiDimensionalFitness-Function` which operates on the graph-based representation of the model used in Henshin

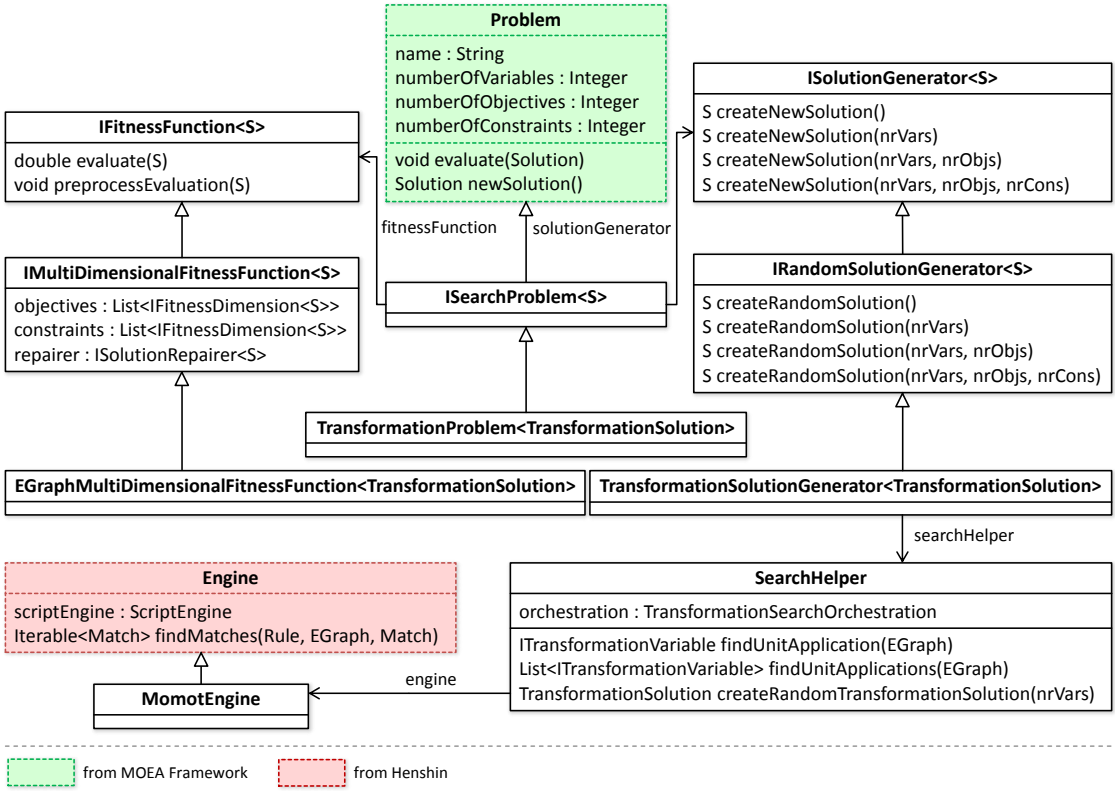


Figure 4.11: Transformation Problem Definition in MOMoT

and which takes several separate objective and constraint dimensions. Each dimension corresponds to one objective and constraint in the multi-dimensional objective space, has an optimization direction, i.e., minimization or maximization, and is identified by a unique name. During the evaluation, each dimension produces a `double` value which is stored in the objective and constraint vector of the solution. A user can provide dimensions either as a separate class, as an anonymous Java function (or Xbase expression in the configuration DSL), or as an OCL expression that is evaluated on the root element of the problem model. For each solution evaluation, a preprocess method is called which may be used to calculate the objective and constraint values for all dimension in one run and store the results in the attributes of the solution to avoid recalculation. The fitness function also holds the repair mechanism for solutions in case the search operators produce a non-feasible solution that needs to be adapted.

Solution Generator. The second main concern for a problem is the generation of new, random solutions as this functionality is needed by most algorithms in order to provide an initial solution or an initial population. Since we are dealing with model transformations, we leverage the use of a model transformation engine to create these random solutions. In MOMoT, we implement a dedicated `MOMoTEngine` which extends the transformation

engine from Henshin with additional scripting capabilities, i.e., the usage of OCL inside of negative application conditions. In order to produce a random solution, we take a random transformation unit from the transformation orchestration (cf. Section 4.9.2) and find matches for this unit using the transformation engine. If no matches are found, another unit is randomly selected. If no matches can be found for any of the transformation units, we produce solutions which only consist of placeholder variables. If a match is found, the match gets converted into a transformation variable usable in the search process. This process is repeated until the given number of transformation variables have been produced and a solution consisting of these variables is returned.

Search Execution

In order to use MOMoT, detailed knowledge about how the solution and the problem is encoded is not necessary. Instead, we provide the following five classes which can be used to execute the complete search process, cf. Figure 4.12

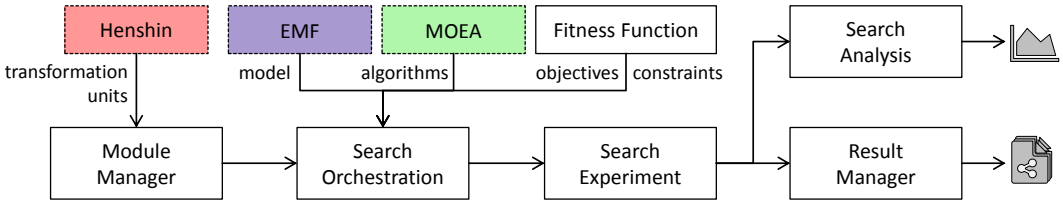


Figure 4.12: Involved Classes for Executing a Search in MOMoT

ModuleManager. The `ModuleManager` is responsible for loading and configuring the Henshin transformation units. Therefore, the user needs to provide a set of Henshin modules (`*.henshin` files) The manager then loads all the modules and provides an interface to query specific modules, transformation units, and unit parameters in a name based fashion. Using the module manager a user may select which transformation units should be considered during the search process; by default all units are considered. Furthermore, a user can specify how user parameters can be calculated, i.e., parameters that are not matched by the graph transformation engine. In order to ease this specification, a set of convenience classes provide methods to create random parameter values for the Java primitive types.

Transformation Search Orchestration. The search orchestration class is the main configuration class for the search problem configuration. It takes as input the problem model as `xmi` file, the module manager containing the transformation units that can manipulate the problem model, a solution length to specify how many transformation units should be used in a solution, a fitness function for evaluating the objectives and constraints on each solution, and a set of algorithms that should be deployed for the search. To ease the definition of a fitness function, a dedicated class may be used (cf. Section 4.9.2). For the specification of the used algorithms, two factory classes

are provided by the search orchestration, `EvolutionaryAlgorithmFactory` and `LocalSearchAlgorithmFactory`. These factory classes provide several overloaded convenience methods to create fully configured search algorithms.

Due to the way Java and EMF handle objects by default, two object references are only equal if they are referring to the same JVM object. This behavior, however, yields a problem if an object is used as parameter value matched in one graph, but then applied on another graph, e.g., when two solutions are recombined, as the object does not conform to any object in the new graph. The problem of having two different model elements which refer to the same objects in the real world has also been identified by Langer et al. [LWG+12]. They propose to identify objects explicitly by their natural identifier, i.e., their properties which make them unique in the real world. This explicit object identity can be implemented through *equals* method where the criteria for the equality of two objects can be given. For example, in the modularization problem, we can specify that two classes are equal if they have the same fully qualified name as this is also the identification method used in Java and most software systems [QGC00]. Unfortunately, if for some reason, we do not have access to the implementation of the model elements that are used in the parameters, this solution can not be applied, e.g., when working with UML or Ecore classes which are already available in their compiled form. We therefore provide a hook for the user to provide a method to implement a dedicated identification strategy through the `IEObjectEqualityHelper` interface. This interface has one method that is equivalent to the standard `equals` method in Java. The helper instance is specified once in the transformation search orchestration and injected into each individual solution.

From the provided input, the search orchestration class produces the search problem formulation, instantiates the solution generator with the respective search helper, and registers the configured algorithms.

Search Experiment. The search experiment class handles the execution of the configured algorithms on the specified problem. It takes the orchestration class as input and needs the following execution parameters: the maximum number of fitness evaluations that should be done in each algorithm run and the number of runs per algorithm. Optionally, a user can specify *progress listeners* that are called after each algorithm iteration, e.g., to provide information about the search on the console. Furthermore, a user may be interested in certain search performance indicators such as Hypervolume or Inverted Generational Distance. In this case, the user must specify the respective *collectors* to record these indicator values during the search and may provide a reference set for the indicator calculation (cf. Section 2.2.1). After executing the search experiment, we can retrieve all the Pareto sets of solutions for each algorithm and the performance indicators collected during the search.

Result Manager. The result manager is responsible for handling the Pareto set of solutions and provides methods to print these solutions or save these solutions in files. Specifically, we can save the actual solutions, i.e., the sequence of transformation

applications, in a text file, we can save the models resulting from the transformations as model xmi-files which can be further processed by other EMF tools, and we can save the Pareto front, i.e., the objectives of the solutions in a separate file which can be used as a reference file in future experiments and which gives a good overview over the found solutions. The user may choose to group different sets of solutions based on the respective algorithm, e.g., we may save the Pareto set solutions of each algorithm separately to compare the solution sets afterwards or we may save the Pareto set of solutions of all algorithms to cover as much of the true Pareto front as possible.

Search Analysis. The search analysis class enables the user to perform statistical analysis of the performance indicators recorded for each algorithm. Here we build upon the analysis functionality provided by the MOEA framework and can use, for instance, the Mann-Whitney U test [MW47] (cf. Section 4.7). The class takes the respective experiment as input and the user needs to specify significance level that should be used in the statistical analysis. The test is then applied on the values of each algorithm and indicates whether the results of two algorithms are significantly different. Furthermore, we calculate the effect size based on Cohen's d statistic [Coh88] to not only provide the user with the information that two algorithms are significantly different, but also about the magnitude of this difference. In addition, we generate box plots that show the distribution of the different indicator values for each algorithm. All these tests and artifacts aim to support the user in reasoning about which configured algorithm may be better suitable for a given problem.

4.9.3 MOMoT Configuration DSL

In order to support the user in defining a problem and a search experiment, we provide a dedicated configuration DSL, cf. Section 4.8. We make the configuration DSL executable by providing the semantics of the language through a mapping of the domain concepts to concepts of the Java virtual machine. Using this DSL, the user can make all necessary specification and configurations in a single file and receives additional support in the form of validation informations, warnings, and errors. In the following sections, we give an overview of grammar of the DSL containing the textual concrete syntax, the mapping procedure, and how validation rules can be specified. The complete specification of the grammar and an instance of this grammar for the modularization case study can be found in Appendix A

Language Grammar

Our DSL covers all concepts that have been discussed in the core classes related to the search execution in Section 4.9.2. We develop our DSL using a generative approach using Xtext. This means, we specify the textual concrete syntax of our language together with its abstract syntax and generate the respective metamodel from the language grammar. Conceptually, we build our language on the expression language Xbase which already has defined execution semantics through a mapping to the concepts of the Java virtual

machine. Using Xbase, we can provide an expressive language to the user that resembles the syntax of Java and enables him to specify behavior directly in the model, e.g., for calculating objectives. An excerpt of the specification of our configuration DSL is shown in Listing 4.2.

Listing 4.2 Excerpt of the Configuration DSL Grammar

```
1: grammar at.ac.tuwien.big.momot.lang.MOMoT with org.eclipse.xtext.xbase.Xbase
2:
3: import "http://www.eclipse.org/xtext/xbase/Xbase" as xbase
4: generate momot "http://www.big.tuwien.ac.at/momot/lang/MOMoT"
5:
6: MOMoTSearch:
7:    ("package" package=QualifiedName)?
8:    importSection=XImportSection?
9:    variables += VariableDeclaration*
10:   ("initialization" OpSingleAssign initialization=XBlockExpression)?
11:   "search" (name=ValidID)? OpSingleAssign searchOrchestration=SearchOrchestration
12:   "experiment" OpSingleAssign experimentOrchestration=ExperimentOrchestration
13:   ("analysis" OpSingleAssign analysisOrchestration=AnalysisOrchestration)?
14:   ("results" OpSingleAssign resultManagement=ResultManagement)?
15:   ("finalization" OpSingleAssign finalization=XBlockExpression)?;
16:
17: VariableDeclaration:
18:   "var" type=JvmTypeReference? name=ValidID (OpSingleAssign init=XExpression)?;
19:
20: FitnessDimensionSpecification:
21:   FitnessDimensionConstructor | FitnessDimensionXBase | FitnessDimensionOCL;
22:
23: enum FitnessDimensionType:
24:   MINIMIZE = "minimize" | MAXIMIZE = "maximize";
25:
26: FitnessDimensionConstructor:
27:   name=ValidID OpKeyAssign type=FitnessDimensionType call=XConstructorCall;
28:
29: FitnessDimensionXBase:
30:   name=ValidID OpKeyAssign type=FitnessDimensionType value=XBlockExpression;
31:
32: FitnessDimensionOCL:
33:   name=ValidID OpKeyAssign type=FitnessDimensionType query=XStringLiteral
34:   ("{" defExpressions += DefExpression* "}")?;
35:
36: DefExpression:
37:   "def" expression=STRING;
```

In this listing, we define that our grammar uses Xbase as base and generates a metamodel with the uri `http://www.big.tuwien.ac.at/momot/lang/MOMoT`. The entry parser rule of our grammar is called `MOMoTSearch` and also represents the root element in our metamodel. Rules starting with an 'X' are imported from the Xbase language. The root element in our DSL has, besides the configuration of the core classes, an optional package declaration (indicated by the question mark `?`), an optional import section, contains a set of zero or many variable declarations (indicated by the operator `+=` and the asterisk `*`) and has optional initialization and finalization block in which the user may define behavior that is executed before the search is started and after the analysis and result management has been performed, e.g., in order to register the metamodel in a standalone setup or perform cleanup actions. In our DSL, we use '=' to assign values,

'{}' to define objects, '[]' to denote lists and ':' to separate key-value pairs. A variable declaration consists of the keyword 'var', an optional type reference, a name and an optional initialization of the variable value. In this listing, we also show the different capabilities to specify fitness dimensions. A fitness dimension may be specified through an explicit constructor call to an external fitness dimension class, a block expression returning the double value of the fitness evaluation, or through an OCL query with an optional set of definition expressions. In order to ensure that the constructor call actually creates a valid instance of our fitness dimension class, we map the dimension specification to a Java method that returns such an instance. Through this mapping, we automatically generate the Java code for this specification and any errors produced by an invalid instance are mapped back to the textual notation of our DSL.

From this language specification, Xtext generates the metamodel containing the abstract syntax of the language, the model classes reflecting the abstract syntax, a parser and compiler for reading and writing models conforming to the textual concrete syntax, and a textual editor. The textual editor is opened whenever a file with a dedicated file extension is opened. For the MOMoT configuration files, we use the extension `*.momot`. Furthermore, Xtext enables us to easily implement the formatting of the textual models, the scope of elements, the validation for elements, a quickfix mechanism for common problem, and content assist. Additionally, Xbase provides a hook for mapping our own domain concepts to JVM concepts to automatically generate JVM code for our language.

Mapping to JVM Concepts

In order to map our domain concepts to JVM concepts, we need to provide an implementation of the `JvmModelInInferrer` class. Using the JVM concepts, we can automatically generate Java code which is evaluated by the Java compiler. Any errors, warning or information messages that are raised on the Java code level are translated to our DSL level using the provided mapping. Of course, additional validation checks specifically for our DSL can also be provided, cf. Section 4.9.3.

An excerpt of our implementation is depicted in Listing 4.3. In this listing, we show how the root element of our model is mapped to a class and any contained elements become fields, methods, and parameters. In particular, we show how the specified solution length in the search orchestration is mapped to a protected, final, and static field with the name `LENGTH`. The type of this field is Integer (`int`) and the field is initialized with the expression given in the attribute `solutionLength` of the orchestration. Using this mapping, Xbase is now able to raise an error if the expression given by the user does not compile to a statement that can be assigned to an integer field.

In a similar manner, variable declarations are mapped to fields. However, for variable declarations, the name and the type of the fields may come from the user. If the user did not define a dedicated type, we need to infer the type from the given value expression (`init`). If the value cannot be inferred, we assume variables of type `String`. Please note that in the Xtend language, the right hand side of the Elvis operator `?:` is only

evaluated if the left hand side evaluates to null. Similarly, Xtend provides the null-safe navigation operator `?.` that only accesses the attribute or method if the object is not null.

If the user has provided an initialization expression, we map this expression to a static method in the class. The body of this method is simply the expression given by the user.

Listing 4.3 Excerpt of the Mapping From DSL Concepts to JVM Concepts

```
1: def dispatch void infer(MOMoTSearch search, IJvmDeclaredTypeAcceptor acceptor,
2:                         boolean isPreIndexingPhase) {
3:    acceptor.accept(search.toClass(searchClassName)) [ // map search to class
4:       // fields - example: solution length, variable declarations
5:       members += search.searchOrchestration.solutionLength.toField(
6:          "LENGTH", typeRef(int)) [ // name: LENGTH, type: int
7:          static = true
8:          final = true
9:          visibility = JvmVisibility::PROTECTED
10:         initializer = search.searchOrchestration.solutionLength
11:      ]
12:      for (declaredVariable : search.variables) {
13:         val type = declaredVariable.type ?: declaredVariable?.init?.inferredType
14:                    ?: typeRef(String) // default type
15:         members += declaredVariable.toField(declaredVariable.name, type) [
16:            static = true
17:            visibility = JvmVisibility::PROTECTED
18:            initializer = declaredVariable.init
19:         ]
20:      }
21:      // methods - Example: initialization and main method to run
22:      if(search.initialization != null)
23:         members += search.toMethod("initialization", typeRef(void)) [
24:            static = true
25:            body = search.initialization
26:         ]
27:      val searchType = typeRef(searchClassName).type
28:      members += search.toMethod("main", typeRef(void)) [
29:         parameters += search.toParameter("args", typeRef(String).addArrayType)
30:         varArgs = true
31:         static = true
32:         body = new StringConcatenationClient() {
33:            override protected appendTo(TargetStringConcatenation it) {
34:               if(search.initialization != null) appendLine("initialization();");
35:               appendLine(searchType, " search = new ", searchType, "();");
36:               appendLine("search.performSearch(MODEL, LENGTH);")
37:               if(search.finalization != null) appendLine("finalization();");
38:            }
39:         }
40:      ]
41:   ]
42: }
```

And finally, to make our model executable, we specify a dedicated main method in its JVM representation. In this method, we can not refer to any expressions defined by the user to specify the method body. Instead, we must define our own expression. In order to avoid working with the abstract syntax tree of an expression, Xbase allows us to specify the expression as text and parses it correctly into an expression. In MOMoT, the main method of a search calls the provided initialization and finalization method if

necessary, instantiates an object of the search class and performs the experiment. An UI extension in Eclipse enables the user to simply right-click on a MOMoT configuration file to execute the defined experiment.

Model Validation

Before the search is executed, we evaluate whether the configuration provided by the user is valid or not. The EMF Validation Framework provides three types of validation markers: *errors*, *warnings*, and *informations*. Any configuration that has at least one error marker is considered invalid and we therefore prevent the execution of such an experiment.

In the EMF Validation Framework and its integration into Xtext, we can attach information, warning, and error markers to our model elements by providing a dedicated validator class. In this class, we provide one method for each of the validation checks described in Section 4.8. The validation class iterates over the complete model tree and automatically calls the validation methods (indicated by a `@Check` annotation) that take such an element as parameter. Listing 4.4 demonstrates how the rule for the validation of the number of algorithms is implemented in Xtend.

Listing 4.4 Example Validation Rule for a Warning on Algorithm Runs

```
1: @Check
2: def checkAlgorithmRuns(ExperimentOrchestration it) {
3:    val intNrRuns = interpreter.evaluate(it.nrRuns)
4:    val runs = CastUtil.asClass(intNrRuns, typeof(Integer))
5:    if(runs == null)
6:       return; // not specified or can not be interpreted as number
7:    if(runs < 30)
8:       warning("Since we are using metaheuristics, at least 30 runs should be " +
9:          "specified to draw statistically valid conclusions from the results.",
10:         it, MomotPackage.Literals.EXPERIMENT_ORCHESTRATION__NR_RUNS)
11: }
```

In this listing, we validate part of the experiment orchestration by interpreting the expression specified by the user for the number of runs. Through the mapping to the JVM concepts, we ensure that the type of the number of runs is compatible to an Integer. Otherwise, the element is already marked with an error with a detailed error message explaining the problem. However, an element might have more than one marker, e.g., a warning and an error. Therefore, we need to ensure that we only check the given if it actually is an Integer value. If this is not the case, we let the remaining validation methods handle the problem. If we can interpret the expression as an Integer value, we check whether the user has specified at least the recommended number of algorithm runs, i.e., 30 runs. If not, we provide a warning with the given message on the element in the experiment orchestration (`it`) that is reachable through the `NR_RUNS` reference provided through the package literals. This message is then displayed on the respective element in the editor. Similarly, the remaining validation checks are implemented.

4.10 Related Work

With respect to the contribution of this work, we discuss three main threads of related work. First, we review the application of search-based techniques for generating model transformations from examples which has been the first application target of search-based optimization (SBO) methods concerning model transformations. Second, we discuss approaches which apply search-based techniques to optimize models. Finally, we survey work done in the related field of program transformation.

Generating Model Transformations through SBO. An alternative approach to develop model transformations from scratch is to learn model transformations from existing transformation examples, i.e., input/output model pairs. This approach is called model transformation by example (MTBE) [Var06, WSKK07, KLR+12] and several dedicated approaches have been presented in the past. Because of the huge search space when searching for possible model transformations for a given set of input/output model pairs, search-based techniques have been applied to automate this complex task [KSB08, KBSB10, KSBB12, BSC+14, FSB13, SHNS13]. While MTBE approaches do not foresee the existence of model transformation rules, on the contrary, the goal is to produce such rules, we discussed in this work the orthogonal problem of finding the best sequence of rule applications for a given set of transformation rules in combination with transformation goals. Furthermore, MTBE approaches are mostly concerned with out-place transformations, i.e., generating a new model from scratch based on input models, while we focused in this work on in-place transformations, i.e., rewriting input models to output models. Finally, the authors in [SKT13] propose the use of search-based optimization in MDE for optimizing regression tests for model transformations. In particular, they use a multi-objective approach to generate test cases, in the form of models that are the input for testing updated transformations. In our work we assume to have correct model transformation rules available as a prerequisite but foresee as a possible future work the inclusion of oracle functions for model transformations in the search process.

Optimizing Models through SBO. Searching for transformation rule applications with search-based optimization methods for high-level change detection has been presented in [bFKLW12]. In the scenario of high-level change detection, the input model and the output model are given as well as the possible transformation rules. The goal is to find the best sequence of rule applications which give the most similar output model when applying the rule application sequence to the input model. In other words, the high-level change detection we have investigated previously is a special case which is now more generalized in the proposed framework by having the possibility to specify arbitrary goals for the search. Another combination of model engineering and metaheuristic optimization is presented in [EWZ14], however, in this framework the possible changes to the models are not defined as transformation rules, but are generally defined directly on the generic genotype representations of the models. Similarly, the authors in [MHF+15] present an

approach to extend the model API generated from metamodels to provide an optimization layer on top of models. This layer provides generic optimization concepts such as a fitness function and concepts from evolutionary algorithms such as selection, mutation and crossover operators which can be implemented by the user as Java classes using the domain concepts defined by the metamodel.

Searching for model transformation results is currently supported by approaches using some kind of constraint solver. For instance, Kleiner et al. [KDDFQS13] introduce an approach they call *transformation as search* where they use constraint programming to search and produce a set of target models from a given source model. Another approach is proposed by et al. [GHH14] where so called transformation models are defined with OCL and then translated to a constraint solver to find valid output models for given input models. Compared to MOMoT, these approaches aim for a full enumerative approach where some concrete bounds for constraining the search space have to be given. Furthermore, these approaches search for models fulfilling some correctness constraints, but finding optimal models based on some objectives is not natively supported in such approaches. In [DGM11, DGM15] an approach extending the QVT Relations language is presented which also foresee the inclusion of transformation goals in the transformation specifications including different transformation variants. However, the search for finding the most suitable transformation variant is not based on metaheuristics but delegated to the model engineers who have to make decisions for guiding the search process.

Another very recent line of research concerning the search of transformation results is already applying search-based techniques to orchestrate the transformation rules to find models fulfilling some given objectives. The authors in [DJVV14] propose a strategy for integrating multiple single-solution search techniques directly into a model transformation approach. In particular, they apply exhaustive search, randomized search, Hill Climbing, and Simulated Annealing. Their goal is to explicitly model the search algorithms as graph transformations. Compared to this approach, our approach is going into the opposite direction. We are reusing already existing search algorithms provided by dedicated frameworks from the search-based optimization domain. The most related approach to MOMoT is presented by Abdeen et al. [AVS+14]. They also addresses the problem of finding optimal sequences of rule applications, but they focus on population-based search techniques. Thereby, they consider the multi-objective exploration of graph transformation systems, where they apply NSGA-II [DAPM02] to drive rule-based design space explorations of models. For this purpose, they have directly extended a model transformation engine to provide the necessary search capabilities. Our presented work has the same spirit as the previous mentioned two approaches, however, our aim is to provide a loosely coupled framework which is not targeted to a single optimization algorithm, but allows to (re)use the most appropriate one for a given transformation problem. Additionally, we aim to support the model engineer in using these algorithms through a dedicated configuration DSL and provide analysis capabilities to evaluate the performance of different algorithms. As an interesting line of future research we consider the evaluation of the flexibility and performance of the different approaches we have now for combining

MDE and SBO: modeling the search algorithms as transformations [DJVV14], integrating the search algorithms into transformation engines [AVS+14], or combining transformation engines with search algorithm frameworks as we are doing with MOMoT.

Program Transformation. Program transformation is a field closely related to model transformation [Vis01], thus, similar problems are occurring in both fields. One challenging program transformation scenario is to enhance the readability of source code given certain metrics. In this context, we are aware of a related approach that discusses the search-based transformation of programs [FHH03, FHH04]. In particular, a set of rewriting rules is presented to optimize the readability of the code and dedicated metrics are proposed and used as fitness function. As search techniques random search, Hill Climbing, and genetic algorithms are used. Our approach follows a similar idea of finding optimal sequences of rule applications, but in our case we are focussing on model structures and model transformations instead of source code. However, we consider the instantiation of our framework for the problem of program transformation in combination with model-driven reverse engineering tools [BCJM10] as an interesting subject for future work to further evaluate our approach.

CHAPTER 5

Evaluation

In this chapter, we evaluate our model-level optimization approach called MOMoT, which we introduced in Chapter 4. Specifically, we are interested in the application of our approach and whether we can use it to tackle specific problems from the area of software engineering and MDE. Consequently, the evaluation is performed on several case studies that cover different aspects of the approach and focus on different research questions. Section 5.1 provides an overview of these case studies and the related research questions. The first set of research questions relating to the features of MOMoT are answered in Section 5.2. Afterwards, we develop two new problem solving approaches based on MOMoT to tackle the problem of modularizing transformations in Section 5.3 and the problem of generically modularizing transformation languages in Section 5.4.

5.1 Overview

In order to organize the evaluation of MOMoT, we group case studies that focus on similar research questions together. As such, we have grouped case studies that evaluate different features of MOMoT using existing case studies from literature under the term *reproduction case studies*. Additionally, we introduce two novel, MOMoT-based approaches that tackle the problem case studies of *modularizing model transformations* and the problem of *modularizing modeling languages* in order to demonstrate how the features of MOMoT can be used to formulate a new problem and analyze the respective results. An introduction to the case studies and the tackled research questions is given in the next paragraphs while the case study details can be found in the following sections.

5.1.1 Reproduction Case Studies

The reproduction case studies evaluate the applicability of our approach for existing problems, the overhead MOMoT introduces compared to a native search-based encoding,

and the search features provided by MOMoT. In particular, we use the following four case studies to validate these features.

The *Stack Load Balancing* case study problem domain is a set of stacks that are interconnected in a circular way, where each stack can have a different number of boxes referred to as load The main goal of this case study is to find a good load balance. The *Class Modularization* case study represents a classic problem in software architecture design where the aim is to group classes into modules to create high-quality object-oriented models. This case study has been used as running example in Section 4.2. The *Class Diagram Restructuring* case study is related to the restructuring of class diagrams in order to enhance the object-oriented design while preserving the given behavior. The aim is to remove duplicate properties from the overall class diagram, and to identify new entities which abstract data features shared in a group of entities in order to minimize the number of elements in the class diagram. This case study is based on a case study presented in the Transformation Tool Contest (TTC) of 2013 [GRK13]. The *EMF Refactor* case study is built upon EMF Refactor [Ecl14, AT13], an open-source Eclipse project that supports a structured model quality assurance process for Ecore and UML models. As part of their process, EMF Refactor allows the calculation of different properties on model level. With this case study, we show how existing property calculation can be incorporated into MOMoT.

The research questions tackled by these case studies are:

RQ1 *Applicability*: Is our approach applicable to challenging problems in model-based software engineering?

RQ2 *Overhead*: How much runtime overhead is introduced by our approach compared to a native encoded solution?

RQ3 *Search Features*: What are the additional search features offered by MOMoT w.r.t. pure transformation approaches?

5.1.2 Model Transformation Modularization

This case study tackles the newly formulated problem of modularizing model transformations in order to improve characteristics like maintainability or testability.

Model transformation programs are iteratively refined, restructured, and evolved due to many reasons such as fixing bugs and adapting existing transformation rules to new metamodels version. Thus, modular design is a desirable property for model transformations as it can significantly improve their evolution, comprehensibility, maintainability, reusability, and thus, their overall quality. Although language support for modularization of model transformations is emerging, model transformations are created as monolithic artifacts containing a huge number of rules. To the best of our knowledge, the problem of automatically modularizing model transformation programs was not addressed before in the current literature. These programs written in transformation languages, such

as ATL, are implemented as one main module containing a large number of rules. To tackle this problem and improve the quality and maintainability of model transformation programs, we propose an automated search-based approach to modularize model transformations based on higher-order transformations whose application and execution is guided by MOMoT. We demonstrate the feasibility of our approach by using ATL as concrete transformation language and NSGA-III as optimization method to find a trade-off between different well-known conflicting design metrics for the fitness functions to evaluate the generated modularized solutions. To validate our approach, we apply it to a comprehensive dataset of model transformations, statistically analyze the results of our experiments, and perform a user study to confirm the relevance of the recommended modularization solutions for several maintenance activities based on different scenarios and interviews.

The research questions tackled in this case study are:

RQ1 *Search Validation*: Do we need an intelligent search for the transformation modularization problem?

RQ2 *Search Quality*: How does the proposed many-objective approach based on NSGA-III perform compared to other multi-objective algorithms?

RQ3.1 *Automatic Solution Correctness*: How close are the solutions generated by our approach to solutions a software engineer would develop?

RQ3.2 *Manual Solution Correctness*: How good are the solutions of our approach based on manual inspection?

RQ4.1 *Usability in Bug Fixing*: How useful are modularizations when identifying or fixing bugs in a transformation?

RQ4.2 *Usability in Metamodel Evolution*: How useful are modularizations when adapting transformation rules due to metamodel changes?

5.1.3 Modeling Language Modularization

Based on the results from the previous case study, we explore the *generic* modularization of modeling languages in this case study and problem formulation.

Modularization concepts have been introduced in several modeling languages in order to tackle the problem that real-world models quickly become large monolithic artifacts. Having these concepts at hand allows for structuring models during modeling activities. However, legacy models often lack a proper structure, and thus, still remain monolithic artifacts. In order to tackle this problem, we present in this case study a modularization transformation approach which can be reused for several modeling languages by binding their concrete concepts to the generic concepts offered by the modularization transformation. This provided binding is enough to reuse different modularization strategies

provided by our search-based model transformations through MOMoT. In this case study, we demonstrate the applicability of our modularization approach for Ecore models.

The research questions tackled in this case study are:

RQ1 *Approach Feasibility*: Is the binding between Ecore and the generic modularization metamodel feasible with the proposed approach?

RQ2 *Result Quality*: How good are the results of the modularization task, i.e., the results of applying the generic modularization strategies?

5.2 Reproduction Case Studies

5.2.1 Introduction

As described in Chapter 4, MOMoT is an approach that combines MDE with search-based optimization methods and allows model engineers to formulate search problems using MDE techniques. In this section we present an evaluation of MOMoT based on four different case studies from the area of MDE and software engineering. Section 5.2.2 defines the research questions to be answered in this evaluation. In Section 5.2.3 we introduce the four case studies in detail while in Section 5.2.4 introduces the measures we apply on the case studies to answer the research questions. Finally, Section 5.2.5 presents the results which we use to answer the research questions and Section 5.2.6 elaborates on the threats to validity of our results.

5.2.2 Research Questions

To evaluate our approach, we are interested in answering the following research questions that target the applicability of MOMoT, the runtime overhead it introduces, and the search features it provides.

RQ1 *Applicability*: Is our approach applicable to challenging problems in model-based software engineering?

RQ2 *Overhead*: How much runtime overhead is introduced by our approach compared to a native encoded solution?

RQ3 *Search Features*: What are the additional search features offered by MOMoT w.r.t. pure transformation approaches?

In order to answer the RQ1 and RQ2, we use four case studies that target different problem areas and differ in their level of complexity with regard to the transformation implementation. The answer to RQ3 is given by-argumentation based on the analysis of the default Henshin transformation engine compared to the MOMoT extension. As a result, this discussion is valid for all provided case studies.

5.2.3 Case Study Setup

This section explains the four case studies used to evaluate MOMoT with respect to the RQs. The first case study is known as the stack balancing example, and is used to exemplify a simple problem domain and to show that our approach is not only applicable to software systems, but to any system where MDE is used in its construction process. The remaining case studies represent classical problems of model-based software engineering.

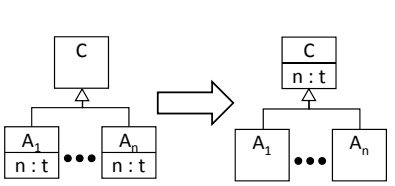
Stack Load Balancing. The problem domain is a set of stacks that are inter-connected in a circular way, where each stack can have a different number of boxes referred to as *load*. The main goal of this study is to find a good load balance. The two objectives that should be minimized are the *standard deviation of the loads* and the *solution length*.

Class Modularization. This is a classic problem in software architecture design that is also used as running example in Section4.2. The goal is to group a number of classes that have inter-dependencies into modules in order to optimize five objectives described previously. Among these objectives, coupling and cohesion are two well-known conflicting objectives. A problem instance of this case study consists of a set of classes and their inter-dependencies, as shown in Figure 4.3 for the *mtunis* system [Hol83]. To manipulate instances of this kind we need to (*i*) create new modules and (*ii*) assign classes to existing modules. A valid solution assigns each class to exactly one module and has no empty modules.

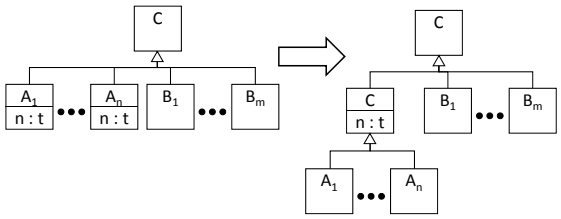
Class Diagram Restructuring. Refactoring is a technique tailored at enhancing object-oriented software designs through the application of behavior-preserving operations [Fow99]. However, there can be a high number of choices and complex dependencies and conflicts between them that makes it difficult to choose an optimal sequence of refactoring steps that would maximize the quality of the resulting design and minimize the cost of the transformation [QH11]. Refactoring can become very complex when we deal with large systems because existing tools offer only limited support for their automated application [MTR07]. Therefore, search-based approaches have been suggested in order to provide automation in discovering appropriate refactoring sequences [SSB06, HJ01, OC08, MKB+14]. Thereby, the idea is to see the design process as a combinatorial optimization problem attempting to derive the best solution, with respect to a given quality measure or objective function, from a given initial design [QH11, OC03].

This case study is based on a case study from the Transformation Tool Contest (TTC) of 2013 [GRK13]. The aim of the TTC series is to compare the expressiveness, the usability, and the performance of graph transformation tools along a number of selected software-engineering case studies. Specifically, we use the class diagram restructuring case study [LKR13, RLP+14], which consists of an in-place refactoring transformation on UML class diagrams realized in the design or maintenance phase of systems construction. A problem instance consists of a set of entities, e.g., classes, and their properties, e.g., attributes or methods, as well as possible inheritance relationships between the entities

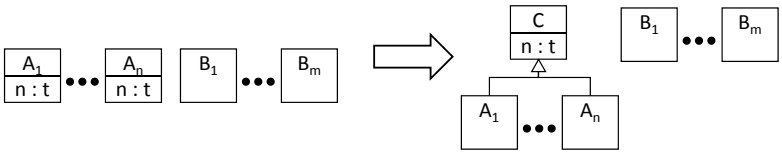
(we only consider single inheritance in this case study). The goal is to remove duplicate properties from the overall class diagram, and to identify new entities which abstract data features shared in a group of entities in order to minimize the number of elements in the class diagram, i.e., entities and properties. The three ways used to achieve this objective are (*a*) *pulling up* common properties of all direct sub-entities into a super-entity, (*b*) *extracting a super-entity* for duplicated properties of entities that already have a super-entity, and (*c*) *creating a root entity* for duplicated properties of entities that have no super-entity. They are graphically shown in Figure 5.1.



(a) *Pull Up Property* Rule



(b) *Extract Super-Entity* Rule



(c) *Create Root Entity* Rule

Figure 5.1: Class Diagram Restructuring Transformations (from [RLP+14])

In this case study, the order in which the three rules are executed has a direct effect on the quality of the resulting model. In fact, considering only the number of elements in the class diagram as objective and the three rules, we can give a canonical solution that always yields the best result. First, properties need to be pulled up (Rule *a*), then super entities should be extracted (Rule *b*), and then new root entities should be created (Rule *c*). If the same rule can be applied on more than one property, it must be applied in the one that has more occurrences first. The way the approaches described in [RLP+14] deal with the rules execution ordering problem is by implementing a rules prioritization mechanism when writing the transformations. In our approach, this can be done using the *Priority Unit*, cf. Table 4.1. Therefore, we can express the class diagram restructuring case study in our approach without any problem.

However, if such a clear ordering of rule applications can be given in advance, a search-based approach is not suitable and the problem should be solved as indicated. In order to make the case study interesting for metaheuristic search, we extend it with an additional domain-specific objective: the *minimization of the maximal depth of the inheritance tree* (DIT) [CK94], i.e., we aim to minimize the longest path from an entity to the root of the hierarchy. The DIT is a metric that can be used to measure the structural complexity of class diagrams [GPC05] and the deeper the hierarchy, the greater the number of

properties an entity is likely to inherit, making it more complex to predict its behavior yielding a higher design complexity. However, the deeper a particular entity is in the hierarchy, the greater the potential reuse of inherited entities [CK94]. Therefore, we have a conflict between the DIT and the objective to minimize the overall number of elements in the class diagram.

EMF Refactor. EMF Refactor is an open-source Eclipse project [Ecl14, AT13] that supports a structured model quality assurance process for Ecore and UML models. In particular, EMF Refactor builds upon the following three concepts:

Metrics Several metrics can be calculated on model-level to gain insight into the model, e.g., average number of parameters of an operation of a class or the depth of the inheritance tree.

Smells Smells are anti-patterns that may be an indicator for insufficient quality in the models. These smells can be generally defined, e.g., the model contains an abstract class without any concrete subclasses, or based on the detected model metrics, e.g., the model contains an operation with more input parameters than the specified limit (*Long Parameter List*).

Refactoring Refactorings are changes made on model-level with the aim of improving the quality of that model. For example, to remove the smell of the *Long Parameter List* we can group a set of parameters together to introduce a separate parameter object which is used instead of the list of parameters.

The aim of this case study is to show how we can integrate the external metrics calculation from EMF Refactor into MOMoT in order to avoid re-definition and re-implementation of these metrics.

5.2.4 Measures

To assess the applicability of MOMoT (*RQ1*), we use all four case studies as they are known problems that have been extensively discussed in the literature. In particular, we consider our approach to be applicable for a specific case study if the respective problem domain can be represented using our approach. This will indicate whether the formalisms used in our approach, i.e., using metamodels and graph transformation rules, are expressive enough for real-world problems. Then, to assess the overhead of our approach (*RQ2*), we compare the time it takes to obtain solutions for a particular problem (total runtime performance) in our approach with the time it takes in a native encoded problem in the MOEA framework. The overhead of our approach will be evaluated for the modularization and the stack case study as representatives of different-sized problems by varying the population size parameter. Finally, in order to demonstrate how our approach advances the current research state w.r.t. search features offered by existing MDE frameworks (RQ3), we compare the search features of the pure Henshin transformation engine with the search features contributed by MOMoT.

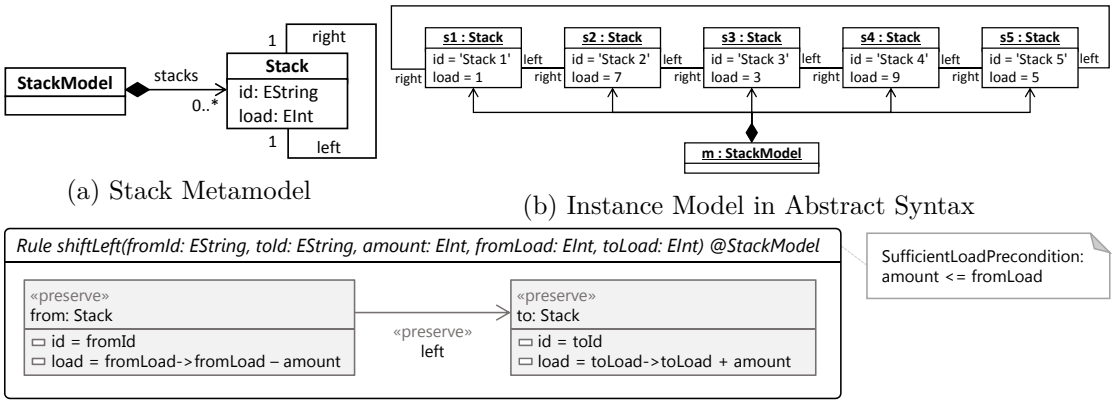
5.2.5 Results

In this section, we describe the experiments conducted with the four case studies and discuss the answers to our research questions based on the obtained results. All results are retrieved through 30 independent runs of the NSGA-III algorithm to deal with the stochastic nature of metaheuristic optimization [HMTY10]. To ensure the sanity of the solutions found during these runs, we manually inspected the solutions for any constraint violations or contradicting objective values. Additionally, we selected one solution for each case study selected using a kneepoint strategy [BBSG11] to evaluate the quality of the solutions. The artifacts created for this evaluation are described in the following sections and are available on our project website [FTW16a].

RQ1: Applicability

To answer the first research question, we have modeled the problem domain of all case studies as Ecore metamodels and have developed rules that manipulate instances of these metamodels.

The **Stack Load Balancing** case study has been modeled as shown in Figure 5.2. The metamodel that represents the system is depicted in Figure 5.2a. Every stack in the system has a unique identifier, a number that indicates its load, and is connected to a left and right neighbor in a circular manner. A concrete instance of this metamodel composed of five stacks with different loads is shown in Figure 5.2b. To manipulate instance models such as the shown one, we propose two basic rules to shift part of the load from one stack either to the left or to the right neighbor. The *ShiftLeft* rule is shown in Figure 5.2c, and an analogous rule is used to shift parts to the right. The rule contains a precondition, which ensures that the amount that is shifted is not higher than the load of the source stack. This attribute condition is shown as an annotation in the figure, although it has been implemented using JavaScript. Applying our approach on the input



(a) Stack Metamodel

(b) Instance Model in Abstract Syntax

(c) *ShiftLeft* Rule to Shift Load

Figure 5.2: Stack System

model shown in Figure 5.2b, we quickly find the shortest sequence of rule applications (three) that leaves five pieces in each stack.

The **Class Modularization** case study has been shown throughout the two previous sections. Thus, we keep the discussion in this section short. Section 4.2 describes how the problem is modeled and represented in Henshin, while Section 4.3 explains how it can be solved with MOMoT, and several code excerpts of the MOMoT configuration file for such example are shown. We have applied our approach to the *mtunis* system instance shown in Figure 4.3. One of the optimal solutions of the Pareto set is presented in Figure 5.3, showing also the respective fitness values. In order to foster the readability of the figure, arrows inside the same module are green while arrows targeting a different module are colored according to their source module.

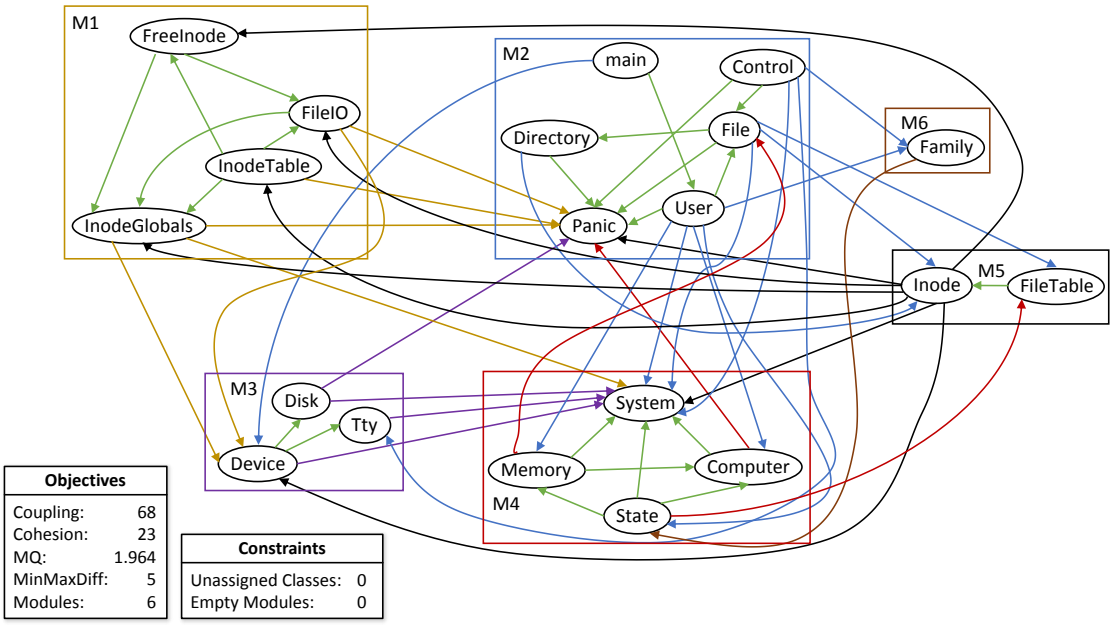


Figure 5.3: Modularization Model Solution for the *mtunis* System

The **Class Diagram Restructuring** case study is the most complex one with regard to rule complexity. Figure 5.4 depicts the metamodel to which problem instances need to conform [RLP+14]. In this metamodel, the inheritance relationship among entities are modeled by using the *Generalization* concept. This way, if an entity has a super type, then the entity has a *generalization* that, in turns, points to a *general* entity. In the same way, if an entity has a child, then it has a *specialization* that, in turns, points to a *specific* entity. Please note that we deal with single inheritance, what is modeled by the cardinality *0..1* of relationship *generalization*. In fact, the problem with multiple inheritance is not considered a search problem and has been solved. For instance, Godin and Mili [GM93] use formal concept analysis and pruned lattices to solve the problem. These techniques are able to reach a unique canonical solution in a polynomial time [BGH+14].

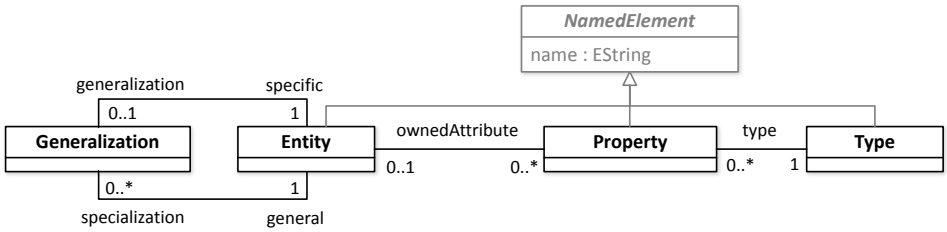


Figure 5.4: Metamodel of the Class Diagram Restructuring Case Study

In order to tackle the class diagram restructuring case study with single inheritance, we translate all three manipulations (cf. Figure 5.1) into graph transformation rules. Each rule needs at least one NAC and contains at least one nested rule. To give an estimate of the complexity to develop such a rule, we depict the rule for creating root classes in Figure 5.5. A nested rule (indicated by a '∗' in the action name in Figure 5.5) is executed as often as possible if the outer rule matches. Therefore nested rules can lead to a large set of overall rule matches, making the application of such rules more expensive.

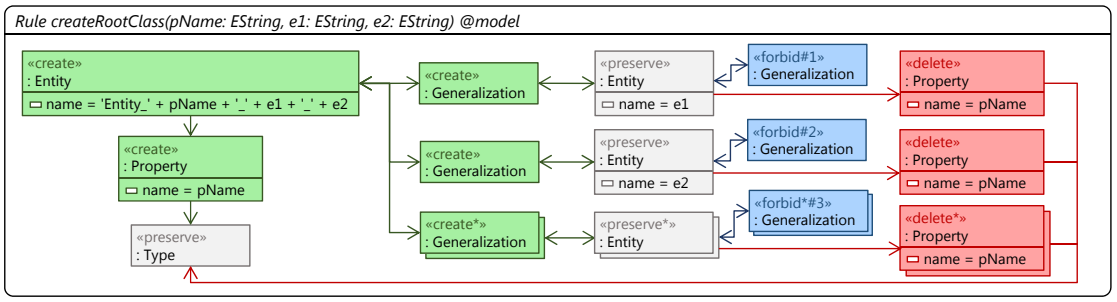


Figure 5.5: Create Root Class Rule With Three NACs and One Nested Rule

The NACs have been implemented both directly as graph patterns (indicated as *forbid* action in Figure 5.5) and as OCL constraints. For example, the NAC of pulling up attributes shown in Listing 5.1 ensures that all sub-classes of the class with the name $eName$ have an attribute with name $pName$ before that attribute is pulled up. Both parameters, $eName$ and $pName$, are matched by the graph transformation engine automatically.

Listing 5.1 OCL-NAC for Pulling Up Attributes of a Class Diagram

```
1: self.entities->select(e | e.name = eName).specialization
2:     ->collect(g | g.specific)
3:     ->forAll(e | e.ownedAttribute->exists(p | p.name = pName))
```

The two objectives for the fitness functions can be translated to OCL or simply calculated in Java. The calculation of the first objective, i.e., the number of elements in the class diagram, is depicted in Listing 5.2. For the calculation of the maximum hierarchy depth,

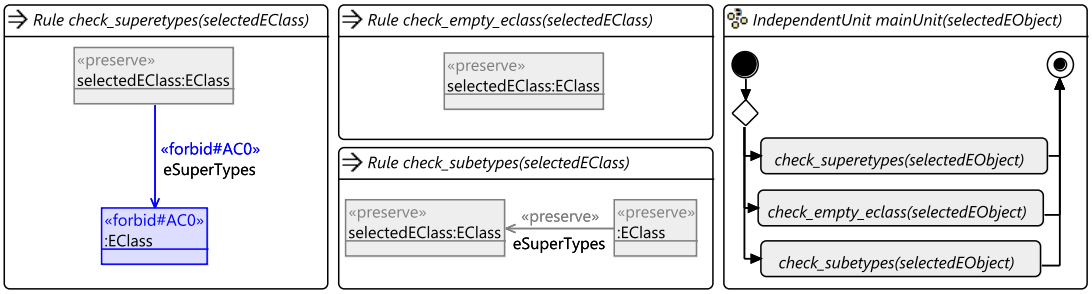
we simply traverse the paths from all leaf entities to its root entity with a derived property called *allSuperEntities* and determine the maximum length as also shown in Listing 5.2.

Listing 5.2 OCL Objectives for the Class Diagram Refactoring Case Study

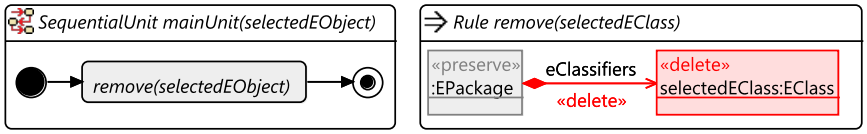
```
1: -- computing the number of elements
2: self.properties->size() + self.entities->size()
3: -- computing the maximum hierarchy depth
4: self.entities -> collect(e|e.allSuperEntities->size())->max()
```

In conclusion, we are able to represent the class diagram restructuring as described in [LKR13, RLP+14]. In addition, we can easily handle the extension with the additional objective which would require a redesign of the other solutions for this case study as presented in [RLP+14].

In the **EMF Refactor** case study, the quality assurance process is partly realized with Henshin to detect metrics by counting how often a given rule can be matched, detecting smells through matching and executing refactorings realized as transformation units. In particular, EMF Refactor uses control-flow units to check whether all pre-conditions of a refactoring are fulfilled before it is actually executed. An example of this is depicted in Figure 5.6, which demonstrates removal of empty subclasses. In the initial check it is evaluated whether the selected class is not the super class of another class (*check_superetypes*), whether it is truly empty (*check_empty_eclass*) and whether it is the sub-class of another class (*check_subetypes*). Only then the actual execution units (cf. Figure 5.6b) are executed which remove the class from the containing package.



(a) Initial Check for *Remove Empty Sub-Class*-Refactoring



(b) Execution for *Remove Empty Sub-Class*-Refactoring

Figure 5.6: *Remove Empty Sub-Class*-Refactoring from EMF Refactor

This case study is the one where we can re-use most existing functionality, since EMF Refactor works on Ecore and UML, two metamodels which already have been defined.

The refactoring rules are also provided, however, since they are split up into an initial check and an actual execution, we have to first combine them into one transformation unit. Here, we make use of the *Conditional Unit* without the *else*-unit. An example for such a combination is depicted in Figure 5.7, which uses the initial check as the *if*-unit and the actual execution as the *then*-unit. Please note that, we renamed the rules in Figure 5.6b as EMF Refactor uses the name *mainUnit* for the combined initial check and the actual execution. Only if the initial check is matched successfully, the actual change will be executed.

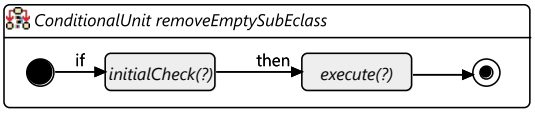


Figure 5.7: Combining Initial Check and Execution of *Remove Empty Sub-Class*

The metrics in EMF Refactor can also be re-used by selecting the elements they can be applied upon from the model (`getDomain`) and calculating the respective metric. Listing 5.3 depicts an example of how we can use the metric calculator that returns the number of all child `EClasses` of a given `EClass` by first selecting all `EClasses` and then calculating the individual values for each class. Re-using the metric calculators from EMF Refactor avoids potential implementation errors and can save a lot of time when working with UML or Ecore models since many metric calculators are already provided.

Listing 5.3 Usage of an EMF Refactor Metric Calculator in MOMoT Objective

```
1: SubClasses : minimize {
2:     val subClassCalculator = new NSUPEC2() // number of all sub-classes
3:     val eClasses = graph.getDomain(EcorePackage.Literals.ECLASS.eClass, true)
4:     var subClasses = 0.0
5:     for(eClass : eClasses) {
6:        subClassCalculator.context = #[ eClass ]   // eClass as list
7:        subClasses += subClassCalculator.calculate // reuse EMF Refactor metric
8:     }
9:     return subClasses
10: }
```

Answering RQ1. Summarizing, we have been able to demonstrate the applicability of our approach in different scenarios with different complexity. In this regard, the rules of the different case studies contain varying features in the different examples, and all of them can be integrated in our approach. For instance, the rules in the stack load balancing case study present a prerequisite that is implemented with JavaScript. The class diagram restructuring case study is modeled with complex rules that use the nesting mechanism provided by Henshin, and have also complex NACs, while the use of Henshin units has been exemplified in the EMF Refactor example. Finally, the modularization problem requires the optimization of five objectives in order to obtain proper solutions, what is feasible with the use of many-objective algorithms [DJ14], such as NSGA-III, also included in MOMoT.

RQ2: Overhead

To evaluate the overhead of our approach, we compare the runtime performance of MOMoT with the performance of a native implementation in the MOEA framework for the modularization and the stack case study. A native implementation only uses classes from the MOEA framework. In order to provide such an implementation the following steps need to be performed:

1. Choose an appropriate representation (*encoding*) for the decision variables and solution.
2. Map the modularization concepts (*classes*, *modules* and their *relationships*) and stack concepts (*stacks*, their *connections*, and *loads*) to that representation, respectively.
3. Provide or select a mechanism to produce new and random solutions.
4. Evaluate the fitness objectives and constraints based on the defined encoding.
5. Decode the representation into a human-understandable form.

The MOEA framework provides binary variables, grammars, permutations, programs, and real variables as well as a set of search operators that can work on these variables.

Class Modularization. For the modularization case study, we use the following strategy. First, all classes are given a unique, consecutive index. Second, we encode the assignments of classes to modules as a sequence of binary variables so that each position in that sequence corresponds to one class and the value of the binary variable indicates the module to which the class is assigned. Therefore we have as many binary variables as there are classes in the model, e.g., for the *mtunis* model (cf. Figure 4.3) a sequence of twenty binary variables would be used. The length of the binary variable is set to represent at most the number of classes in the model, i.e., the case in which all classes are in their own module. Since this is not exactly possible with binary variables, we need to use the next possible value, e.g., to represent twenty modules we need five bits in a variable enabling an actual range from 0 (00000) to 31 (11111). A representation for the described encoding using the *mtunis* model is depicted in Figure 5.8.

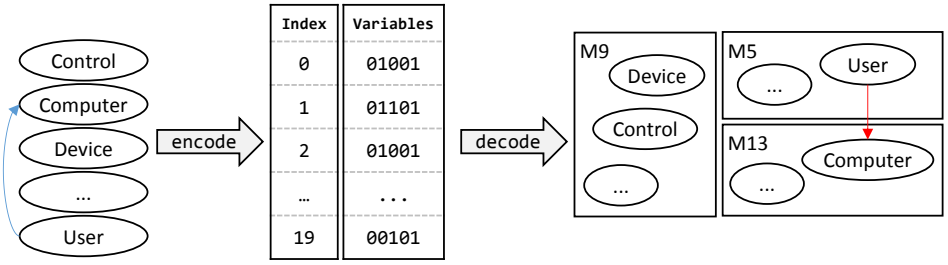


Figure 5.8: Native Encoding for the Modularization Case Study

By choosing binary variables, we can re-use the operators and random solution generators provided by MOEA. However, the evaluation of the fitness of a solution on binary variables is very challenging. We therefore decode the solutions to an internal representation which makes the concepts of classes, modules and dependencies explicit in order to evaluate the objectives and constraints. This representation can then also be used to present the solutions to the user.

To compare the native encoding with the MOMoT encoding, both approaches are run on the same machine with the same algorithm configuration. The machine is a Lenovo T430S with an Intel Core i5-3320M CPU2.60GHz using 8GB RAM and running a 64 bit version of Windows 7 Professional. As algorithm we use the NSGA-III with a maximum of 32768 evaluations, tournament selection with $k = 2$ and the one-point crossover operator with $p = 1.0$. However, due to the differences in the encoding, we cannot use the exact same mutation operators. In the MOMoT approach we mutate parts of the solution sequences by substituting them with applicable, random sequences as explained in Section 4.6. In the native approach, we apply a bit-flip mutation which flips each bit of a binary variable, i.e., switches a 0 to a 1 and vice versa. Both mutation operators are used with $p = 0.1$. To gain insight into the performance, we vary the population size resulting in a different number of iterations, starting with a population size of one and doubling it until we reach the maximum of 32768. Each population size is executed 30 times. A comparison with local search is not possible since MOEA does not provide local search by default.

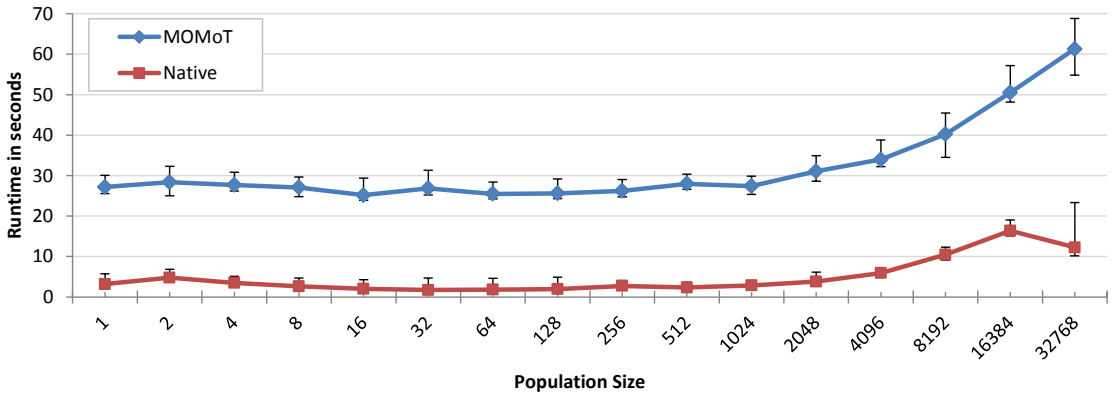


Figure 5.9: Total Runtime Comparison for Modularization: Native Encoding vs MOMoT

The results of the NSGA-III comparison are depicted in Figure 5.9. The solid lines represent the average runtime of all runs while the vertical lines through each point indicate the minimum and the maximum runtime encountered. Table 5.1 depicts the average values and standard deviations for each of the 30 runs. From these experiments we can observe that the native encoding has a stable runtime performance between 1.5 and 20 seconds, while the runtime performance of our approach has a bit more variation, since in each execution it varies between 24 and 69 seconds. Furthermore, our approach performs slower in all execution scenarios.

Table 5.1: Average Runtime and Standard Deviations in Milliseconds for Modularization: Native Encoding vs MOMoT

	MOMoT		Native	
PopulationSize	**Avg**	**StdDev**	**Avg**	**StdDev**
1	27,178	1,141	3,207	833
2	28,394	1,375	4,807	401
4	27,713	846	3,486	317
8	27,082	989	2,658	414
16	25,151	920	2,013	420
32	26,843	1,011	1,753	554
64	25,477	862	1,827	538
128	25,631	968	1,970	570
256	26,249	937	2,734	280
512	27,987	872	2,363	246
1,024	27,396	1,156	2,869	256
2,048	31,065	1,276	3,829	462
4,096	33,967	1,733	5,935	348
8,192	40,233	1,926	10,463	662
16,384	50,472	1,818	16,383	964
32,768	61,282	3,868	12,264	3,220

Stack Load Balancing. In a similar manner, we can encode the stack case study where we need to encode the shifting of load parameters between stacks [FTW15]. The stack system (`StackModel` in Figure 5.2) is represented as a sequence of binary variables. While the integer value of the binary variable indicates how much load is shifted, the position of the binary variable in the sequence indicates from which stack the load is shifted. Therefore we have as many binary variables as there are stacks in the model. In addition, one bit in each binary variable is added as a discriminator for indicating whether the load is shifted to the left or to the right. This bit is not considered to be part of the variables integer value. The number of bits to represent the load value is based on the highest load in the provided stack instance, e.g., to represent 15 we need 4 bits (`1111`) and to represent 16 we need 5 bits (`10000`).

Since in the stack case study we deal with only two objectives, we can use the NSGA-II algorithm for the comparison of MOMoT and a native implementation. To execute the experiment, we fix the problem complexity with 100 stacks having a load between 0 and 200 and stop after 10,000 evaluations. To gain insight into the performance, we vary the population size and the number of iterations respectively. The results of the experiments are depicted in Figure 5.10 and are similar to what we have seen for the modularization case study. In all cases, the performance of our approach is slower than the performance of the opposing native encoding.

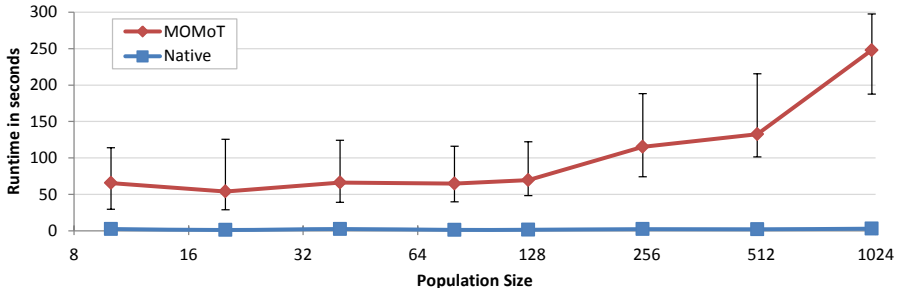


Figure 5.10: Total Runtime Comparison for Stack System: Native Encoding vs MOMoT

This observed loss in performance can be explained two-fold. First, there is a slight difference in the performance of the applied mutation operators. While the bit-flip operator is really fast, the creation of a random transformation sequence is more expensive. Second and more importantly, by using graph transformation rules instead of a native encoding, we inherit the complexity of graph pattern matching and match application, which are expensive tasks on their own. In the worst case, the graph pattern matching problem in Henshin is equivalent to the subgraph isomorphism problem [TKL13] which is NP-complete [Coo71].

To further investigate this behavior we use the JVM Monitor[1] profiler for Java for tracking which operations are the most expensive to execute in MOMoT. The profiling is executed on the same machine as the performance evaluation above and samples the runtime every 50ms. The results obtained from this profiling show that over 90% of the execution time is spent on finding new matches by evaluating all constraints (NACs, PACs, implicit containment and reference constraints as well as explicit attribute constraints) in the graph and then creating the changes that are applied on that graph. These operations are followed by the graph copy operation, the evaluation of the fitness function and the updating of the population set maintained by the algorithm. The graph copy operation is called whenever a new solution is calculated based on the input graph, i.e., the input graph is copied and then the transformation is applied upon that copy resulting in the output graph which is used for the fitness evaluation. The fitness evaluation is called for every created solution and the population set gets updated any time a new solution is added to it, e.g., it needs to check whether the solution is dominated by another solution and how the solution relates to the reference points maintained by NSGA-III.

Answering RQ2. Although Henshin uses constraint solving to tackle the problem of finding pattern matches more efficiently, a loss in performance cannot be avoided. This performance loss is also evident when creating a large initial population in the first iteration, where we have to find and apply a lot of random matches to create solutions, which takes a large proportion of the overall execution time. In fact, when profiling the stack case study, where the performance difference between the native encoding

[1] JVM Monitor, version 3.8.1, available from `http://jvmmonitor.org/`

and MOMoT is even larger, we can see that the larger difference stems from the use of attribute NACs written in JavaScript. The evaluation of these NACs takes a long time as it involves not only the matching process, but also the interpretation of JavaScript statements. Concluding, the performance of MOMoT strongly depends on the complexity of the rules. A similar observation has been made by Arendt and Taentzer [AT13] for EMF Refactor where the time of applying different refactorings can vary from $17ms$ for the *Inline Class* refactoring to $236ms$ for the *Extract Subclass* refactoring.

RQ3: Search Features

In order to evaluate how our approach contributes and integrates search capabilities to existing MDE frameworks, we compare what search features our approach offers with respect to what is currently available in Henshin. This is done by investigating what search capabilities are integrated in Henshin and what problems they can solve. Consequently, we discuss the delta introduced with our approach in order to estimate the effort that is reduced through MOMoT.

In general, with Henshin you need to specify explicitly which transformation unit you want to apply and you need to provide all necessary parameters. The rule engine then tries to find a match for the parameterized transformation unit in the underlying graph. If no matches are found, an error message is displayed. Otherwise, one match is selected either deterministically or non-deterministically, depending on your settings in the rule engine. Multiple units can be applied either manually one after the other or explicitly through a different transformation unit, e.g., the sequential unit. Additionally, Henshin provides the so-called *Henshin State Space Tools* to execute search. These tools are used to verify the correctness of transformations and can generate and analyze a state space given some initial state and the transformation rules. The automatic exploration of Henshin uses a breadth-first search strategy that creates the state space and keeps track of all states that have been visited. Starting from an initial model, all matches for all applicable rules are used to generate a set of new states. Two states are connected through one concrete rule application. In each step of the exploration, a newly created state is checked if it is valid according to a given criteria and if it is a final state (*goal state*) given some other criteria. These criteria can be formulated using OCL or given as a Java class through the configuration. Goal states and states where no transformation rule can be applied are not explored any further (*closed states*). If an invalid state is found, the exploration ends. This is used to evaluate whether the given transformation rules can lead to invalid states by finding counter examples. The automatic explorer stops if all states have been generated and no invalid state has been found. After the exploration process, Henshin provides analysis capabilities to investigate the states space for certain criteria, e.g., finding the shortest path from the initial state to some goal state. Summarized, Henshin provides an exhaustive approach to find paths in an explicitly enumerated state space for specified criteria. More complex behaviour, such as different search heuristics or an optimization procedure where the concrete criteria cannot be specified in advance, needs to be implemented by the user through the API of Henshin.

MOMoT, on the other hand, does not provide an exhaustive approach, but local search and population-based metaheuristic approaches. Therefore, without using MOMoT, one can (*i*) use the Henshin rules manually to derive a solution, (*ii*) implement another approach which is able to apply the rules and check the resulting model or (*iii*) adapt the existing State Space Tools. However, due to the complexity of the problems we are dealing with, i.e., problems that warrant a metaheuristic search, a manual approach is clearly not feasible for larger examples. A search-and-check-approach must be able to take back applied transformations in form of backtracking and in most cases more than one possible solution exists, as also observed by Schätz et al. [SHL10]. Therefore we would at least need to provide rules which are able to negate the effects of another rule. For the modularization case study, this would mean that we need a rule for deleting a class, and another one for removing a feature from a class. For the class diagram restructuring case study, we would need rules to remove root classes and distribute attributes among classes by considering the current class hierarchy in the model. Similar rules need to be implemented for the other case studies. Discarding a manual approach and an approach where we would need to modify the user-provided input, we would aim to adapt the existing functionality in Henshin.

To integrate local search in Henshin, we need to at least implement the following aspects:

- The state space explorer must be able to deal with infinite search spaces as is often created by transformation rules, e.g., the rule which creates modules in the modularization case study. Currently, the explorer would run until it runs out of memory.
- The evaluation of the quality criteria needs to be implemented and should be executed every time a new state is created. Therefore, a concept of *fitness function* needs to be integrated into the existing state space tools. Invalid states (according to the specified constraints) should be marked and the evaluated quality criteria should be saved with each state.
- Currently the tools only derive new states from rules through matching. An extension which allows the use of other transformation units needs to be developed as otherwise the power of control-flow semantics in units cannot be used.
- Additional search strategies besides breadth-first search need to implemented, e.g., depth-first search or A*-search. If possible, the fitness function should guide the direction of the search, similar to what is done in MOMoT.
- In case multiple quality criteria need to be optimized, the search needs a way to compare different solutions, e.g., through the use of the Pareto optimality concept.

In order to implement population-based search, model engineers need to start from scratch. Listing the aspects that need to be considered and implemented in order to provide a similar functionality than MOMoT, it should be clear that this task requires a lot of knowledge and time.

Answering RQ3. Current Henshin search functionality focuses on model-checking aspects and therefore offers an exhaustive search in a bounded, explicitly enumerated search space. However, for the case studies we investigated, such an approach would not yield satisfactory results. First of all it lacks the capabilities to define optimization objectives and second Henshin cannot deal with infinite search spaces. Furthermore, we may only use transformation rules, which means that the expressive power of control-flow units is lost. Therefore a lot of effort needs to be invested in order to solve the given case studies with the default Henshin transformation engine. Consequently, we consider MOMoT a valuable contribution to existing search capabilities with respect to Henshin. In particular, the declarative specification of objectives and constraints and the ability to deal with very large or even infinite transformation search spaces through metaheuristic optimization methods can be considered an advancement.

Summary

In this section, we have demonstrated that our approach is applicable for model-based software engineering problems. However, due to the complexity of graph pattern matching, there is a clear trade-off between the expressiveness provided by MDE through metamodeling and model transformations and the performance of a more dedicated encoding. Inventing such a dedicated encoding is a creative process and is in most cases not as straight forward as in the Modularization case study. Furthermore, once a dedicated encoding has been defined, integrating changes may become quite expensive. The complexity of finding a good, dedicated encoding becomes even more evident when many diverse manipulations with a varying number of parameters of different data types need to be represented in the solution. Additionally, a dedicated encoding may make assumptions about the deployed search algorithm, hampering the switch between different algorithms. Both of these drawbacks are mitigated in our approach, where the parameters and their data types are part of a transformation rule and where the switch to a different algorithm can be done by changing one line in the search configuration. Indeed, this ease of use is especially important for model engineers, who are familiar with MDE technologies such as metamodels, model transformations, and OCL, but may find it challenging to express their problem with a dedicated encoding, corresponding operators and a fitness function as well as converting the obtained solutions back onto model level, where they can be further processed. The use of dedicated encodings is further complicated by the fact that there is often no common consensus how to solve a specific problem. For instance, whereas the works in [SSB06, HT07] both address the problem of refactoring at design level using a modeling approach, each of them proposes a different encoding.

In summary, we conclude that while our approach is applicable for many problems, it introduces an overhead that has its root in the graph matching problem and depends on the complexity of the transformation rules used for solving the respective problem. Nevertheless, our approach enables and guides model engineers to use metaheuristic search to tackle problems which may not be solvable through an exhaustive approach.

5.2.6 Threats to Validity

Although the evaluation of MOMoT features has been carried out with the utmost care, there are several factors that may jeopardize the validity of our results.

Internal validity — Are there factors which might affect the results of this case study? First, a prerequisite of our approach is for the user to be able to create class diagrams (metamodels) and define rule-based systems. While this task was simple for us as model engineers, people from other domains may find these tasks more challenging. Nonetheless, the aim of this approach is to be used in the software engineering process, where models are used as first-class citizens in the design phase, so designers are likely to be knowledgeable about models and model transformations. Second, although we have sanity-checked all solutions, we only had a reference set of optimal solutions for one example (the modularization case study). Further investigation may be needed to ensure that our approach does not introduce problems hindering the algorithm from finding the optimal solutions. In any case, to alleviate this threat, we have implemented native encodings using the MOEA framework for both the stack and modularization case studies. These implementations have yielded similar solutions as the ones obtained with our approach.

External validity — To what extent is it possible to generalize the findings? Even though we have selected four case studies from different areas with varying degrees of complexity, the number of case studies may still not be representative enough to argue that our approach can be applied on any model-based software engineering problem. Therefore, additional case studies need to be conducted to mitigate this threat. Furthermore, we have used Henshin as model transformation language to express in-place model transformations. This means that additional studies are needed in order to know how integrable other model transformation languages are in our approach and to consider out-place model transformations. As part of our future work, we plan to investigate these issues and try to define a minimal set of requirements on the kinds of notations and transformation languages that are amenable to be directly addressed by our approach, cf. Chapter 6.

5.3 Transformation Modularization

In the previous section, we have demonstrated based on four different case studies how MOMoT can be used to tackle existing problems of the MDE and software engineering domain. In this section, we introduce a novel case study to demonstrate the process of formulating a new problem through MOMoT. Specifically, in this case study we investigate the problem of modularizing model transformations.

5.3.1 Motivation

In order to motivate our problem case study, we compare the modularization of software systems with the modularization of model transformation programs.

Modularization of Software Systems. In the last two decades, a large number of research has been proposed to support (semi-)automatic approaches to help software engineers maintain and extend existing systems. In fact, several studies addressed the problem of clustering to find the best decomposition of a system in terms of modules and not improving existing modularizations.

Wiggerts [Wig97] provides the theoretical background for the application of cluster analysis in systems remodularization. He discusses on how to establish similarity criteria between the entities to cluster and provide the summary of possible clustering algorithms to use in system remodularization. Later, Anquetil and Lethbridge [AL99] use cohesion and coupling of modules within a decomposition to evaluate its quality. They tested some of the algorithms proposed by Wiggerts and compared their strengths and weaknesses when applied to system remodularization. Magbool and Babri [MB07] focus on the application of hierarchical clustering in the context of software architecture recovery and modularization. They investigate the measures to use in this domain, categorizing various similarity and distance measures into families according to their characteristics. A more recent work by Shtern and Azerpos [ST09] introduced a formal description template for software clustering algorithms. Based on this template, they proposed a novel method for the selection of a software clustering algorithm for specific needs, as well as a method for software clustering algorithm improvement.

There have also been several developments in search-based approaches to support the automation of software modularization. Mancoridis et al. [MMR+98] presented the first search-based approach to address the problem of software modularization using a single-objective approach. Their idea to identify the modularization of a software system is based on the use of the hill-climbing search heuristic to maximize cohesion and minimize coupling. The same technique has been also used by Mitchell and Mancoridis [MM06, MM08] where the authors present Bunch, a tool supporting automatic system decomposition. Subsystem decomposition is performed by Bunch by partitioning a graph of entities and relations in a given source code. To evaluate the quality of the graph partition, a fitness function is used to find the trade-off between interconnectivity (i.e., dependencies between the modules of two distinct subsystems) and intra-connectivity (i.e., dependencies between the modules of the same subsystem), to find out a satisfactory solution. Harman et al. [HHP02] use a genetic algorithm to improve the subsystem decomposition of a software system. The fitness function to maximize is defined using a combination of quality metrics, e.g., coupling, cohesion, and complexity. Similarly, [SBBP05] treated the remodularization task as a single-objective optimization problem using genetic algorithm. The goal is to develop a methodology for object-oriented systems that, starting from an existing subsystem decomposition, determines a decomposition with better metric values and fewer violations of design principles. Abdeen et al. [ADSA09] proposed a heuristic search-based approach for automatically optimizing (i.e., reducing) the dependencies between packages of a software system using simulated annealing. Their optimization technique is based on moving classes between packages. Mkaouer et al. [MKS+15] proposed to remodularize object oriented software systems using many-objective optimization with

seven objectives based on structural metrics and history of changes at the code level. Praditwong et al. [PHY11] have recently formulated the software clustering problem as a multi-objective optimization problem. Their work aims at maximizing the modularization quality measurement, minimizing the inter-package dependencies, increasing intra-package dependencies, maximizing the number of clusters having similar sizes and minimizing the number of isolated clusters.

Modularization of Model Transformations. In MDE, models and model transformations are considered development artifacts which must be maintained and tested similar to source code in classical software engineering. Similar to any software systems, model transformation programs are iteratively refined, restructured, and evolved due to many reasons such as fixing bugs and adapting existing transformation rules to new metamodels version. Thus, it is critical to maintain a good quality and modularity of implemented model transformation programs to easily evolve them by quickly locating and fixing bugs, flexibility to update existing transformation rules, improving the execution performance, etc. However, while there are several modularization approaches for software engineering to tackle the issue of maintainability and testability, we are not aware of any approach dealing with peculiarities of modularizing rule-based model transformations.

Nevertheless, the introduction of an explicit module concept going beyond rules as modularization concept [KvdBJ07] has been considered in numerous transformation languages to split up transformations into manageable size and scope. In the following, we shortly summarize module support in the imperative transformation language QVT-O [OMG15a], the declarative transformation languages TGGs [KKS07] and QVT-R [OMG15a], and the hybrid transformation languages ATL [JK06, JABK08], ETL [KPP08], and RubyTL [CGM08, CM09] . All these languages allow the import of transformation definitions *statically* by means of explicit keywords. In QVT-O the keyword `extends` is provided, in order to base a new transformation on an existing one. In TGGs, it is possible to *merge* the rule types, i.e., the high-level correspondences from one transformation with those of a new one. In QVT-R it is possible to *import* a dependent transformation file and to *extend* a certain transformation of this file. In ATL it is possible to *use* a set of helper in a transformation or another helper library. ETL allows the *import* of rules from a different transformation definition and so does RubyTL. Going one step further, in [CGdL14] the authors propose transformation components which may be considered as transformation modules providing a more systematic description of their usage context such as required metamodel elements and configurations of a transformation's variability.

Although language support for modularization in model transformation is emerging, it has not been studied in that much detail and has not been widely adopted. This is also reflected by the current application of modularization concepts of transformations within the ATL Transformation Zoo [Ecl], which does not contain any modularized transformation [KSW+13]. Thus, most of the existing ATL transformations are difficult to evolve, test and maintain.

5.3.2 Overview

In this case study, we therefore propose, for the first time in the MDE literature, an automatic approach to modularize large model transformations by splitting them into smaller model transformations that are reassembled when the transformation needs to be executed. Smaller transformations are more manageable in a sense that they can be understood more easily and therefore reduces the complexity of testability and maintainability. In particular, we focus on the modularization of ATL transformations consisting of rules and helper functions. The general usage of ATL has been introduced in Section 2.1.4.

The modularization of model transformation programs is a very subjective process and developers have to deal with different conflicting quality metrics to improve the modularity of the transformation rules. The critical question to answer is what is the best way to regroup the rules that are semantically close by reducing the number of intra-calls between rules in different modules (*coupling*) and increasing the number of inter-calls between rules within the same module (*cohesion*). In such a scenario, it is clear that both of these quality metrics are conflicting. To this end, we leverage the usage of search-based optimization methods to deal with the potentially large search space of modularization solutions. We measure the improvement of both testability and maintainability through common metrics such as coupling and cohesion, which have been adapted for model transformations and which are also used to guide the search process. As a result, our many-objective formulation, based on NSGA-III [DJ14], finds a set of modularization solutions providing a good trade-off between four main conflicting objectives of cohesion, coupling, number of generated modules and the deviations between the size of these modules.

In our evaluation, we demonstrate the necessity for such an approach by outperforming random search in all selected case studies (*sanity check*). Furthermore, we investigate the quality of our generated solutions by determining their recall and precision based on comparison with other algorithms and manual solutions, ensuring quality of the produced results. In this case study, we consider seven different-sized transformations, of which six are available in the ATL Zoo and one has been created within our research group. Specifically, we show the configuration necessary to apply our modularization approach and how the different metrics of the selected transformations can be improved automatically. We found that, on average, the majority of recommended modules for all the ATL programs are considered correct with more than 84% of precision and 86% of recall when compared to manual solutions provided by active developers. The statistical analysis of our experiments over several runs shows that NSGA-III performed significantly better than multi-objective algorithms and random search. We were not able to compare with existing ATL modularization approaches since our study is the first to address this problem. The software developers considered in our experiments confirm the relevance of the recommended modularization solutions for several maintenance activities based on different scenarios and interviews.

Therefore, the contributions of this case study and evaluation over the state of the art can be summarized as follows:

1. **Problem Formulation.** We define the problem of modularizing model transformations as a many-objective optimization problem.
2. **Problem Instantiation.** We instantiate our proposed problem formulation for the use case of ATL, which supports modularization through superimposition, and apply our approach on six different-sized ATL case studies and investigate their results.
3. **Solution Quality.** We demonstrate the quality of our approach by comparing the quality of the automatically generated solutions of NSGA-III with other multi-objective algorithms, one mono-objective algorithm and manually created solutions.
4. **Approach Usability.** The qualitative evaluation of the performed user study confirms the usefulness of the generated modularized solutions based on ATL.

The remainder of this case study is structured as follows. In Section 5.3.3 we introduce modularization capabilities for model transformations. Section 5.3.4 summarizes our generic approach for tackling the model transformation modularization problem using metaheuristic optimization techniques. In Section 5.3.5, the approach is instantiated for the ATL transformation language using the NSGA-III algorithm. Finally, Section 5.3.6 describes the evaluation of our solutions retrieved for the modularization problem.

5.3.3 Modularization in ATL

Domain-Specific Languages (DSLs) are languages that deal with the concepts of the problem domain, cf. Section 2.1.2. These languages tend to support higher-level abstractions than general-purpose modeling languages, and are closer to the problem domain than to the implementation domain. Thus, a DSL follows the domain abstractions and semantics, allowing modelers to perceive themselves as working directly with domain concepts. For transformations, we have transformation DSLs. Among these, we focus in this evaluation on the ATL language, since it has come to prominence in the MDE community. This success is due to ATL's flexibility, support of the main metamodeling standards, usability that relies on good tool integration with the Eclipse world, and a supportive development community [Ecl15]. In general, our proposed approach may be also applicable for other transformation languages providing a module concept.

The Atlas Transformation Language (ATL) is a hybrid model transformation language containing a mixture of declarative and imperative constructs, cf. the class to relational example in Section 2.1.4. Both out-place and in-place transformations can be defined in ATL. An out-place transformation specifies which concepts of the output model are created from which ones of the input model. The *default mode* of ATL is used for this.

In-place rules are defined using the *refining mode* of ATL. In the refining mode, the input model evolves to obtain the output one.

ATL supports three kinds of units: modules, queries, and libraries. An ATL module corresponds to a model to model transformation. This kind of ATL unit enables ATL developers to specify the way to produce a set of target models from a set of source models. Both source and target models of an ATL module must be "typed" by their respective metamodels. Modules are composed of a set of transformation rules, and may also contain helper functions (*helpers*). ATL modules are the only kind of unit that can return output models. ATL queries consist of models to primitive type value transformations. They can be viewed as operations that compute a primitive value from a set of source models. The most common use of ATL queries is the generation of a textual output (encoded into a string value) from a set of source models. ATL queries are not considered in this work due to their rare use [KSW+13]. Finally, ATL libraries enable to define a set of ATL helpers that can be called from different ATL units. Compared to both modules and queries, an ATL library cannot be executed independently.

All ATL units support composition. In particular, ATL libraries can import other libraries, ATL queries can import libraries, and ATL modules can import libraries and other modules. Wagelaar et al. [WSD10] distinguish between external composition, where the output of one transformation serves as input for the next transformation, and internal composition, where one transformation is split into a number of transformation definitions which are combined when the transformation needs to be executed. In that sense, the *superimposition* feature of ATL can be viewed as an internal composition method.

Module superimposition [WSD10] is used to split up transformation modules into modules of manageable size and scope, which are then superimposed on top of each other. This results in (the equivalent of) a transformation module that contains the union of all transformation rules. In addition it is also possible for a transformation module to override rules from the transformation modules it is superimposed upon. Overriding in this sense means replacing the original rule with a new one, whereby the original rule is no longer accessible. When overriding rules the order in which the superimposition is done must be considered. However, in this evaluation, we are not interested in this aspect of superimposition, since our purpose is to split a transformation composed of a large module into modules of smaller size with the aim of improving maintainability, understandability and testability, but we do not modify or override existing rules. On that note, we also consider the creation of ATL libraries. In fact our approach will automatically split a large module into smaller modules and ATL libraries if this improves the aforementioned properties.

5.3.4 Approach

In this case study, we formulate the model transformation modularization problem as a many-objective problem using Pareto optimality. In order to do so, we need to specify three aspects. First, we need to formalize the model transformation domain in which

transformations, both unmodularized and modularized, can be defined in a concise way. This formalization should be independent of any specific transformation language in order to make the approach more widely applicable and generic. Second, we need to provide modularization operations which can be used to convert an unmodularized transformation into a modularized transformation. Each modularization operation serves as decision variables in our solution. And finally, we need to specify a fitness function composed of a set of objective functions in order to evaluate the quality of our solutions and compare solutions among each other. Here, we need to use well-established objectives from the software modularization domain and adapt them for the model transformation domain. An overview of our approach is depicted in Figure 5.11. The basis for this approach is given by MOMoT as described in Chapter 4.

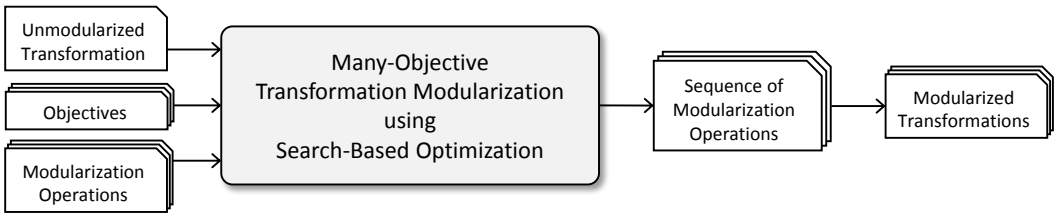


Figure 5.11: Transformation Modularization Approach Overview

Transformation Representation

Our problem domain is the modularization of model transformations. Therefore, we need to be able to represent unmodularized model transformations and modularized model transformations in a concise way. In order to provide a generic approach that is independent of any specific transformation language, we formalize this problem domain using a dedicated *Modularization* domain-specific language (DSL), whose abstract syntax is depicted in the metamodel in Figure 5.12.

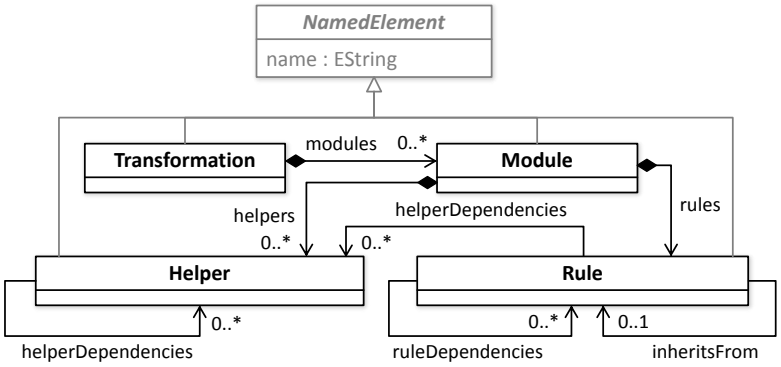


Figure 5.12: Transformation Modularization Metamodel

In our language, a transformation is composed of transformation artifacts. A transformation artifact can be either a transformation rule or an auxiliary function. Following the

syntax of ATL, we have named auxiliary functions in our Modularization DSL *helpers*. In model transformation languages, a transformation rule can inherit the functionality of another rule and may realize its own functionality by implicitly or explicitly invoking other transformation rules or helper functions. A helper function, in turn, provides a piece of executable code which can be called explicitly by any rule or helper. In our DSL, dependencies between rules and helpers are made explicit (`helperDependencies`, `ruleDependencies`) and any hierarchy is flattened. The identification of the three transformation elements, modules, helpers, and rules, is done via a unique name inherited through the class `NamedElement`.

Modularization Operation

For the transformation modularization problem, a solution must be able to convert an unmodularized transformation into a transformation with modules. To represent the process of this conversion, we consider a solution to be a vector of decision variables, where each decision variable in this vector corresponds to one application of a modularization operation. Initially, all rules and helpers of a transformation are contained in one module. The modularization operations assign a rule or a helper function from one existing module to another existing module or a newly created module. Therefore, we only need the two rules depicted in Figure 5.13.

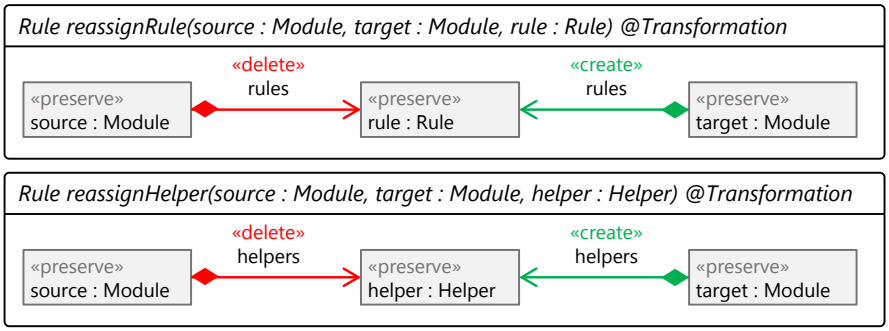


Figure 5.13: Transformation Modularization Rules

The parameters of this operation are the rule or helper that is moved to a module and the respective source and target module. We use an injective matching strategy that specifies that no two left-hand side nodes are assigned to the same model element, i.e., the source and target module parameter in the rules cannot be assigned to the same module element. The bounds for helper and rule parameters are given by the set of rules and helpers in the unmodularized transformation. The bound for the module parameter is a set of modules, where there can be no more than n modules, where n is the total number of rules and helpers, i.e., the case in which all rules and helpers are in their own module. By having such a precise upper bound for the parameters of the modularization operation, we can define the length of the solution vector as n, i.e., a solution where each helper and rule is assigned exactly once.

Solution Fitness

In order to evaluate the quality of the solutions during the search, we consider four objective functions based on the resulting modularized transformation. An overview of these functions is depicted in Table 5.2. Specifically, we aim to minimize the number of modules (NMT), minimize the difference between the lowest and the highest number of transformation artifact, i.e., rules and helpers, in a module (DIF), minimize the coupling ratio (COP) and maximize the cohesion ratio (COH). Since the multi-objective problem formulation only deals with minimization, in practice, we take the negated value of the cohesion ratio.

Table 5.2: Transformation Modularization Objectives

ID	Description	Type
NMT	Number of modules in the transformation	Min
DIF	Min/Max difference in transformation artifacts	Min
COH	Cohesion ratio (intra-module dependencies ratio)	Max
COP	Coupling ratio (inter-module dependencies ratio)	Min

Similar objectives have been used in the class diagram restructuring case study described in Section 4.2. However, for this case study, these objectives have been adapted. Specifically, we do not use coupling and cohesion as absolute number of dependencies, but use the definition of coupling and cohesion ratio as defined by Masoud and Jalili [MJ14]. The concrete formulas for each objective function are given in Equation 5.1 to Equation 5.4. In these formulas, M is the set of all modules and n is the number of all transformations artifacts. $U(m)$, $R(m)$ and $H(m)$ refer to all transformation artifacts, rules, and helpers of a given module m, respectively. Furthermore, $D_{RR}(m_i, m_j)$, $D_{RH}(m_i, m_j)$, and $D_{HH}(m_i, m_j)$ specify the number of rule-to-rule, rule-to-helper and helper-to-helper dependencies between the given modules m_i and m_j, respectively; while $R_R(m_i, m_j)$, $R_H(m_i, m_j)$, and $H_H(m_i, m_j)$ represent the ratio of rule-to-rule, rule-to-helper and helper-to-helper dependencies.

The underlying assumption to minimize the NMT objective is that a small number of modules is easier to comprehend and to maintain. Additionally, distributing the number of rules and helpers equally among the modules mitigates against small isolated clusters and tends to avoid larger modules, as also discussed by [PHY11] in their clustering approach. Furthermore, we optimize the coupling and cohesion ratio which are well-known metrics in clustering problems. Both coupling and cohesion ratios set the coupling, i.e., the number of inter-module dependencies, and the cohesion, i.e., the number of intra-module dependencies, in relation to all possible dependencies between the associated modules. Typically, a low coupling ratio is preferred as this indicates that each module covers separate functionality aspects. On the contrary, the cohesion within one module should be maximized to ensure that it does not contain rules or helpers which are not needed to fulfil its functionality.

$$NMT = |M| \tag{5.1}$$

$$DIF = max(|U(m)|) - min(|U(m)|),\ m \in M \tag{5.2}$$

$$COH = \sum_{m_i \in M} D(m_i, m_i) \tag{5.3}$$

$$COP = \sum_{\substack{m_i, m_j \in M \\ m_i \neq m_j}} D(m_i, m_j) \tag{5.4}$$

$$D(m_i, m_j) = R_R(m_i, m_j) + R_H(m_i, m_j) + H_H(m_i, m_j) \tag{5.5}$$

$$R_R(m_i, m_j) = \frac{D_{RR}(m_i, m_j)}{|R(m_i)| \times |R(m_j) - 1|} \tag{5.6}$$

$$R_H(m_i, m_j) = \frac{D_{RH}(m_i, m_j)}{|R(m_i)| \times |H(m_j)|} \tag{5.7}$$

$$H_H(m_i, m_j) = \frac{D_{HH}(m_i, m_j)}{|H(m_i)| \times |H(m_j) - 1|} \tag{5.8}$$

Finally, to define the validity of our solutions, we enforce through constraints that all transformation artifacts need to be assigned to a module and that each module must contain at least one artifact. Solutions which do not fulfil these constraints are not part of the feasible search space.

5.3.5 Many-Objective Modularization for the case of ATL Programs

To demonstrate our approach, we instantiate it for the ATL transformation language. In particular, we need to perform three steps, which are depicted in the ATL-specific approach in Figure 5.14.

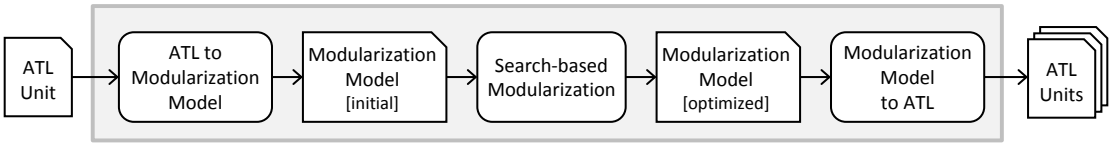


Figure 5.14: ATL Modularization Approach Overview

First, we translate the given ATL transformation into our modularization DSL described in Section 5.3.4. By doing this translation, we make the dependencies within the ATL transformation explicit. Second, we perform the modularization using the modularization operations and fitness function described above. To modularize the transformation expressed in our language we apply our search-based framework called MOMoT with the NSGA-III algorithm. And finally, we translate the optimized modularization model with 1 to n modules back into ATL files, i.e., transformation modules and libraries. All three steps are explained in detail in the following sections.

Converting ATL to Modularization Models

In order to use our generic approach, we need to convert ATL transformations into the modularization language described in Section 5.3.4. While most of this conversion can be done easily as ATL provides explicit concepts for modules, rules, and helpers, the extraction of the dependencies between rules, between helpers and between rules and helpers is more challenging. In fact, in ATL we can distinguish between implicit invocations of *matched rules* and explicit invocation of *lazy rules*, *called rules* and helpers. Lazy and called rules are non-declarative rules that are only executed when they are invoked. Explicit invocation can be visually identified, since it is represented in a similar way as method calls in Java. However, it is much trickier to identify the dependencies among matched rules, i.e., rules that are executed when a respective match on the input model can be found. We have automated the way of producing the dependency model with a high-order transformation (HOT) [TJF+09] that takes the transformation injected into a model-based representation as well as the metamodels of the transformation as input and statically infers information about types in the transformation. The HOT consists of two steps.

First, the types of the rules are statically extracted, i.e., the classes of the metamodels that participate in the rules. While modules and rules already have unique names in ATL, unique names for helpers are derived from the complete helper signature. Second, these types, plus the explicit calls to helpers and non-declarative rules, are used to compute the dependencies. If the type retrieved by an OCL expression of a binding (used to initialize an association) in rule R1 is the same as the type of an `InPatternElement` (element with which the matching is done) in rule R2, then R1 depends on R2 since the object created by the latter rule will be referenced by an object created from the former one. Consequently, the first step consists of extracting such types. Since ATL does not offer any support or API to statically obtain the types of the elements appearing in the rules, the process is not trivial when OCL expressions play part of it. In order to extract the type of an OCL expression, we extract the type of the eventual element that is reached through the navigation, as presented in [BTWV15].

In the second step, after we have extracted the types of the `InPatternElements` of each rule as well as the types appearing in the bindings, we can easily calculate the dependencies. Thus, we consider that a rule, R_1, `depends on` another rule, R_2, if the intersection of the types of the bindings of R_1 with the ones of the `InPatternElements` of R_2 is not empty.

Using this approach, we can obtain a dependencies graph describing the dependencies among rules, between rules and helpers, and among helpers for any ATL model transformation. For the `Class2Relational` example described in Section 2.1.4 and the code excerpt shown in Listing 2.1, we extract that rule `Class2Table` depends on `ClassAttribute2Column` since some of the objects retrieved in the second binding of the former rule, `c.attr -> select(e | not e.multivalued)`, have the same type as the one specified in the `InPatternElement` of the latter rule, i.e.,

`Class!Attribute` where the `multivalued` attribute is set to false. As a result, we get the dependency graph depicted in Figure 5.15.

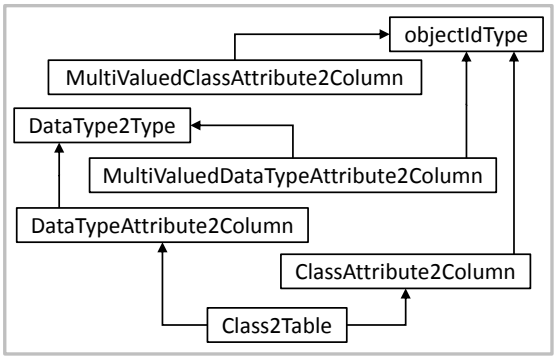


Figure 5.15: Dependencies of the `Class2Relational` Transformation

The model produced by this step conforms to our Modularization DSL and is composed of one module and the same number of rules and helpers as the original ATL transformation. Furthermore, all dependencies between rules and helpers are explicitly declared in the model.

Search-based Modularization

After translating the ATL transformation to our modularization DSL, we can use our search-based framework to find the Pareto-optimal module structure. In order to apply MOMoT for the ATL modularization, we need to specify the necessary input. The instance model is the modularization model obtained in the previous step, while the rules are the modularization operations defined in Section 5.3.4.

Before deciding which artifacts go into which module, we have to create modules. Thereby, we produce input models with different number of modules in the range of $[1, n]$, where n is the number of rules and helpers combined. This covers both the case that all rules and helpers are in one single module and the case in which each helper and rule is in its own module. The objectives and constraints described in Section 5.3.4 are implemented as Java methods to provide the fitness function for MOMoT. Finally, we need to select an algorithm to perform the search and optimization process. For this task, we choose the NSGA-III, as detailed in the next section, since it can handle not only multi-objective problems, but also many-objective problems.

NSGA-III. NSGA-III is a very recent many-objective algorithm proposed by Deb et al. [DJ14]. The basic framework remains similar to the original NSGA-II algorithm with significant changes in its selection mechanism. The pseudo-code of the NSGA-III procedure for a particular generation t is displayed in Algorithm 5.1. First, the parent population P_t (of size N) is randomly initialized in the specified domain, and then the binary tournament selection, crossover and mutation operators are applied to create an

offspring population Q_t. Thereafter, both populations are combined and sorted according to their domination level and the best N members are selected from the combined population to form the parent population for the next generation.

Algorithm 5.1: NSGA-III procedure at generation t

Input: H structured reference points Z_s, parent population P_t.
Output: P_{t+1}

$S_t \leftarrow \emptyset, i \leftarrow 1$
$Q_t \leftarrow$ Variation(P_t)
$R_t \leftarrow P_t \cup Q_t$
$(F_1, F_2, \dots) \leftarrow$ Nondominated_Sort(R_t)
repeat
    $S_t \leftarrow S_t \cup F_i;\ i \leftarrow i + 1$
until $|S_t| \geq N$
$F_l \leftarrow F_i$ // last front to be included
if $|S_t| = N$ **then**
    $P_{t+1} \leftarrow S_t$
else
    $P_{t+1} \leftarrow \bigcup_{j=1}^{l-1} F_j$
    // number of points to be chosen from F_l
    $K \leftarrow N - |P_{t+1}|$
    // normalize objectives and create reference set Z^r
    Normalize(F^M, S_t, Z^r, Z^s)
    // Associate each member s of S_t with a reference point
    // $\pi(s)$: closest reference point
    // $d(s)$: distance between s and $\pi(s)$
    $[\pi(s), d(s)] \leftarrow$ Associate(S_t, Z^r)
    // compute niche count of a reference point $j \in Z^r$
    $p_j \leftarrow \sum_{s \in S_t/F_l}((\pi(s) = j)\ ?\ 1 : 0)$
    // Choose K members one at a time from F_l to construct P_{t+1}
    Niching$(K, p_j, \pi(s), d(s), Z^r, F_l, P_{t+1})$
end

The fundamental difference between NSGA-II and NSGA-III lies in the way the niche preservation operation is performed. Unlike NSGA-II, NSGA-III starts with a set of reference points Z^r. After non-dominated sorting, all acceptable front members and the last front F_l that could not be completely accepted are saved in a set S_t. Members in S_t/F_l are selected right away for the next generation. However, the remaining members are selected from F_l such that a desired diversity is maintained in the population. Original NSGA-II uses the crowding distance measure for selecting well-distributed set of points, however, in NSGA-III the supplied reference points (Z^r) are used to select these remaining members as described in Figure 5.16. To accomplish this, objective values and reference points are first normalized so that they have an identical range. Thereafter, orthogonal

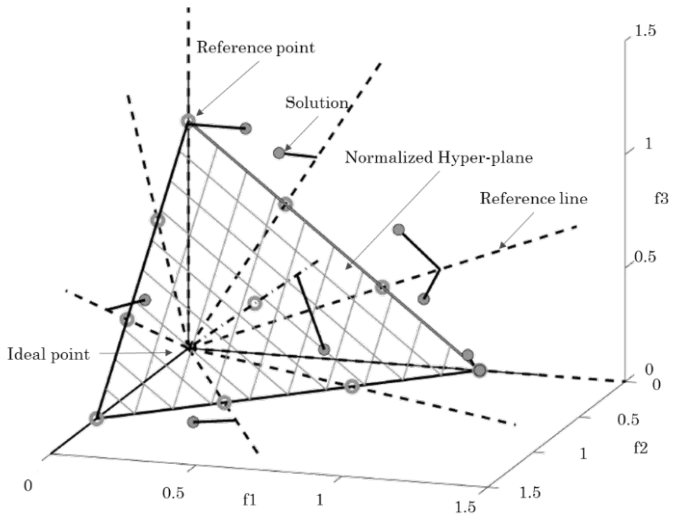


Figure 5.16: Normalized Reference Plane for Three Objectives

distance between a member in S_t and each of the reference lines (joining the ideal point and a reference point) is computed. The member is then associated with the reference point having the smallest orthogonal distance. Next, the niche count p for each reference point, defined as the number of members in S_t/F_l that are associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and the member from the last front F_l that is associated with it is included in the final population. The niche count of the identified reference point is increased by one and the procedure is repeated to fill up population P_{t+1}.

It is worth noting that a reference point may have one or more population members associated with it or need not have any population member associated with it. Let us denote this niche count as p_j for the j-th reference point. We now devise a new niche-preserving operation as follows. First, we identify the reference point set $J_{\min} = \{j : \text{argmin}_j(p_j)\}$ having minimum p_j. In case of multiple such reference points, one ($j^* \in J_{\min}$) is chosen at random. If $p_{j*} = 0$ (meaning that there is no associated P_{t+1} member to the reference point j^*), two scenarios can occur. First, there exists one or more members in front F_l that are already associated with the reference point j^*. In this case, the one having the shortest perpendicular distance from the reference line is added to P_{t+1}. The count p_{j*} is then incremented by one. Second, the front F_l does not have any member associated with the reference point j^*. In this case, the reference point is excluded from further consideration for the current generation. In the event of $p_{j*} \geq 1$ (meaning that already one member associated with the reference point exists), a randomly chosen member, if exists, from front F_l that is associated with the reference point F_l is added to P_{t+1}. If such a member exists, the count p_{j*} is incremented by one. After p_j counts are updated, the procedure is repeated for a total of K times to increase the population size of P_{t+1} to N.

Search Operators. 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. We describe in the following the two main used search operators of crossover and mutation (cf. Section 2.2.2).

Crossover. When two parent individuals are selected, a random cut point is determined to split them into two sub-vectors. The crossover operator selects a random cut-point in the interval [0, *minLength*] where *minLength* is the minimum length between the two parent chromosomes. Then, crossover swaps the sub-vectors from one parent to the other. Thus, each child combines information from both parents. This operator must enforce the maximum length limit constraint by eliminating randomly some modularization operations. For each crossover, two individuals are selected by applying the *SUS* selection. Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creating two offspring $P1'$ and $P2'$ from the two selected parents $P1$ and $P2$. It is defined as follows. A random position k is selected. The first k operations of $P1$ become the first k elements of $P1'$. Similarly, the first k operations of $P2$ become the first k operations of $P2'$. Then, the remaining elements of $P1$ become the remaining elements of $P2'$ and the remaining elements of $P2$ become the remaining elements of $P1'$.

Mutation. The mutation operator consists of randomly changing one or more dimensions (modularization operator) in the solution. Given a selected individual, the mutation operator first randomly selects some positions in the vector representation of the individual. Then, the selected dimensions are replaced by other operation. When applying the mutation and crossover, we used also a repair operator to delete duplicated operations after applying the crossover and mutation operators.

Converting Modularization Models to ATL Files

After retrieving the solutions produced by the search, each module is automatically translated back to an ATL unit, resulting in n ATL files. Modules only containing helper functions are translated into libraries and modules which have at least one rule are translated into normal ATL modules. This translation can be done using the unique naming in our modularization model and the original ATL transformation. The whole transformation is again implemented as a HOT.

In the next section we evaluate our approach to modularize ATL transformations.

5.3.6 Evaluation

In order to evaluate our approach through the ATL instantiation, we answer four research questions regarding the need for such an approach, the correctness of the solutions and the usability of the modularization results. In the next subsections, we describe our research questions and the seven case studies and metrics we use to answer these questions. Finally, we discuss the answer to each research question and overall threats to validity of our approach.

Research Questions

Our study addresses the following four research questions. With these questions, we aim to justify the use of our metaheuristic approach, compare the use of NSGA-III with other algorithms (Random Search, ε-MOEA and SPEA2), argue about the correctness of the modularization results retrieved from our approach and validate the usability of our approach for software engineers in a real-world setting.

RQ1 *Search Validation: Do we need an intelligent search for the transformation modularization problem?* To validate the problem formulation of our modularization approach, we compared our many-objective formulation with Random Search (RS). If Random Search outperforms a guided search method, we can conclude that our problem formulation is not adequate. Since outperforming a random search is not sufficient, the question is related to performance of NSGA-III, and a comparison with other mutli-objective algorithms.

RQ2 *Search Quality: How does the proposed many-objective approach based on NSGA-III perform compared to other multi-objective algorithms?* Our proposal is the first work that tackles the modularization of model transformation programs. Thus, our comparison with the state of the art is limited to other multi-objective algorithms using the same formulation. We select two algorithms, ε-MOEA and SPEA2, to do this comparison. We have also compared the different algorithms when answering to the next questions.

RQ3.1 *Automatic Solution Correctness: How close are the solutions generated by our approach to solutions a software engineer would develop?* To see whether our approach produces sufficiently good results, we compare our generated set of solutions with a set of manually created solutions by developers based on precision and recall.

RQ3.2 *Manual Solution Correctness: How good are the solutions of our approach based on manual inspection?* While comparison with manually created solutions yields some insight into the correctness of our solutions, good solutions which have an unsuspected structure would be ignored. In fact, there is no unique good modularization solution, thus a deviation with the expected manually created solutions could be just another good possibility to modularize the ATL program. Therefore, we perform a user study in order to evaluate the coherence of our generated solutions by manually inspecting them.

The goal of the following two questions is to evaluate the usefulness and the effectiveness of our modularization tool in practice. We conducted a non-subjective evaluation with potential developers who can use our tool related to the relevance of our approach for software engineers:

RQ4.1 Usability in Bug Fixing: How useful are modularizations when identifying or fixing bugs in a transformation? Identifying and fixing bugs in a transformation is a common task in MDE, where transformations are seen as development artifacts. As such, they might be developed incrementally and by many people, leading to potential bugs in the transformation. We investigate whether the performance of this task can be improved through modularization.

RQ4.2 Usability in Metamodel Evolution: How useful are modularizations when adapting transformation rules due to metamodel changes? During the life-cycle of an application, the input and/or output metamodel of a model transformation might change, e.g., due to new releases of the input or output language. When the input or output metamodel changes, the model transformation has to be adapted accordingly to not alter the system semantics. We evaluate whether the adaptation of the transformation rules can be improved through modularization.

In order to answer these research questions we perform experiments to extract several metrics using seven case studies and two user studies. The complete experimental setup is summarized in the next section.

Experimental Setup

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 [Ecl], a repository where developers can upload and describe their ATL transformations.

CS1 Ecore2Maude: This transformation takes an Ecore metamodel as input and generates a Maude specification. Maude [CDE+07] is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications.

CS2 OCL2R2ML: This transformation takes OCL models as input and produces R2ML (REWERSE I1 Markup Language) models as output. Details about this transformation are described in [MGG+06].

CS3 R2ML2RDM: This transformation is part of the sequence of transformations to convert OCL models into SWRL (Semantic Web Rule Language) rules [Mil07]. In this process, the selected transformation takes a R2ML model and obtains an RDM model that represents the abstract syntax for the SWRL language.

CS4 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.

CS5 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.

CS6 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.

CS7 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 5.3 summarizes the number of rules (R), the number of helpers (H), the number of dependencies between rules (D_{RR}), the number of dependencies between rules and helpers (D_{RH}) and the number of dependencies between helpers (D_{HH}) for each case study.

Table 5.3: Size and Structure of All Case Studies

ID	Name	R	H	D_{RR}	D_{RH}	D_{HH}
CS1	Ecore2Maude	40	40	27	202	23
CS2	OCL2R2ML	37	11	54	25	8
CS3	R2ML2RDM	58	31	137	68	17
CS4	XHTML2XML	31	0	59	0	0
CS5	XML2Ant	29	7	28	33	5
CS6	XML2KML	84	5	0	85	2
CS7	XML2MySQL	6	10	5	16	5

Evaluation Metrics. To answer our research questions, we use several metrics depending on the nature of the research question.

Search Performance Metrics: In order to evaluate research questions RQ1 and RQ2, we compare the results of NSGA-III with Random Search, ε-MOEA and SPEA2 based on Hypervolume and Inverted Generational Distance for all case studies.

- Hypervolume (IHV) corresponds to the proportion of the objective space that is dominated by the Pareto front approximation returned by the algorithm and delimited by a reference point. The larger the proportion, the better the algorithm performs. It is interesting to note that this indicator is Pareto dominance compliant

and can capture both the convergence and the diversity of the solutions. Therefore, IHV is a common indicator used when comparing different search-based algorithms.

- Inverse generational distance (IGD) is a convergence measure that corresponds to the average Euclidean distance between the Pareto front approximation produced by the algorithm and the reference front. We can calculate the distance between these two fronts in an M-objective space as the average M-dimensional Euclidean distance between each solution in the approximation and its nearest neighbor in the reference front. Better convergence is indicated by lower values.

Solution Correctness Metrics: In order to evaluate research questions RQ3.1 and RQ3.2, we inspect our solutions with respect to manual solutions and as standalone solutions. Specifically, for RQ3.1, we automatically calculate the precision (PR) and recall (RE) of our generated solutions given a set of manual solutions. Since there are many different manual solutions, only the best precision and recall value is taken into account, as it is sufficient to conform to at least one manual solution. For answering RQ3.2 with the manual validation, we asked groups of potential users to evaluate, manually, whether the suggested solutions are feasible and make sense given the transformation. We therefore define the manual precision (MP) metric.

- To automatically compute precision (PR) and recall (RE), we extract pair-wise the true-positive values (TP), false-positive values (FP) and false-negative values (FN) of each module. TPs are transformation artifacts which are in the same module and should be, FPs are artifacts which are in the same module but should not be and FNs are artifacts which should be together in a module but are not:

$$\begin{aligned} PR &= \frac{TP}{TP+FP} \in [0,1] \\ RE &= \frac{TP}{TP+FN} \in [0,1] \end{aligned}$$

  Higher precision and recall rates correspond to results that are closer to the expected solutions and are therefore desired.

- Manual precision (MP) corresponds to the number of transformation artifacts, i.e., rules and helpers, which are modularized meaningfully, over the total number of transformation artifacts. MP is given by the following equation:

$$MP = \frac{|\text{coherent artifacts}|}{|\text{all artifacts}|} \in [0,1]$$

  A higher manual precision indicates more coherent solutions and therefore solutions that are closer to what a user might expect.

Modularization Usability Metrics: In order to evaluate research questions RQ4.1 and RQ4.2, we consider two dimensions of usability: the estimated difficulty and the time that is needed to perform each task. These tasks are related to bug fixing in the transformations (T1) and adapting the transformations due to metamodel changes (T2).

- Subjects in the usability study are able to rate the difficulty to perform a certain task (DF) using a five-point scale. The values of this scale are *very difficult*, *difficult*, *neutral*, *easy* and *very easy*. The more easy or less difficult in the rating, the better the result.
- In order to get a better estimate about the efficiency a modularized transformation can provide, we ask our study subjects to record the time that is needed to perform each of the tasks. The time unit we use is *minutes* and the less time is needed, the better the result.

Statistical Tests. Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 30 independent simulation runs for each case study and the obtained results are statistically analyzed by using the Mann-Whitney U test [AB11] with a 99% significance level ($\alpha = 0.01$). The Mann-Whitney U test [MW47], equivalent to the Wilcoxon rank-sum test, is a nonparametric test that allows two solution sets to be compared without making the assumption that values are normally distributed. Specifically, we test the null hypothesis (H_0) that two populations have the same median against the alternative hypothesis (H_1) that they have different medians. The p-value of the Mann-Whitney U test corresponds to the probability of rejecting the H_0 while it is true (type I error). A p-value that is less than or equal to α means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than α means the opposite.

For each case study, we apply the Mann-Whitney U test for the results retrieved by the NSGA-III algorithm and the results retrieved by the other algorithms (Random Search, ε-MOEA and SPEA2). This way, we determine whether the performance between NSGA-III and the other algorithms is statistically significant or simply a random result.

User Studies. In order to answer research questions RQ3.1 to RQ4.2, we perform two studies, a correctness study for RQ3.1 and RQ3.2 and a usability study for RQ4.1 and RQ4.2. The correctness study retrieves the precision, recall and manual precision of our generated solutions in order to evaluate how *good* these solutions are. The usability study consists of two tasks that aim to answer the question of usefulness of modularized transformations.

Solution Correctness Study: For RQ3.1, we produce manual solutions to calculate the precision and recall of our automatically generated solutions. These manual solutions are developed by subjects from the University of Michigan which have knowledge of ATL but are not affiliated with this thesis. Our study involved 23 subjects from the University of Michigan. Subjects include 13 undergraduate/master students in Software Engineering, 8 PhD students in Software Engineering, 2 post-docs in Software Engineering. 9 of them are females and 14 are males. All the subjects are volunteers and familiar with MDE and ATL. The experience of these subjects on MDE and modeling ranged from 2 to 16 years. All the subjects have a minimum of 2 years experience in industry (Software companies).

For RQ3.2 we need transformation engineers to evaluate our generated solutions, independent from any solution they would provide. More precisely, we asked the 23 subjects from the University of Michigan to inspect our solutions. For each case study and algorithm, we select one solution using a knee point strategy [BBSG11]. 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 Rachmawati and Srinivasan [RS09] 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. The subjects were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study. 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. Based on the results retrieved through this study, we calculate the manual precision metric as explained in Section 5.3.6.

Modularization Usability Study: In order to answer RQ4.1 and RQ4.2, we perform a user study using two of the seven case studies: Ecore2Maude (CS1) and XHTML2XML (CS4). These two case studies have been selected because they represent a good diversity of case studies as they differ in their size and structure. The Ecore2Maude transformation has a balanced and high number of rules and helpers and quite a high number of dependencies of all kinds. The XHTML2XML transformation, on the other hand, only consists of rules and has a comparatively low number of rule dependencies. In this study, subjects are asked to perform two tasks (T1 and T2) for each case study and version, once for the original, unmodularized transformation and once for the modularized transformation:

T1 Fixing a Transformation: The first task (T1) is related to fixing a model transformation due to bugs that have been introduced throughout the developing cycle. Such bugs usually alter the behavior of a transformation without breaking it, i.e., the transformation still executes but produces a wrong output. To simulate such a scenario, we introduced two bugs into the XHTML2XML transformation and four bugs into the Ecore2Maude transformation since it is larger. The bugs that are introduced in the original and modularized versions of the transformation are of equal nature, e.g., a name change, an addition of values or the change of a rule binding, in order to avoid distorting the results for the comparison. To gain more insight in our evaluation, we split this task into two subtasks: the task of locating the bugs (T1a) and the task of actually fixing the bugs (T1b).

T2 Adapting a Transformation: The second task (T2) we ask our subjects to perform is to adapt a model transformation due to changes introduced in the input or output metamodels. These changes can occur during the lifecycle of a transformation when the metamodels are updated, especially when the metamodels are not maintained by the transformation engineer. In many cases, these changes

break the transformation, i.e., make it not compilable and therefore not executable. To simulate such a scenario, we changed three elements in the XHTML metamodel and two elements in the Maude metamodel. The changes are again equal in nature.

The usability study was performed with software engineers from the Ford Motor Company and students from the University of Michigan. The software engineers were interested to participate in our experiments since they are planning to adapt our modularization prototype for transformation programs implemented for car controllers. Based on our agreement with the Ford Motor Company, only the results for the ATL case studies described previously can be shared in this evaluation. However, the evaluation results of the software engineers from Ford on these ATL programs are discussed in this section. In total, we had 32 subjects that performed the tasks described above including 9 software engineers from the IT department and innovation center at Ford and 23 participants from the University of Michigan (described previously). All the subjects are volunteers and each subject was asked to fill out a questionnaire which contained questions related to background, i.e., their persona, their level of expertise in software engineering, MDE and search-based software engineering. To rate their expertise in different fields, subjects could select from *none*, *very low*, *low*, *normal*, *high* and *very high*. After each task, in order to evaluate the usability of the modularized transformations against the original, unmodularized transformations, subjects also had to fill out the experienced difficulty to perform the task and the time they spent to finish the task.

For our evaluation, we divided the 32 subjects into four equal-sized groups, each group containing eight people. The first group (G1) consists of most of software engineers from Ford, the second and third groups (G2 and G3) are composed of students from the University of Michigan and the fourth group (G4) contains one software engineer from Ford, 2 post-docs and 5 PhD students from the University of Michigan. All subjects have high to very high expertise in software development, model engineering and software modularization and on average a little bit less experience in model transformations and specifically ATL. To avoid the influence of the learning effect, no group was allowed to perform the same task on the same case study for the modularized and unmodularized versions. The actual assignment of groups to tasks and case studies is summarized in Table 5.4.

Table 5.4: Task and Case Study Group Assignment

CS	**Task**	**Original**	**Modularized**
CS1	Task 1	Group 1	Group 3
	Task 2	Group 2	Group 4
CS4	Task 1	Group 3	Group 1
	Task 2	Group 4	Group 2

Parameter Settings. In order to retrieve the results for each case study and algorithm, we need to configure the execution process and the algorithms accordingly. To be precise, all our results are retrieved from 30 independent algorithm executions to mitigate the influence of randomness. In each execution run, a population consists of 100 solutions and the execution finishes after 100 iterations, resulting in a total number of 10,000 fitness evaluations.

To configure all algorithms except Random Search, which creates a new, random population in each iteration, we need to specify the evolutionary operators the algorithms are using. As a selection operator, we use deterministic tournament selection with $n = 2$. Deterministic tournament selection takes n random candidate solutions from the population and selects the best one. The selected solutions are then considered for recombination. As recombination operator, we use the one-point crossover for all algorithms. The one-point crossover operator splits two parent solutions, i.e., orchestrated rule sequences, at a random position and merges them crosswise, resulting in two, new offspring solutions. The underlying assumption here is that traits which make the selected solutions fitter than other solutions will be inherited by the newly created solutions. Finally, we use a mutation operator to introduce slight, random changes into the solution candidates to guide the search into areas of the search space that would not be reachable through recombination alone. Specifically, we use our own mutation operator that exchanges one rule application at a random position with another with a mutation rate of five percent. With these settings, the NSGA-III algorithm is completely configured. However, the ε-MOEA takes an additional parameter called *epsilon* that compares solutions based on ε-dominance [LTDZ02] to provide a wider range of diversity among the solutions in the Pareto front approximation. We set this parameter to 0.2. Furthermore, in SPEA2 we can control how many offspring solutions are generated in each iteration. For our evaluation, we produce 100 solutions in each iteration, i.e., the number of solutions in the population.

As fitness function we use the four objectives described in Section 5.3.4. As a reminder, these objectives are the number of modules in the transformation (NMT), the difference between the number of transformation artifacts, i.e., rules and helpers, in the module with the lowest number of artifacts and the module with the highest number of artifacts (DIF), the cohesion ratio (COH) and the coupling ratio (COP). The initial objective values for each case study are listed in Table 5.5. The arrow next to the objective name indicates the direction of better values.

Result Analysis

Results for RQ1. In order to answer RQ1 and therefore evaluate whether a sophisticated approach is needed to tackle the model transformation problem, we compare the search performance of our approach based on NSGA-III with the performance of Random Search (RS). If RS outperforms our approach, we can conclude that there is no need to use a sophisticated algorithm like NSGA-III. Comparing an approach with RS is a common practice when introducing new search-based problem formulations in

Table 5.5: Initial Objective Values for All Case Studies

ID	Name	NMT ↓	DIF ↓	COH ↑	COP ↓
CS1	Ecore2Maude	1	0	0.15830	0.0
CS2	OCL2R2ML	1	0	0.17469	0.0
CS3	R2ML2RDM	1	0	0.79269	0.0
CS4	XHTML2XML	1	0	0.06344	0.0
CS5	XML2Ant	1	0	0.31609	0.0
CS6	XML2KML	1	0	0.30238	0.0
CS7	XML2MySQL	1	0	0.48888	0.0

order to validate the search effort. Specifically, in our evaluation we investigate the Hypervolume indicator (IHV) and the Inverted Generational Distance indicator (IGD) on 30 independent algorithm runs for all case studies.

The results of our evaluation are depicted in Figure 5.17 using box plots obtained for the metrics of the two algorithms. Each box plot shows the minimum value of the indicator (shown by the lower whisker), the maximum value of the indicator (shown by the upper whisker), the second quantile (lower box), the third quantile (upper box), the median value (horizontal line separating the boxes) and the mean value of the indicator (marked by an 'x'). We can clearly see that for the Hypervolume indicator, RS has lower and therefore worse values than NSGA-III for all case studies. In order to investigate these results, we have deployed the Mann-Whitney U test with a significance level of 99%. As a result, we find a statistical difference between NSGA-III and RS for all case studies, except XHTML2XML. One reason for this result might be that the XHTML2XLM case study has a rather simple structure compared to most of the other case studies. To further investigate the differences between RS and NSGA-III we calculate the *effect size* for both indicators. To be precise, we calculate the effect size using Cohen's d statistic [Coh88], cf. Section 4.7. As a reminder, Cohen's d is defined as the difference between the two mean values $\overline{x_1} - \overline{x_2}$ divided by the mean squared standard deviation calculates by $\sqrt{(s_1^2 + s_2^2)/2}$. The effect size is considered: (i) small if $0.2 \leq d < 0.5$, (ii) medium if $0.5 \leq d < 0.8$, or (iii) large if $d \geq 0.8$. Following this classification, all differences for Hypervolume are considered large.

Interestingly, when we compare RS and NSGA-III for the Inverted Generational Distance (IGD) indicator the same way, the results are different. Please note that for IGD, lower values are considered better, as they indicate an overall better convergence of the algorithm. For IGD, there is no significant difference between the results of NSGA-III and RS, except for the simplest case study, XML2MySQL, where also the effect size yields a large difference. At the same time, in none of the cases the results of RS were significantly better due to the huge number of possible solutions to explore (high diversity of the possible remodularization solutions). Also interesting is the fact that RS produces solutions with a much lower variance of values.

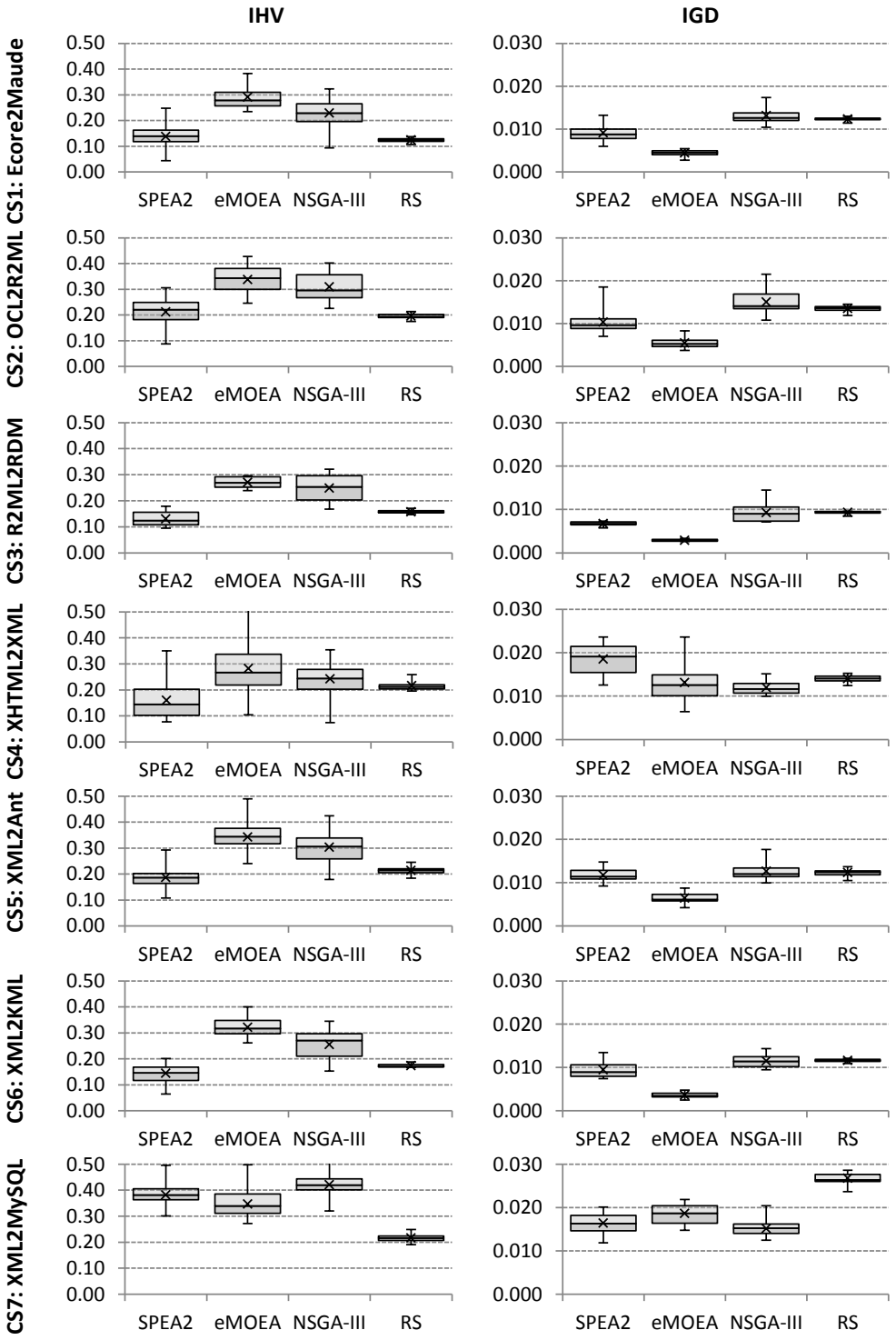


Figure 5.17: Hypervolume (IHV) and Inverted Generational Distance (IGD) Indicator Values Retrieved from 30 Independent Algorithm Runs

While IHV and IGD capture the efficiency of the search, we are also interested in the solutions found by each algorithm. To be more precise, we look at the mean value of each objective value and its standard deviation. The results are depicted in Table 5.6, at the bottom two lines of each case study. The arrow next to the objective name indicates the direction of better values, e.g., we aim to minimize the number of modules in the transformation (NMT). As we can see from the table, in the median case, the results of NSGA-III are better for the number of modules, coupling and cohesion by a factor of around 2 in some cases. The only exception is the difference between the module with the least and most rules and helpers, where RS yields lower values in most case studies. This may be explained through the way NSGA-III tries to balance the optimization of all objective values and by doing so yields good results for all objectives, but may be outperformed when looking only at single objectives.

Table 5.6: Median Objective Values and Standard Deviations Retrieved from 30 Independent Algorithm Runs

CS	Approach	NMT ↓		DIF ↓		COH ↑		COP ↓	
CS1	SPEA2	28	10.09	40	15.08	1.89	1.22	31.14	27.45
	ε-MOEA	21	7.81	45	12.00	2.72	1.42	10.37	13.87
	NSGA-III	23	7.96	44	12.48	3.72	1.72	13.10	11.87
	RS	35	4.26	28	5.28	2.66	1.26	49.66	19.62
CS2	SPEA2	14	4.96	27	8.03	2.68	1.38	3.91	5.15
	ε-MOEA	13	4.51	28	7.45	3.44	1.17	0.84	3.75
	NSGA-III	13	2.94	23	3.77	5.23	1.03	3.09	2.31
	RS	19	3.56	21	4.72	2.56	1.15	5.80	4.58
CS3	SPEA2	30	9.76	49	14.76	1.12	0.87	12.17	12.88
	ε-MOEA	27	8.09	50	12.60	1.28	0.91	2.83	6.20
	NSGA-III	25	3.54	46	6.56	3.22	1.19	7.31	5.84
	RS	39	4.82	32	5.65	1.45	1.01	19.13	12.24
CS4	SPEA2	7	2.76	21	3.85	0.78	0.54	1.35	2.80
	ε-MOEA	7	2.74	20	3.66	0.57	0.36	0.36	2.01
	NSGA-III	7	3.00	18	4.38	1.06	0.66	0.43	2.64
	RS	6	2.31	22	2.97	0.52	0.34	0.31	1.87
CS5	SPEA2	10	3.99	19	6.58	1.55	0.86	4.95	4.30
	ε-MOEA	8	3.36	19	5.48	1.76	1.01	2.06	2.80
	NSGA-III	9	2.18	18	3.89	2.76	0.98	3.05	2.05
	RS	13	3.03	15	3.98	1.53	0.89	6.60	4.67
CS6	SPEA2	23	9.38	57	13.70	0.73	0.69	10.11	8.30
	ε-MOEA	18	7.00	59	10.32	1.50	0.85	7.30	5.88
	NSGA-III	19	3.25	55	6.82	2.08	0.96	11.13	4.63
	RS	30	5.30	47	6.92	1.00	0.82	19.17	6.73

CS7	SPEA2	5	1.78	6	2.84	2.93	1.30	1.50	2.23
	ε-MOEA	4	1.72	7	2.70	3.04	1.16	1.33	1.84
	NSGA-III	4	1.75	6	3.16	3.42	1.48	1.05	2.16
	RS	6	1.73	5	2.61	2.23	1.16	3.17	2.35

In conclusion, we determine that the model transformation problem is complex and warrants the use of a sophisticated search algorithm. Since in none of the cases RS significantly outperforms NSGA-III, while on the other hand there are many instances where NSGA-III dominates RS, we further infer that our many-objective formulation surpasses the performance of random search thus justifying the use of our approach and metaheuristic search.

Results for RQ2. To answer RQ2, we compared NSGA-III with two other algorithms, namely ε-MOEA and SPEA2, using the same quality indicators as in RQ1: Hypervolume (IHV) and the Inverted Generational Distance (IGD). All results are retrieved from 30 independent algorithm runs and are statistically evaluated using the Mann-Whitney U test with a significance level of 99%.

A summary of the results is illustrated in Figure 5.17. As we can see, NSGA-III and ε-MOEA produce better results than SPEA2 for the Hypervolume indicator. In fact, the statistical analysis shows that NSGA-III produces significantly better results than SPEA2 and is on par with ε-MOEA for most case studies. While ε-MOEA has a more efficient search for CS1 and CS6, NSGA-III is the best algorithm for CS7. A slightly reversed picture is shown for the IGD indicator, where ε-MOEA always produces the best results and NSGA-III produces worse results than SPEA2. An exception to that is CS4 where ε-MOEA and NSGA-III are equally good and SPEA2 is worse, and CS5 and CS7 where NSGA-III and SPEA2 produce statistically equivalent results. One possible explanation for this might be that these case studies are small compared to the remaining ones. According to Cohen's d statistic, the magnitude of all differences is large.

Investigating the results further on basis of the retrieved objective values (cf. Table 5.6), we see that NSGA-III and ε-MOEA produce similar median values and standard deviations for most objectives and case studies, closely followed by SPEA2. For NMT, the difference between NSGA-III and ε-MOEA is very small, while for DIF NSGA-III produces better median results for all case studies. The reverse is true for COH and COP where ε-MOEA produces the best results.

In conclusion, we can state that NSGA-III produces good results, but is occasionally outperformed by ε-MOEA. This is interesting as NSGA-III has already been applied successfully for the remodularization of software systems [MKS+15]. However, in the case of software remodularization, the authors used up to seven different objectives in the fitness function which makes the difference of using many-objective algorithms compared to multi-objective algorithms more evident. Therefore, we think that NSGA-III is still a good choice of algorithm for our model transformation problem as it allows the extension

of the number of objectives without the need to switch algorithms. Nevertheless, we also encourage the use of other algorithms. If necessary, only little work is needed to use our approach with a different algorithm [FTW15].

Results for RQ3.1. In order to provide a quantitative evaluation of the correctness of our solutions for RQ3.1, we compare the produced modularizations of NSGA-III, ε-MOEA, SPEA2 and RS with a set of expected modularization solutions. Since no such set existed prior to this work, the expected solutions have been developed by the subjects of our experiments (cf. Section 5.3.6). We had a consensus between all the groups of our experiments when considering the best manual solution for every program. In fact, every participant proposed a possible modularization solution. Then, after rounds of discussions, we selected the best one for every ATL program based on the majority of the votes. Specifically, to quantify the correctness of our solutions, we calculate the precision and recall of our generated solutions as described previously.

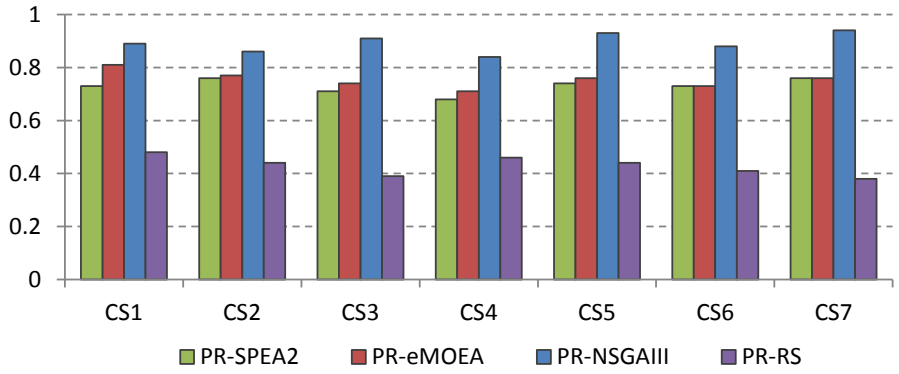


Figure 5.18: Qualitative Correctness Evaluation Using Precision (PR)

Our findings for the average precision (PR) for each algorithm and for all case studies are summarized in Figure 5.18. From these results, we can see that, independent of the case study, NSGA-III has the solutions with the highest precision value, while RS produces solutions that are rather far away from what can be expected. More precisely, our approach based on NSGA-III produces solutions with an average of 89% precision and significantly outperforms the solutions found by the other algorithms. The solutions found by ε-MOEA have an average precision of 75% and the solutions found by SPEA2 have an average precision of 73%. The modularizations produced by RS have the least precision with an average of 43% which cannot be considered good. Based on the average and individual values for all case studies, a ranking of the algorithms would be NSGA-III on the first place, ε-MOEA on second place, SPEA2 on third place, and RS on the last place.

A similar result can be seen for recall (RE) depicted in Figure 5.19, where NSGA-III produces solutions with the highest values, followed by ε-MOEA and SPEA2, and RS produces solutions with the lowest values. Particularly, the average recall of the solutions

found across all case studies by NSGA-III is 90%, for ε-MOEA it is 82%, for SPEA2 it is 72% and for RS it is 48%. The performance of all algorithms is stable independent of the case study size, the highest standard deviations are RS and SPEA2 with 4%. As with precision, the values produced by the sophisticated algorithms can be considered good whereas RS solutions have a too small recall to be considered good. Based on the average and individual values for all case studies, a ranking between the algorithms would look the same as for the precision value.

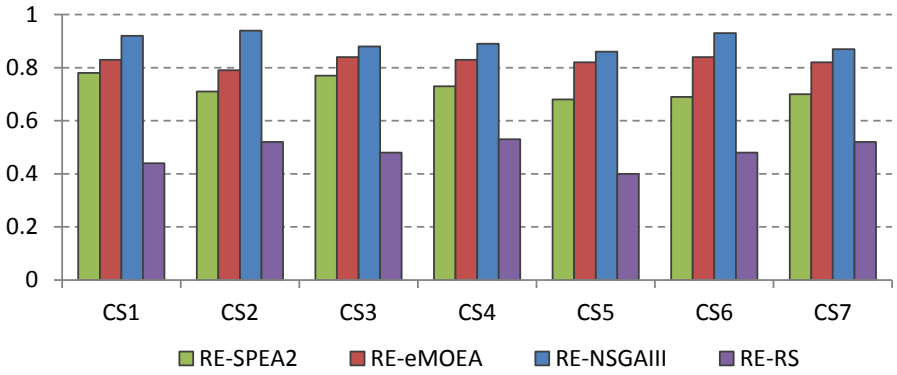


Figure 5.19: Qualitative Correctness Evaluation Using Recall (RE)

Concluding, we state based on our findings that our approach produces good modularization solutions for all cases studies in terms of structural improvements compared to a set of manually developed solutions. In fact, NSGA-III produces the solutions with the highest precision and recall in all case studies compared to the other sophisticated algorithms, ε-MOEA and SPEA2. Furthermore, all sophisticated algorithms significantly outperform RS. It is interesting to note that the quality of the solutions and the ratio among the algorithms are quite stable across all case studies.

Results for RQ3.2. In RQ3.2, we focus more on the qualitative evaluation of the correctness of our solutions by gaining feedback from potential users in an empirical study (cf. Section 5.3.6) as opposed to the more quantitative evaluation in RQ3.1. To effectively collect this feedback, we use the manual precision metric which corresponds to the number of meaningfully modularized transformation artifacts as described previously.

The summary of our findings based on the average MP for all considered algorithms and for all case studies is depicted in Figure 5.20. From these results, we can see that the majority of our suggested solutions can be considered meaningful and semantically coherent. In fact, for NSGA-III, the average manual precision for all case studies is around 96% and for the smaller case studies, i.e., XML2Ant (CS5) and XML2MySQL (CS7), even 100%. This result is significantly higher than that of the other algorithms. To be precise, ε-MOEA yields solutions with an average of 85% MP and SPEA2 has an average of 77% MP over all case studies. On the other hand, the solutions found by RS

only yield solutions with an average of 49% and the worst being 44% for the R2ML2RDM case study (CS3).

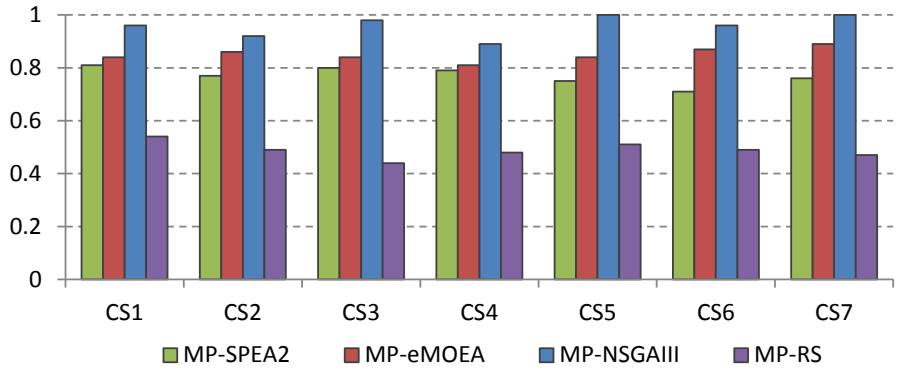


Figure 5.20: Qualitative Correctness Evaluation Using Manual Precision (MP)

In conclusion, we state that our many-objective approach produces meaningfully modularized transformation solutions with respect to the MP metric. While other sophisticated algorithms also yield satisfactory results that can be considered good, our approach based on NSGA-III clearly outperforms these algorithms.

Results for RQ4.1. In order to answer RQ4.1 to evaluate how useful modularizations are when faced with the task of fixing bugs in a transformation, we have performed a user study as described previously. In this study, subjects first needed to locate several bugs in the transformation (T1a) and then fix those bugs by changing the transformation (T1b). Both subtasks were performed for the original and the modularized version of the Ecore2Maude (CS1) and XHTML2XML (CS4) case studies. For the evaluation, we focused on the experienced difficulty and the time that was spent to perform the task.

The results retrieved from the questionnaires for the experienced complexity to perform the task are depicted in Figure 5.21, *CS1-T1a Original* to *CS4-T1b Modularized*. In the figure we see how many of the eight people in each group have rated the experienced difficulty from *very easy* to *very difficult*. As can be seen, the modularized version only received ratings between *very easy* and *neutral*, while the original, unmodularized version received only ratings from *neutral* to *very difficult*. This is true for both subtasks, i.e., locating a bug and actually fixing the bug.

The second dimension we investigate to answer RQ4.1 is the time that is spent to perform the task. To gain this data, subjects were asked to record their time in minutes. The results of this part of the study are depicted in Figure 5.22, *CS1-T1a Original* to *CS4-T1b Modularized*. In the figure, each subtask performed by a group for a specific case study and a specific version is shown as a box plot indicating the minimum and maximum time recorded by each member of the group as well as the respective quartiles. The mean value is marked by an 'x'. As we can see, there is a significant difference between the time needed to perform the tasks on an unmodularized transformation compared to a

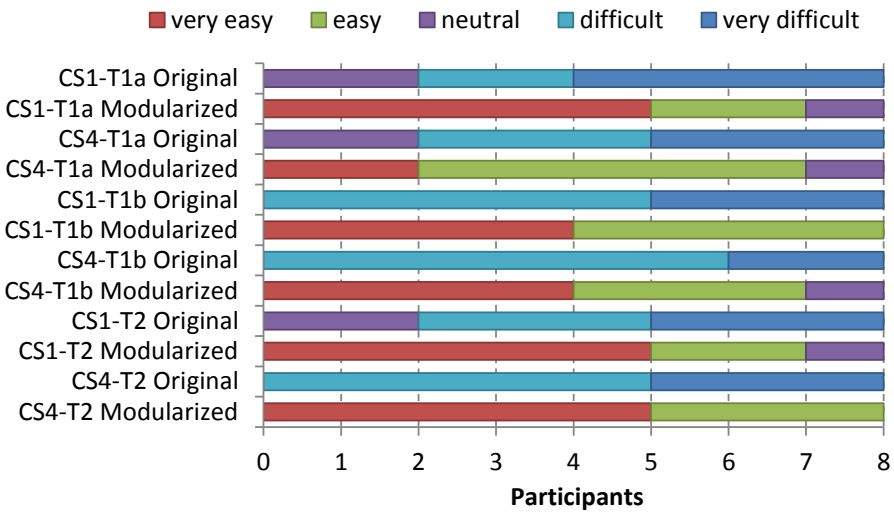


Figure 5.21: Task Difficulty Evaluation for Ecore2Maude (CS1) and XHTML2XML (CS4): Original vs Modularized Transformations

modularized transformation. In fact, the data shows that in all cases, the time needed for the modularized version is around 50% and less of the time needed in the unmodularized version. This seems to be true for both subtasks, even though the distribution within one group may vary.

Concluding, we state that the results clearly show that, independent of the group that performed the evaluation and independent of the respective case study, the task of bug fixing in a model transformation is much easier and faster with a modularized model transformation than with an unmodularized transformation. In this aspect, we think our approach can help model engineers to automate the otherwise complex task of transformation modularization and therefore increase the investigated aspects of the usability when working with model transformations.

Results for RQ4.2. To answer RQ4.2, which is concerned with the adaptability of model transformations due to metamodel changes, we have performed a user study. In this part of the study, subjects were asked to adapt a model transformation after the input or output metamodel has been changed. The necessary changes have been introduced by us, as described previously. As for RQ4.1, the task was performed for the original and the modularized versions of the Ecore2Maude (CS1) and XHTML2XML (CS4) case studies and we focused on the experienced difficulty and the necessary time.

The results retrieved for the experienced complexity are depicted in Figure 5.21, *CS1-T2 Original* to *CS4-T2 Modularized*. Similar to what we have seen for the task of fixing a transformation, there is a significant difference between performing this task for the original, unmodularized transformation and for the modularized transformation. The modularized version received ratings between *very easy* and *neutral* while the original,

unmodularized version received ratings from *neutral* to *very difficult*. Compared to the bug fixing task, the results may suggest that the gain in modularizing transformations when adapting transformations is a bit higher. This difference, however, may be caused by the personal interpretation of a few subjects in one group and cannot be said to be statistically significant.

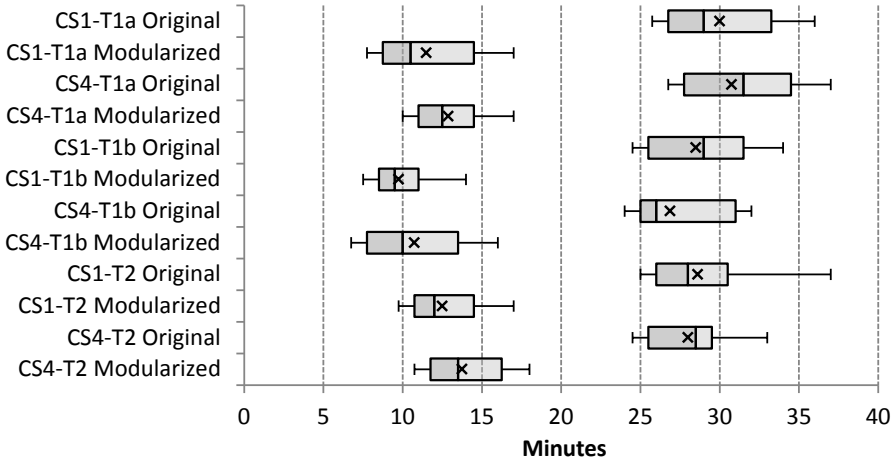


Figure 5.22: Task Time Evaluation for Ecore2Maude (CS1) and XHTML2XML (CS4): Original vs Modularized Transformations

The time the subjects spent on adapting the transformation for each case study and version is depicted in Figure 5.22, *CS1-T2 Original* to *CS4-T2 Modularized*. Here we can see the same trend as with the bug fixing task: a significant reduced time of around 50% and more for the modularized version of the transformation compared to the unmodularized version. Interestingly, we can see that while the time needed to adapt the larger of the two transformations (Ecore2Maude, CS1) is higher than for the smaller transformation as expected, the gain for the larger transformation is also higher, resulting in a reversed result for the two case studies.

In conclusion, we determine that modularizing a transformation has a significant impact on the complexity and time needed to perform model transformation adaptations. Therefore, we think our approach can be useful for model engineers to automate the otherwise complex task of transformation modularization and improve these two metrics with respect to the investigated task.

5.3.7 Threats to Validity

Internal Validity

Are there factors which might affect the results of this evaluation? We consider the following internal threats to the validity of our evaluation based on the executed experiments and the performed user studies. First, we use stochastic algorithms which by their nature produce slightly different results with every algorithm run. To mitigate this

threat, we perform our experiment based on 30 independent runs for each case study and algorithm and analyze the obtained results statistically with the Mann-Whitney U test with a confidence level of 99% ($\alpha = 0.01$). Second, even though the trial-and-error method we used to define the parameters of our search algorithms is one of the most used methods [ES11], other parameter settings might yield different results. Therefore, we need to investigate this internal threat in more detail in our future work. In fact, parameter tuning of search algorithms is still considered an open research challenge. ANOVA-based techniques could be an interesting direction to study the parameter sensitivity. Third, the order in which we placed the objectives might influence the outcome of the search. We plan to further investigate this influence by evaluating different combinations of the objectives in future work. Furthermore, our objectives are limited to static metrics analysis to guide the search process. The use of additional metrics that also capture the runtime behavior of a transformation, e.g., execution time, might yield different results. While it is quite easy to introduce new objectives into our approach, we need to further investigate the use of other metrics in future work, e.g., capturing the performance of a transformation before and after modularization.

Finally, there are four threats to the validity of the results retrieved from the user studies: selection bias, learning effect, experimental fatigue, and diffusion. The selection bias is concerned with the diversity of the subjects in terms of background and experience. We mitigate this threat by giving the subjects clear instructions and written guidelines to assert they are on a similar level of understanding the tasks at hand. Additionally, we took special care to ensure the heterogeneity of our subjects and diversify the subjects in our groups in terms of expertise and gender. Finally, each group of subjects evaluated different parts of the evaluation, e.g., no group has worked on the same task or the same case study twice. This also mitigates the learning effect threat that can occur if the subjects become familiar with a certain case study. The threat of experimental fatigue focuses on how the experiments are executed, e.g., how physical or mentally demanding the experiments are. In particular, we prevent the fatigue threat by providing the subjects enough time to perform the tasks and fill out the questionnaires. All subjects received the instructions per e-mail, were allowed to ask questions, and had two weeks to finish their evaluation. Finally, there is the threat of diffusion which occurs when subjects share their experiences with each other during the course of the experiment and therefore aim to imitate each other's results. In our study, this threat is limited because most of the subjects do not know each other and are located at different places, i.e., university versus company. For the subjects who do know each other or are in the same location, they were instructed not to share any information about their experience before a given date.

External Validity

To what extent is it possible to generalize the findings? The first threat is the limited number of transformations we have evaluated, which externally threatens the generalizability of our results. Our results are based on the seven case studies we have studied and the user studies we have performed with our expert subjects. None of the subjects

were part of the original team that developed the model transformations and to the best of our knowledge no modularized transformations exist for the evaluated case studies. Therefore, we cannot validate the interpretation of the model transformation and what constitutes a good modular structure of our subjects against a "correct" solution by the transformation developers. We cannot assert that our results can be generalized also to other transformations or other experts. Additional experiments are necessary to confirm our results and increase the chance of generalizability.

Second, we focus on the ATL transformation language and its *superimposition* feature, what enables the division of model transformation into modules. However, ATL is not the only rule-based model transformation language. In order for our approach to be generalized also to other model transformation languages, we aim to apply it also to other popular model transformation languages which also provide the notion of modules, such as QVT.

5.4 Modeling Language Modularization

Based on the positive evaluation results retrieved for the modularization of model transformations, we further investigate the modularization capabilities of other MDE concepts. In particular, in this case study we propose an approach to generically modularize modeling languages by formulating it as a search-based problem using MOMoT and providing a generic modularization metamodel.

5.4.1 Motivation

As mentioned in the previous case study, several approaches have been proposed for software modularization considering programming artifacts (cf. Section 5.3.1). In this work, we follow the same search-based spirit, but aim to provide a generic representation of the modularization problem which may be reused for several modeling languages.

Modeling is considered as *the* technique to deal with complex and large systems [BCW12]. Consequently, modularization concepts have been introduced in several modeling languages in order to tackle the problem of real-world models quickly becoming monolithic artifacts, especially when large systems have to be modeled [RM08]. In recent years, different kinds of modularization concepts have been introduced such as modules, aspects [WSK+11], concerns [AKM13], views [ASB09], or subsets [BED14], to name just a few. Having these concepts at hand enables us to structure models during modeling activities. However, legacy models often lack a proper structure as these modularization concepts have not been available or have not been applied, and thus, the models are still monolithic artifacts.

In literature, there are already some approaches which aim in splitting large models into more manageable chunks. First of all, EMF Splitter [GGKdL14] provides means to split large EMF models based on metamodel annotations. Here the user has to come up with a meaningful configuration for the model splitting and the main use case is to deal

with very large models. The same is true concerning Fragmenta [AdLG15] which is a theory on how to fragment models in order to ensure technical constraints. In [SST13], a graph clustering approach has been presented to modularize large metamodels by proposing a specific transformation. In [HJZA07, HHJZ09], an approach is presented how modularity can be added to existing languages. Finally, in [SRTC14] the authors propose an approach for extracting submodels based on textual description.

While these approaches can be used to modularize modeling languages, they do not support a clear definition of quality metrics and in many cases expect the user to come up with a good modularization which is then automatically executed. However, coming up with a good modularization is a difficult task and declaratively specifying the aim of the modularization as quality metrics provides additional documentation and increases comprehensibility. The explicit use of quality metrics has even been mentioned as part of the future work in [SRTC14].

Therefore, in order to tackle the problem of modularizing modeling languages, we present a generic modularization transformation which can be reused for several modeling languages by binding their concrete concepts to the generic concepts offered by our modularization metamodel. This binding is enough to reuse different modularization strategies, mostly based on design quality metrics, provided by our search-based model transformations. Currently, we support the definition of modules which are composed of different entities. The applicability of this modularization approach is demonstrated for Ecore-based languages. Consequently, the contribution of this case study is threefold: (*i*) we provide generic modularization support for modeling languages having at least some kind of basic modularization support; (*ii*) we combine query-driven model transformations and generic model transformations to provide generic means to deal with heterogeneities between the generic modularization metamodel and the specific metamodels; (*iii*) we provide an application of the generic modularization support for real-world Ecore-based languages.

5.4.2 Background

In this section, we present the foundations on which we build our modularization approach. First, we explain how modularization is supported in modeling languages, which kind of transformations may be combined to implement a reusable generic model transformation, and how to instantiate such a transformation for performing a concrete modularization.

Modularization: The Basics

The basic modularization problem, also often referred to as software clustering, considers as input a potentially huge set of elements having certain relationships among them [PHY11, MKS+15]. The goal for the modularization task is to find meaningful modules or clusters for these elements which consider certain quality characteristics. Traditionally, modularization approaches, or re-modularization approaches if some initial module structure is already given which should be improved, consider code-based soft-

ware engineering artifacts. For instance, classes are modularized into modules based on their dependencies such as method invocations or field access. The considered quality characteristics are mostly based on object-oriented design metrics.

Similarly, the question arises how to provide automated modularization support for modeling languages. For instance, languages such as UML and Ecore provide the package concept to structure classifiers, and other languages such as SDL and BPMN provide some means for modularization of models as well. While the languages themselves seem heterogeneous, i.e., generic modeling language vs. domain-specific language, structural vs. behavioural, and so on, the modularization concepts in its basic form of having modules containing entities seems common. Thus, we aim to explore how we may provide a generic modularization transformation based on the combination of three different transformation types.

Generic Model Transformation

Currently there is little support for reusing transformations in different contexts since they are tightly coupled to the metamodels on which they are defined. Hence reusing them for other metamodels becomes challenging. Inspired from generic programming, generic model-to-model transformations [CGdL11] have been proposed, which are defined over so-called metamodel concepts, which are later bound to specific metamodels. Thus, with the help of generic model transformations, we are able to define a generic modularization metamodel which can be bound to different specific modeling languages. Please note that a similar approach has been proposed as role-based model transformations for in-place transformations such as model refactorings [RSA10].

Query Structured Transformation

One major challenge for generic model transformations is to deal with the heterogeneities [WKR+11] between the generic and the specific metamodels, i.e., one concept is defined in the generic metamodel as a class, but the same concept is represented in the specific metamodel as a pattern of different classes having specific relationships. We propose to combine the generic model transformation approach with another emerging transformation approach called query structured transformation [GDM14]. The main idea behind the latter is to enhance the source metamodel with concepts of the target metamodel in a query-driven manner. Using this idea, the source and the target metamodel are adjusted and the mapping between the metamodels is reduced to one-to-one correspondences. As we see in Section 5.4.4, this approach allows us to easily use one-to-one correspondences for binding the generic modularization metamodel to the specific metamodels.

Search-based Model Transformations

Having the mechanisms to define modularization as a generic transformation and using query structured transformations to enhance the binding mechanisms, we still need to

implement the generic transformation actually performing the modularization. In code-based modularization problems, the usage of search-based algorithms, such as genetic algorithm, simulated annealing, and so on, has been proposed to actually compute the optimal module structure for a given problem [PHY11, MKS+15]. The success of search-based algorithms is mostly based on the metaheuristic search capabilities, i.e., finding a good solution without enumerating the whole search space. Following the same spirit, we implement the transformation rules needed for the generic modularization transformation and define the objectives of the modularization using MOMoT. This separation of the transformation rules and the transformation objectives also allows customization of the generic modularization transformation for specific modeling languages more easily.

5.4.3 A Generic Modularization Transformation

In his section, we introduce our concept metamodel for modularization, our modularization chain, and outline several strategies for performing modularization based on the information provided by the concept metamodel.

Generic Modularization Metamodel

Our generic modularization metamodel is presented in Figure 5.23. Abstract classes and relationships are depicted in grey. Elements of type *Language* represent concrete instances of modeling languages (MLs). The concepts of a ML are therefore simplified to *Modules*, *Entities* and *Relationships*. We can see that a language is composed of modules, which represent the clusters that group entities with similarities. Such similarities can come in different ways. For instance, we can consider the similarities between the names of the entities, or the relationships among the entities. Furthermore, we have defined weights for the relationships, since some of them may be more important that others (cf. Section 5.4.4). We can see that an entity can have several relationships with other entities. Each relationship ends in a specific entity.

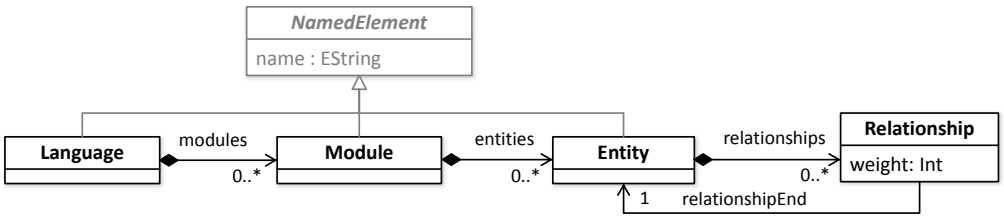


Figure 5.23: Generic Modularization Metamodel

The idea is to express any modeling language in terms of our generic modularization metamodel. This means that the concepts appearing in the MLs are mapped to the three concepts described: modules, entities and relationships.

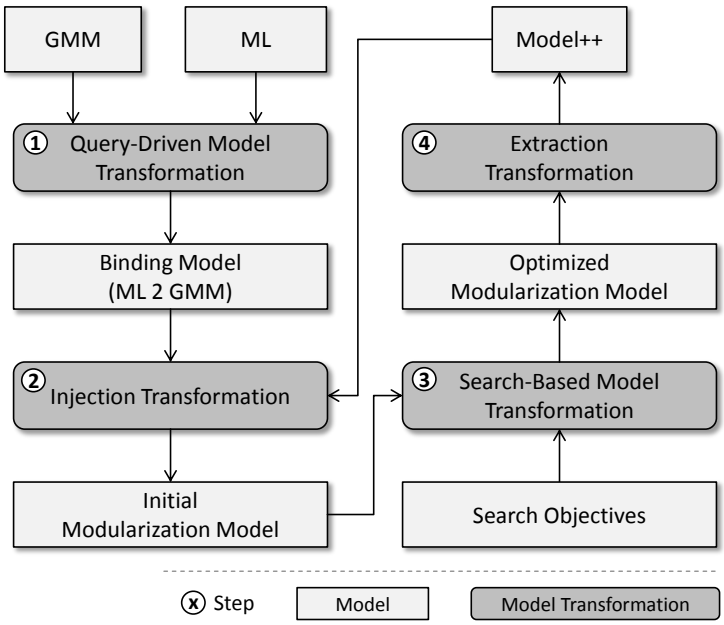


Figure 5.24: Generic Modularization Chain

Modularization Transformation Chain

The overall transformation chain for the generic modularization of modeling languages is shown in Figure 5.24. Steps 1 and 2 are explained in this section, while Step 3 is explained in Section 5.4.3. These three steps are exemplified with an application study in Section 4. Finally, Step 4 is left for future work.

Our approach takes as input a modeling language (ML) and the generic modularization metamodel (GMM). A ML is defined in terms of a domain-specific language (DSL). The structure of a DSL is expressed with a metamodel, which defines the concepts of the language and the relationships among them. The first step of our approach is implemented with a *Query Structured Model Transformation*, whose purpose is to make it easier and more generic the weaving of different DSLs. In our case, we want to translate a ML to our generic modularization metamodel (cf. Figure 5.23).

Therefore, we seek the homomorphism between the metamodel of the ML and our generic modularization metamodel, i.e., the bindings between these two. This homomorphism, also called mapping, has to be manually identified by the software engineer, since she/he has to decide what is a module, an entity and a relationship in the ML. Such mapping can be defined by simply annotating the metamodel of the ML or by adding new derived properties and classes to it that represent the mapping. The outcome of this step is a *Model with Bindings between ML and Generic Language*, "binding model" for short from here on. Thus, we obtain a model where specific concepts of the input ML are virtually connected to specific concepts of our generic modularization language.

The second step is to apply a *Generic Model Transformation* to such model. This transformation takes the binding model as input and generates the *Initial Generic Modularization Model*. In order to do so, it examines the bindings specified in the binding model. The generated model conforms to our generic modularization metamodel and is composed of only one module that contains all the entities. The entities, in turn, have relationships among them. The generic model transformation produces these relationships according to the information specified in the binding model.

In our proof-of-concept implementation [FTW16d], we have implemented the both steps with an ATL transformation for the Ecore case study.

Modularization Strategies

The third step in our approach has to do with the optimal grouping of entities into modules. In order to do so, we apply search-based techniques using our MOMoT approach, cf. Chapter 4. Therefore, we need to specify as input the *Search Objectives* that we want to optimize in our modularization. According to the fitness function defined by such objectives, our *Search-Based Model Transformation* decides the optimal modularization, i.e., how to optimally split the entities contained by the only module in the initial generic modularization model into different modules.

In this evaluation, we define four objectives: (*i*) coupling (COP), (*ii*) cohesion (COH), (*iii*) the difference between the maximum and minimum number of entities in a module (DIF) and (*iv*) number of modules (MOD). These objectives have been described for the modularization of model transformations in Section 5.3 and are in this case study adapted for the modularization of modeling languages. Listing 5.4 illustrates how these objectives are defined as fitness function in MOMoT. Please note that the actual calculation of most objective metrics has been outsourced to the `MetricsCalculator` class, as shown in the `preprocess` method of the fitness function.

Listing 5.4 Fitness Function For Language Modularization

```
1: fitness = {
2:    preprocess = {
3:       // use attribute for external calculation
4:       val root = MomotUtil.getRoot(solution.execute, typeof(Language))
5:       solution.setAttribute("metrics", MetricsCalculator.calculate(root))
6:    }
7:    objectives = {
8:       Coupling : minimize { // java-like syntax
9:          val metrics = solution.getAttribute("metrics", typeof(LanguageMetrics))
10:         metrics.coupling
11:      }
12:      Cohesion : maximize {
13:         val metrics = solution.getAttribute("metrics", typeof(LanguageMetrics))
14:         metrics.cohesion
15:      }
16:      NrModules : maximize {
17:         (root as Language).^modules.filter[ m | !m.entities.empty].size
18:      }
19:      MinMaxDiff : minimize {
20:         val sizes = (root as Language).^modules.filter[m | !m.entities.empty]
```

```
21:             .map[m | m.entities.size]
22:         sizes.max - sizes.min
23:       }
24:     }
25: }
```

The way our search approach works is the following. Given the initial generic modularization model and a set of search objectives defined in our configuration language, our approach introduces a set of empty modules in the language. The number of empty modules that is initially introduced varies between a given range to investigate different areas of the search space. In order to evolve the initial generic modularization model, we only need to define one very simple Henshin rule (cf. Figure 5.25). This rule moves an entity from one module to another. Our tool instantiates the input parameters of the rule with specific entities and modules names. According to the fitness function conformed by the objectives defined, the search engine searches for the optimal assignments of entities into modules. The output of the search is given as (*i*) the *Optimized Modularization Model*, as well as (*ii*) the sequence of rule applications and their input parameters.

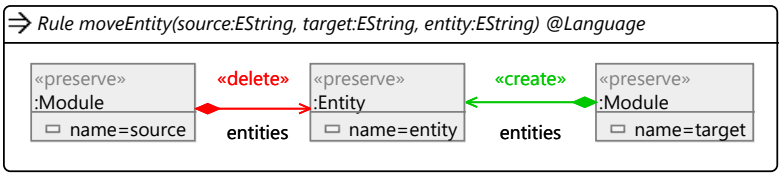


Figure 5.25: *MoveEntity* Rule

Having our generic metamodel, our modularization transformation chain, the necessary objectives, and the generic model transformation, we can apply our approach on a concrete examples.

5.4.4 Application Study: Ecore

In this section we apply our generic modularization chain presented in the previous section for modularizing Ecore-based languages. In order to address this, we (*i*) define the research questions for our study, (*ii*) explain how Ecore models are translated to our modularization language (cf. Figure 5.23), and finally (*iii*) describe the results obtained by our search-based approach for real-world Ecore models.

Research Questions

In particular, we are interested in answering the following research questions (RQs).

RQ1 *Feasibility*: Is the binding between Ecore and the generic modularization metamodel feasible with the proposed approach?

RQ2 *Result Quality*: How good are the results of the modularization task, i.e., the results of applying the generic modularization strategies?

Binding between Ecore and the Generic Modularization Metamodel

The first step is to conceptually bind the concepts of the Ecore language with those of our generic modularization language by means of a query-driven model transformation. A simplified version of the Ecore metamodel with the concepts that we take into account is presented in Figure 5.26. Please note that unmapped Ecore elements are depicted with grey color. We can see that in Ecore `EPackages` contain `EClasses` and `EDataTypes`, whereas classes can inherit structure and functionality from other classes (`eSuperTypes` relationship). At the same time, classes contain `EReferences` and `EAttributes`. The former are used to specify general relationships among classes. Consequently, a reference needs to have an `eReferenceType` that points to the respective class. Furthermore, a reference can either represent a `containment` or not. Attributes (`EAttributes`), on the other hand, have a specific type (`eAttributeType`), which is specified by a datatype or an enumeration (`EEnum`).

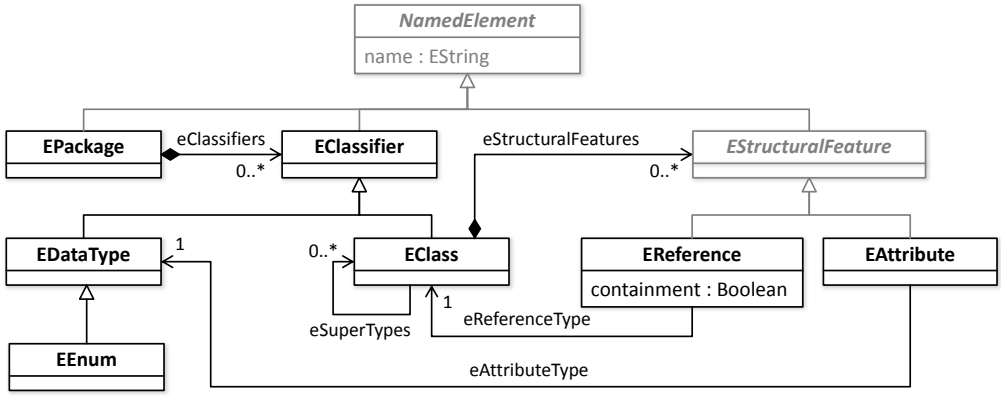


Figure 5.26: Simplified Ecore Metamodel

The binding between the Ecore language and our Modularization DSL is summarized in Table 5.7. Bindings for `EPackage`, `EClass`, `EDataType` and `EEnum` are quite straightforward. Each package is mapped to a module and all classifiers are mapped to a respective entity. More interesting are the bindings that produce the relationships in the generic modularization model. As we mentioned in Section 5.4.3, relationships can have different weights, depending on how strong the relationship is. In an Ecore model, we consider three relationships: containment, inheritance, and references. In particular, we define that the containment relationship is the strongest one, since a contained element cannot exist without its container. Therefore, for those references that are of type *containment*, we create a relationship with weight 3 between the entities representing the classes that act as source and target of the reference. References that are not of type *containment* are the weakest relationships in our mapping, so we give a weight of 1 to the relationships created from them. As previously mentioned, classes contain attributes that are types with a datatype or an enumeration. Therefore, a relationship that is created between an entity representing a class and those entities representing datatypes and enumerations are also given weight of 1. Finally, there are inheritance

relationships between those entities representing classes. As inheritance relationships span up type hierarchies, we give them a weight of 2, i.e., they are stronger them simple references but not as strong as containment.

Table 5.7: Correspondences Between Ecore and Generic Modularization Language

Ecore	**Generic Modularization Language**
EPackage	Module
EClass	Entity
EDataType	Entity
EEnum	Entity
eSuperType	Relationship (weight=2)
EAttribute	Relationship (weight=1)
EReference (non-containment)	Relationship (weight=1)
EReference (containment)	Relationship (weight=3)

Of course, these weights may be adjusted depending on the language or the desired modularization, e.g., when type hierarchies are preferred over containments.

According to this mapping, we have implemented an ATL transformation[2] that takes any Ecore model as input and produces a model conforming to our generic modularization metamodel as output. In the output model we create one module for each package in the source Ecore model. All entities and the relationships among them are created accordingly. Next, we implement the transformation consisting of the two steps as explained before. First, helper functions are defining the queries needed to incorporate the concepts of the modularization metamodel in the Ecore metamodel. Conceptually one can think of them as derived properties. Second, we employ one-to-one rules to actually produce the initial modularization models from the Ecore models.

Table 5.8: Ecore Models as Modularization Models

Model	**#Mod**	**#Ent**	**#Rel(w=1)**	**#Rel(w=2)**	**#Rel(w=3)**
HTML	2	62	14	42	7
Java	1	132	179	145	129
OCL	2	77	47	73	37
QVT	8	151	199	152	100

As example Ecore-based languages we use HTML, Java, OCL and QVT. These languages are available in the ATL transformation zoo[3] and represent middle-sized as well as large metamodels [KSW+13]. The initial number of modules, entities, and relationships for each of the three types for these Ecore models are summarized in Table 5.8.

[2] All artifacts can be downloaded from our website [FTW16d]

[3] `http://www.eclipse.org/atl/atlTransformations/`

Answering RQ1. Concluding, we can answer RQ1 by having provided a successful binding between Ecore models and the generic modularization models. The binding has proven to be feasible and has been implemented with our proposed approach.

Search-Based Optimization

In order to investigate the modularization quality of our approach, we apply it on the same four Ecore-based languages. The actual optimization is executed using our MOMoT approach as explained in Chapter 4. The objectives that we use as input are those described in Section 5.4.3. In order to execute the search, we deploy the NSGA-III algorithm with the following operators: tournament selection with $k = 2$, one-point crossover with $p = 1.0$, and placeholder and variable mutation, each with $p = 0.1$. The algorithm has been executed 30 times with a population size of 300 and a maximum number of $21,000$ evaluations. The result of a search-based algorithm is the set of Pareto optimal solutions.

Some works [BDDO04] argue that the most interesting solutions of the Pareto front are solutions where a small improvement in one objective would lead to a large deterioration in at least one other objective. These solutions are sometimes also called *knees*. In order to show some values for the solutions we retrieve, we extract the knee point solution for each of the four languages by calculating the proper utilities for each solution [SBS13]. The solutions shown in Table 5.9 displays the value of the different objectives before the search and for the solution considered as knee point after the search.

Table 5.9: Objectives Values Before and After Optimization

Example	**Model**	**COH ↑**	**COP ↓**	**DIF ↓**	**MOD ↑**
HTML	Before	119	0	56	2
	After	101	18	31	5
Java	Before	856	0	-	1
	After	517	339	2	7
OCL	Before	304	0	69	2
	After	262	42	45	4
QVT	Before	587	216	38	8
	After	448	355	2	8

Answering RQ2. Let us respond to RQ2 by studying the numbers in the table. As we can see, the value of coupling (COP) for the first three example models before the optimization is 0. In the case of Java, this is obvious as there is only one module (MOD). Regarding HTML and OCL, there are two modules, where one of those only has 3 to 4 entities without any dependency with any entity from the other module. This is due to the fact that these modules contain only the primitive types such as Boolean or

Integer. As for QVT, since there are 8 modules, the value of coupling is larger than 0. These modules come from the packages in the metamodel and are for example QVT Template, Imperative OCL, EMOF, or QVT Operational. As for the difference between the minimum and maximum number of entities in each module (DIF), this value is quite high for HTML, OCL, and QVT, i.e., the entities between the modules are not distributed equally. In the Java language, on the other hand, this value does not make sense as there is only one module. Finally, regarding cohesion (COH), all initial languages present a very high to optimal value due to the fact that most, if not all, entities are in the same module.

If we investigate the values after the optimization is performed, we see that in HTML we have now 5 modules, 7 in Java, 4 in OCL and we keep the same number of modules in QVT. Since in most cases we have more modules than before the optimization, the values of cohesion and coupling have gotten worse, as it is obvious. However, the value of DIF is improved to a great extent. Therefore, we are sacrificing coupling and cohesion in favour of having several balanced modules.

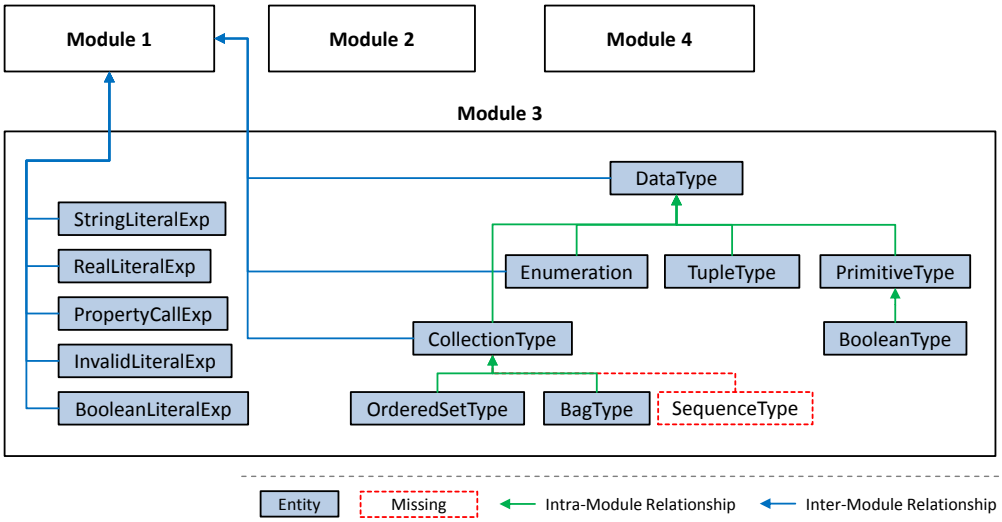


Figure 5.27: Excerpt of the OCL Knee-Point Solution

This is especially visible when we manually inspect the knee solution of OCL depicted in Figure 5.27. In this solution, two groups of entities have been grouped into a single module, i.e., the datatypes and the literal expressions, although they are not related in order to balance the total number of entities between the modules. Furthermore, we can see that, although our approach already provides good objective values, there is still room for improvement as the `SequenceType` is not in the same module as its super class `CollectionType`. However, this may be solved by letting the optimization procedure run longer.

Another interesting result we found is the HTML knee solution, partially depicted in Figure 5.28. Besides some single entities which are probably grouped together to optimize

DIF, we expected to see the elements of the description list (DL), i.e., the term that is being described (DT) and the description data (DD), in the same module as they are used in the same semantic context. However, it seems that in the HTML metamodel that we have used for our case study, there is no connection between these elements. As such, our solutions helped us realize that the metamodel is probably faulty in this aspect.

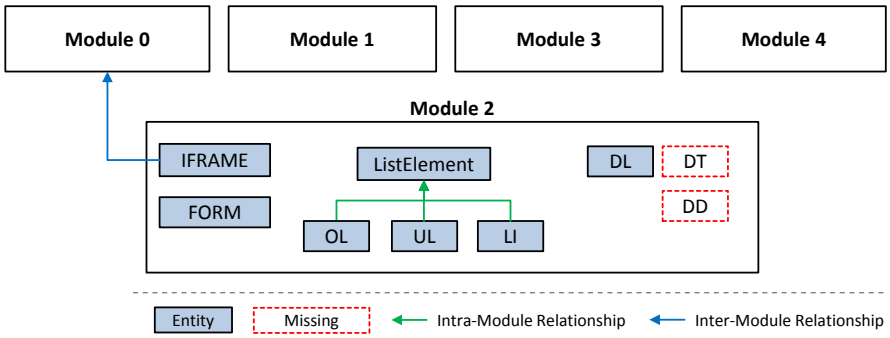


Figure 5.28: Excerpt of the HTML Knee-Point Solution

Resulting from this manual inspection, we argue that balancing the number of the entities between the modules might be a good metric for software system, however for languages another approach might yield results closer to what we would expect.

It should be noted, that these are the results produced by Step 3 of our approach, i.e., the results conform to the optimized generic modularization model. The last step would be to transform these solutions back to the original modeling language. This is left as future work and would require to inverse the transformation produced in Step 1 and Step 2. As we are using a query structured approach and one-to-one mappings (isomorphism) for Step 1 and Step 2, we the inverse transformation should be straight-forward. Nevertheless, this may not be true for all modeling languages, resulting in the challenging problem of bridging the syntactic and semantic heterogeneities between the respective language and our generic modularization metamodel. This challenge has also been recognized by Durán et al. [DZT12] who specify explicit conditions to construct a syntactically consistent and a semantically valid binding between two metamodels and by Cuadrado et al. [CGdL14] who created a dedicated binding DSL to express the bridging of structural heterogeneities between different concepts present in the metamodels.

CHAPTER 6

Conclusion and Future Work

Summarized, the context of this thesis is the area of MDE where models are the central artifacts that represent problems under consideration. These models conform to modeling languages that enable us to express the problems with domain-specific vocabulary. Any changes on these models are realized through model transformations, most of which are based on transformation rules. However, depending on the number of rules, the number of parameters, and the objectives that should be achieved by the transformation, the orchestration of these transformations is a difficult task typically delegated to the model engineer. In general, search-based optimization methods can be used to solve such problems, however using those methods is also non-trivial as they are based on a formalism very different from MDE. Therefore, in this thesis, we presented contributions towards the integration of search-based optimization methods with model-driven engineering techniques in order to facilitate the solving of complex problems on model level.

6.1 Conclusion

We introduced an approach called MOMoT, which enables model engineers to apply metaheuristic optimization methods in order to solve these using MDE techniques by automatically deriving Pareto-optimal transformation orchestrations. In particular, we presented an algorithm-agnostic approach that is able to support various optimization methods while remaining in the model-engineering technical space. The core idea of our approach is to encode solutions on the model level generically as a sequence of transformation units. Whereas these transformation units can incorporate knowledge specific to the respective problem domain, their execution is done generically using a transformation engine. The search for good transformation orchestrations is guided by a declarative set of objective and constraints which are realized through model-based property analysis techniques. By staying on the model level and using the transformation rules directly in the encoding for the search process, we produce solutions that can be

easily understood by model engineers. Another strong point of our approach is that the user does not need to be an expert in metaheuristic optimization techniques. In fact, we implemented an easy-to-use configuration language where the user can configure different options for the search process, retrieve information on different aspects of the search, and receive initial feedback on the search configuration through advices, warnings, and errors provided by a static analysis procedure.

Besides MOMoT, we presented an approach to analyze dynamic properties that consider the contention of resources directly on model level by utilizing the novel fUML standard that provides execution semantics for a subset of UML. By analyzing properties directly on model level we avoid the problems of approaches that translate models into dedicated analysis formalisms but suffer from the complexity of defining these translation transformations and mapping the results back onto model level. Most notably, the fUML subset covers the most important modeling concepts for defining the structure of a system using UML classes and its behavior using UML activities and is accompanied by a virtual machine which serves as a reference implementation of the execution semantics. Using this virtual machine and a pre-defined set of MARTE stereotypes on UML models, we obtain timed, model-based execution traces for a particular workload. These traces are then analyzed and processed to compute non-functional, dynamic properties such as the utilization or throughput of a particular resource modeled as a class and the overall execution time. In particular, for our approach we focused on the contention of resources by considering the temporal relation of several execution traces. The analysis results are then directly annotated on the provided models. This approach has been demonstrated using a software application as case study and the retrieved results have been confirmed by comparing them with the results of an established queuing network solver. As such, this approach constitutes a contribution to the state of the art realized in the course of this thesis.

Apart from detailing our approach of combining search-based optimization and model transformations and our approach to analyze dynamic properties on model level, we also evaluated MOMoT using several case studies. In order to use MOMoT, the problem domains of the case studies have been expressed through dedicated modeling languages, the problem instances have been specified as models, the modification operations have been realized as model transformations, and the objectives and constraints have been expressed declaratively.

As a first evaluation of MOMoT, we used four known case studies from the area of software engineering and model-driven engineering to assess the applicability of our approach, investigate the overhead that is introduced in comparison to a native encoded solution, and discuss the search features provided by MOMoT. Summarizing, we have shown that our approach produces correct and sound solutions w.r.t. the constraints and the objectives defined by the model engineer. However, the performance of MOMoT is not as good as the performance of a native encoding as we inherit the problem of of finding and applying matches in graphs as part of using model transformations, which is highly complex and computationally expensive. Nevertheless, MOMoT removes the complexity

of finding such a native encoding and provides features which are not provided in existing transformation approaches such as the application of search-based optimization methods.

Additionally, we presented two MOMoT-based approaches to tackle novel problems in model-driven engineering. In particular, we were able to propose, for the first time in the MDE literature, a new automated search-based approach based on NSGA-III that tackles the challenge of model transformation modularization, and introduce a first approach to deal with the modularization of modeling languages in a generic and reusable way. Specifically, the approaches were realized by formulating the problems as a many-objective search problem and use search-based methods to calculate a set of Pareto-optimal solutions based on different quality properties proposed in modularization literature. The first approach has been evaluated in detail for seven ATL transformations to demonstrate the analysis features of MOMoT and the results have been manually analyzed based on two user studies with participants from academia and engineers from Ford. Our results show that modularizing model transformations requires a sophisticated approach and that our approach based on MOMoT produces good results. Furthermore, the use of modularized transformations versus non-modularized ones can reduce the complexity to perform common tasks in model-driven engineering and can improve productiveness in terms of time needed to perform these tasks. Based on these good results, in the second approach we proposed to deal with the modularization of any kind of modeling language in a generic and reusable way. We achieved this goal by combining generic, query-structured, and search-based transformation approaches. The result of this approach for Ecore-based languages as a case study seem very promising.

In conclusion, we can state that MOMoT is a useful approach that can be applied to solve highly complex problems on model level. MOMoT is problem-independent and supports several different optimization methods such as local search and evolutionary multi-objective algorithms. Using MOMoT, a model engineer can declaratively specify the objectives and constraints and receives dedicated support when configuring the search process. Moreover, MOMoT features sufficient analysis capabilities to make it applicable for evaluating novel problem formulations.

6.2 Future Work

Building upon the research conducted in this thesis and the promising results retrieved during the evaluation, we discuss in the following several, interesting lines of research for the future. In particular, we foresee further investigations on the applicability of MOMoT using out-place transformations, and work on the integration of memetic algorithms and dedicated optimization method modeling languages.

Out-place Transformations. In the evaluation of this thesis, we focused on several case studies that use in-place transformations to optimize a given model based on a set of objectives. However, there are no conceptual restrictions that prevent our approach from being also used on problems that warrant the use of out-place transformations, i.e.,

transformations that create new models from scratch. For instance, problems that need to translate a model from one semantic domain to another semantic domain modeled through different metamodels fall under this category. In future research, we envision the application of MOMoT also on these kinds of problems.

Memetic Algorithms. In the course of this thesis, we have demonstrated how local search algorithms and, in particular, evolutionary algorithms as representative for population-based algorithms can be used to solve complex problems on model level. While local search is better suited to intensify the search in a particular area of the search space, population-based algorithms are better in providing a diversified search in the search space. Memetic algorithms [Mos89], also called genetic local search algorithms [Tal09], are a family of hybrid metaheuristic algorithms which incorporate local search algorithms within population-based algorithms in order to balance these two aspects. Another main idea of memetic algorithms is to introduce problem and instance-dependent knowledge to speed up the search process [Mos99]. As a result, many memetic algorithms of previously standalone evolutionary algorithms have been proposed, such as the M-PAES [KC00] or the algorithm incorporating neighborhood search in NSGA-II [DG01], called M-NSGA-II in [CLV07]. As part of future research, we foresee the integration of memetic algorithms in MOMoT and the investigation of their performance in comparison with other algorithms. In practice, this integration is most easily done on the implementation level by providing abstract and concrete memetic algorithm classes which inherit from the abstract algorithm classes used in MOMoT.

Optimization Method Languages. Besides the integration of memetic algorithms as a combination of population-based search and local, neighborhood search, there exist also dedicated modeling languages to support the implementation of algorithms. For instance, Localizer [MH97, MH99, MH00] is a language that allows the expression of local search algorithms in a notation close to their informal description in scientific papers. Similar, Numerica [HMD97] is a language for global optimization algorithms that provides similar functionality. As future research, such languages can be incorporated into MOMoT to give experienced users additional capabilities to control the behavior of the optimization method within the search space besides algorithm selection. In practice, this incorporation can be done by either importing the language concepts into the language of MOMoT and allow the algorithm specification directly as part of the configuration or by generating the respective algorithm classes from the specifications which are then automatically recognized by the MOMoT language.

APPENDIX A

MOMoT Configuration DSL

This appendix provides the complete grammar of the MOMoT configuration DSL described in Section 4.8 and the configuration for the Modularization case study described in Section 4.2. The implemented static analysis checks based on the DSL and further example configurations can be found on GitHub and our project website [FTW16a].

A.1 Grammar

Listing A.1 Grammar of the MOMoT Configuration DSL

```
1: grammar at.ac.tuwien.big.momot.lang.MOMoT with org.eclipse.xtext.xbase.Xbase
2:
3: import "http://www.eclipse.org/xtext/xbase/Xbase" as xbase
4: generate momot "http://www.big.tuwien.ac.at/momot/lang/MOMoT"
5:
6: MOMoTSearch:
7:    ("package" package=QualifiedName)?
8:    importSection=XImportSection?
9:    variables += VariableDeclaration*
10:
11:    ("initialization" OpSingleAssign initialization=XBlockExpression)?
12:    "search" (name=ValidID)? OpSingleAssign searchOrchestration=SearchOrchestration
13:    "experiment" OpSingleAssign experimentOrchestration=ExperimentOrchestration
14:    ("analysis" OpSingleAssign analysisOrchestration=AnalysisOrchestration)?
15:    ("results" OpSingleAssign resultManagement=ResultManagement)?
16:    ("finalization" OpSingleAssign finalization=XBlockExpression)?;
17:
18: OpKeyAssign:
19:    ":";
20:
21: VariableDeclaration:
22:    "var" type=JvmTypeReference? name=ValidID (OpSingleAssign init=XExpression)?;
23:
24: ArrayLiteral returns xbase::XListLiteral:
25: {xbase::XListLiteral}
26:    '[' elements+=XExpression (',' elements+=XExpression )* ']';
27:
```

```
28: ModuleOrchestration:
29:    "{"
30:       "modules" OpSingleAssign modules = ArrayLiteral
31:       ("ignoreUnits" OpSingleAssign unitsToRemove = ArrayLiteral)?
32:       ("ignoreParameters" OpSingleAssign nonSolutionParameters = ArrayLiteral)?
33:       ("parameterValues" OpSingleAssign "{"
34:          (parameterValues += ParmeterValueSpecification)*
35:       "}")?
36:    "}";
37:
38: ParmeterValueSpecification:
39: {ParmeterValueSpecification}
40:    name=XExpression OpKeyAssign call=XConstructorCall;
41:
42: SearchOrchestration:
43: {SearchOrchestration}
44:    "{"
45:       "model" OpSingleAssign model = InputModel
46:       "solutionLength" OpSingleAssign solutionLength = XExpression
47:       "transformations" OpSingleAssign moduleOrchestration = ModuleOrchestration
48:       "fitness" OpSingleAssign fitnessFunction = FitnessFunctionSpecification
49:       "algorithms" OpSingleAssign algorithms = AlgorithmList
50:       ("equalityHelper" OpSingleAssign equalityHelper=EqualityHelper)?
51:    "}";
52:
53: InputModel:
54:    path = XExpression (adaptation=XBlockExpression)?;
55:
56: EqualityHelper:
57:    (call = XConstructorCall | method = XBlockExpression);
58:
59: AlgorithmList:
60:    "{" (specifications += AlgorithmSpecification)+ "}";
61:
62: FitnessFunctionSpecification:
63:    (constructor = XConstructorCall)? "{"
64:       ("preprocess" OpSingleAssign preprocess = XBlockExpression)?
65:       "objectives" OpSingleAssign "{"
66:          (objectives += FitnessDimensionSpecification)+
67:       "}"
68:       ("constraints" OpSingleAssign "{"
69:          (constraints += FitnessDimensionSpecification)+
70:       "}")?
71:       ("postprocess" OpSingleAssign postprocess = XBlockExpression)?
72:       ("solutionRepairer" OpSingleAssign solutionRepairer = XConstructorCall)?
73:    "}";
74:
75: FitnessDimensionSpecification:
76:    FitnessDimensionConstructor | FitnessDimensionXBase | FitnessDimensionOCL;
77:
78: enum FitnessDimensionType:
79:    MINIMIZE = "minimize" | MAXIMIZE = "maximize";
80:
81: FitnessDimensionConstructor:
82:    name=ValidID OpKeyAssign type=FitnessDimensionType call=XConstructorCall;
83:
84: FitnessDimensionXBase:
85:    name=ValidID OpKeyAssign type=FitnessDimensionType value=XBlockExpression;
86:
87: FitnessDimensionOCL:
88:    name=ValidID OpKeyAssign type=FitnessDimensionType query=XStringLiteral
89:    ("{" defExpressions += DefExpression* "}")?;
```

```
DefExpression:
   "def" expression = STRING;

AlgorithmSpecification:
   name=ValidID OpKeyAssign call=XExpression;

ExperimentOrchestration:
{ExperimentOrchestration}
   "{"
      "populationSize" OpSingleAssign populationSize = XExpression
      "maxEvaluations" OpSingleAssign maxEvaluations = XExpression
      "nrRuns" OpSingleAssign nrRuns = XNumberLiteral
      ("referenceSet" OpSingleAssign referenceSet = XExpression)?
      ("progressListeners" OpSingleAssign "["
         (progressListeners += XConstructorCall
         "," progressListeners += XConstructorCall)*)?
      "]")?
      ("collectors" OpSingleAssign "["
         (collectors = CollectorArray customCollectors+= XConstructorCall
         "," customCollectors += XConstructorCall)*)?
      "]")?
   "}";

CollectorArray:
{CollectorArray}
   ((hypervolume ?= "hypervolume")? &
   (generationalDistance ?= "generationalDistance")? &
   (invertedGenerationalDistance ?= "invertedGenerationalDistance")? &
   (spacing ?= "spacing")? &
   (additiveEpsilonIndicator ?= "additiveEpsilonIndicator")? &
   (contribution ?= "contribution")? &
   (r1 ?= "R1")? &
   (r2 ?= "R2")? &
   (r3 ?= "R3")? &
   (adaptiveMultimethodVariation ?= "adaptiveMultimethodVariation")? &
   (adaptiveTimeContinuation ?= "adaptiveTimeContinuation")? &
   (approximationSet ?= "approximationSet")? &
   (epsilonProgress ?= "epsilonProgress")? &
   (elapsedTime ?= "elapsedTime")? &
   (populationSize ?= "populationSize")?);

AnalysisOrchestration:
   "{"
      "indicators" OpSingleAssign indicators=IndicatorArray &
      "significance" OpSingleAssign significance=XNumberLiteral &
      "show" OpSingleAssign show=ShowArray &
      ("grouping" OpSingleAssign grouping = AnalysisGroupList)? &
      (saveCommand = SaveAnalysisCommand)? &
      (boxplotCommand = BoxplotCommand)? &
      (printCommand = PrintAnalysisCommand)?
   "}";

AnalysisGroupList:
   "{" (group += AnalysisGroupSpecification)+ "}";

IndicatorArray:
{IndicatorArray}
   "["
      ((hypervolume ?= "hypervolume")? &
      (generationalDistance ?= "generationalDistance")? &
      (invertedGenerationalDistance ?= "invertedGenerationalDistance")? &
      (spacing ?= "spacing")? &
```

```
152:       (additiveEpsilonIndicator ?= "additiveEpsilonIndicator")? &
153:       (contribution ?= "contribution")? &
154:       (r1 ?= "R1")? &
155:       (r2 ?= "R2")? &
156:       (r3 ?= "R3")? &
157:       (maximumParetoFrontError ?= "maximumParetoFrontError")?)
158:    "]";
159:
160: ShowArray:
161: {ShowArray}
162:    "["
163:       ((individual ?= "individualValues")? &
164:       (aggregate ?= "aggregateValues")? &
165:       (statisticalSignificance ?= "statisticalSignificance")?)
166:    "]";
167:
168: AnalysisGroupSpecification:
169: {AnalysisGroupSpecification}
170:    name=ValidID OpKeyAssign algorithms=AlgorithmReferences;
171:
172: AlgorithmReferences:
173: {AlgorithmReferences}
174:    "["
175:       (elements+=[AlgorithmSpecification]
176:       (',' elements+=[AlgorithmSpecification])*)?
177:    "]";
178:
179: AnalysisCommand:
180:    PrintAnalysisCommand | SaveAnalysisCommand | BoxplotCommand;
181:
182: PrintAnalysisCommand:
183: {PrintAnalysisCommand}
184:    "printOutput";
185:
186: SaveAnalysisCommand:
187:    "outputFile" OpSingleAssign file=XStringLiteral;
188:
189: BoxplotCommand:
190:    "boxplotDirectory" OpSingleAssign directory=XStringLiteral;
191:
192: ResultManagement:
193: {ResultManagement}
194:    "{" (commands += ResultManagementCommand)* "}";
195:
196: ResultManagementCommand:
197:    ObjectivesCommand | SolutionsCommand | ModelsCommand ;
198:
199: ObjectivesCommand:
200: {ObjectivesCommand}
201:    "objectives" OpSingleAssign "{"
202:       (("algorithms" OpSingleAssign algorithms=AlgorithmReferences)? &
203:       ("neighborhoodSize" OpSingleAssign  &
204:       (neighborhoodSize = INT | maxNeighborhoodSize ?= "maxNeighborhoodSize"))? &
205:       ("outputFile" OpSingleAssign file=STRING)? &
206:       (printOutput ?= "printOutput")?)
207:    "}";
208:
209: SolutionsCommand:
210: {SolutionsCommand}
211:    "solutions" OpSingleAssign "{"
212:       (("algorithms" OpSingleAssign algorithms=AlgorithmReferences)? &
213:       ("neighborhoodSize" OpSingleAssign &
```

```
214:        (neighborhoodSize = INT | maxNeighborhoodSize ?= "maxNeighborhoodSize"))? &
215:        ("outputFile" OpSingleAssign file=STRING)? &
216:        ("outputDirectory" OpSingleAssign directory=STRING)? &
217:        (printOutput ?= "printOutput")?)
218:     "}";
219:
220: ModelsCommand:
221: {ModelsCommand}
222:     "models" OpSingleAssign "{"
223:        (("algorithms" OpSingleAssign algorithms=AlgorithmReferences)? &
224:        ("neighborhoodSize" OpSingleAssign  &
225:        (neighborhoodSize = INT | maxNeighborhoodSize ?= "maxNeighborhoodSize"))? &
226:        ("outputDirectory" OpSingleAssign directory=STRING)? &
227:        (printOutput ?= "printOutput")?)
228:     "}";
```

A.2 Example: Modularization Configuration

Listing A.2 Textual Configuration for the Modulariztion Case Study

```
 1: var attribute = "calculation"
 2:
 3: initialization  = {
 4:    ModularizationPackage.eINSTANCE.class // register package in standalone
 5: }
 6:
 7: search = {
 8:    model = { file = "input/models/mtunis.xmi" }
 9:    solutionLength = 50 // maximum number of transformation units
10:
11:    transformations = {
12:       modules = [ "data/modularization_jsep.henshin" ]
13:       parameterValues = { // for user parameters
14:          ModularizationRules.CreateModule.Parameter::MODULE_NAME :
15:             new IncrementalStringValue("Module", "A")
16:       }
17:    }
18:
19:    fitness = {
20:       preprocess = { // use attribute storage for external calculation
21:          val root = MomotUtil.getRoot(
22:             solution.execute,
23:             typeof(ModularizationModel))
24:          solution.setAttribute(attribute, new ModularizationCalculator(root))
25:       }
26:       objectives = {
27:          Coupling : minimize { // java-like syntax
28:             val calculator = solution.getAttribute(
29:                attribute,
30:                typeof(ModularizationCalculator))
31:             calculator.metrics.coupling
32:          }
33:          Cohesion : maximize {
34:             val calculator = solution.getAttribute(
35:                attribute,
36:                typeof(ModularizationCalculator))
37:             calculator.metrics.cohesion
38:          }
39:          NrModules : maximize "modules->size()" // OCL-specification
```

```
40:          MQ : maximize {
41:             val calculator = solution.getAttribute(
42:                attribute,
43:                typeof(ModularizationCalculator))
44:             calculator.metrics.modularizationQuality
45:          }
46:          MinMaxDiff : minimize {
47:             val calculator = solution.getAttribute(
48:                attribute,
49:                typeof(ModularizationCalculator))
50:             calculator.metrics.minMaxDiff
51:          }
52:          SolutionLength : minimize new TransformationLengthDimension // generic
53:       }
54:       constraints = { // mark invalid solutions
55:          UnassignedClasses : minimize {
56:             (root as ModularizationModel).classes
57:                .filter[c | c.module == null].size
58:          }
59:          EmptyModules : minimize {
60:             (root as ModularizationModel).^modules
61:                .filter[m | m.classes.empty].size
62:          }
63:       }
64:       solutionRepairer = new TransformationPlaceholderRepairer
65:    }
66:
67:    algorithms = {
68:       NSGAIII : moea.createNSGAIII(
69:          0, 6,
70:          new TournamentSelection(2),
71:          new OnePointCrossover(1.0),
72:          new TransformationPlaceholderMutation(0.10),
73:          new TransformationVariableMutation(orchestration.searchHelper, 0.10))
74:       eMOEA : moea.createEpsilonMOEA(
75:          0.02,
76:          new TournamentSelection(2),
77:          new OnePointCrossover(1.0),
78:          new TransformationPlaceholderMutation(0.10),
79:          new TransformationVariableMutation(orchestration.searchHelper, 0.10))
80:       RS : moea.createRandomSearch
81:    }
82:
83:    // define index-based equivalence between modules
84:    equalityHelper = {
85:       if(left instanceof Module && right instanceof Module) {
86:          val lhs = (left.eContainer as ModularizationModel).^modules.indexOf(left)
87:          val rhs = (right.eContainer as ModularizationModel)
88:             .^modules.indexOf(right)
89:          return lhs.equals(rhs)
90:       }
91:       left.equals(right)
92:    }
93: }
94:
95: experiment = {
96:    populationSize = 300
97:    maxEvaluations = 21000
98:    nrRuns = 30
99:    progressListeners = [ new SeedRuntimePrintListener ]
100: }
101:
```

```
102: analysis = {
103:    indicators = [ hypervolume generationalDistance contribution ]
104:    significance = 0.01
105:    show = [ aggregateValues statisticalSignificance individualValues ]
106:    outputFile = "output/analysis/mtunis_statistic.txt"
107: }
108:
109: results = {
110:    // save objectives
111:    objectives = {
112:       outputFile = "output/objectives/mtunis_objectives_all.pf"
113:       printOutput
114:    }
115:    // save MOEA objectives
116:    objectives = {
117:       algorithms = [ eMOEA NSGAIII ]
118:       outputFile = "output/objectives/mtunis_objectives_moea.pf"
119:    }
120:    // save solutions, i.e., transformation orchestrations
121:    solutions  = {
122:       outputFile = "output/solutions/all_solutions.txt"
123:       outputDirectory = "output/solutions/"
124:    }
125:    // save models resulting from the orchestrated transformations
126:    models = {
127:       outputDirectory = "output/models/mtunis_statistic2/"
128:    }
129:    // select kneepoint models for further inspection
130:    models = {
131:       neighborhoodSize = maxNeighborhoodSize
132:       outputDirectory  = "output/models/kneepoints/"
133:    }
134: }
```

List of Figures

List of Tables

List of Listings

Bibliography

[AB05] Cyrille Artho and Armin Biere. Combined Static and Dynamic Analysis. *Electronic Notes in Theoretical Computer Science*, 131:3–14, 2005.

[AB11] Andrea Arcuri and Lionel 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)*, pages 1–10, 2011.

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 121–135, 2010.

[AdLG15] Nuno Amálio, Juan de Lara, and Esther Guerra. Fragmenta: A Theory of Fragmentation for MDE. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 106–115. IEEE, 2015.

[ADS11] Hani Abdeen, Stephane Ducasse, and Houari Sahraoui. Modularization Metrics: Assessing Package Organization in Legacy Large Object-Oriented Software. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 394–398, 2011.

[ADSA09] Hani Abdeen, Stéphane Ducasse, Houari A. Sahraoui, and Ilham Alloui. Automatic Package Coupling and Cycle Minimization. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, pages 103–112, 2009.

[Agr03] Aditya Agrawal. Graph Rewriting And Transformation (GReAT): A Solution For The Model Integrated Computing (MIC) Bottleneck. In *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*, pages 364–368, 2003.

[AKH03] Colin Atkinson, Thomas Kühne, and Brian Henderson-Sellers. Systematic Stereotype Usage. *Software and Systems Modeling*, 2(3):153–163, 2003.

[AKM13] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-Oriented Software Design. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, pages 604–621, 2013.

[AL99] Nicolas Anquetil and Timothy Lethbridge. Experiments With Clustering as a Software Remodularization Method. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*, pages 235–255, 1999.

[ALS08] Carsten Amelunxen, Elodie Legros, and Andy Schürr. Generic and Reflective Graph Transformations for the Checking and Enforcement of Modeling Guidelines. In *Proceedings of the 9th Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 211–218, 2008.

[And14] AndroMDA Team. AndroMDA. `http://andromda.sourceforge.net`, 2014. Accessed March 2016.

[ASB09] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. Supporting View-Based Development through Orthographic Software Modeling. In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 71–86, 2009.

[AT13] Thorsten Arendt and Gabriele Taentzer. A Tool Environment for Quality Assurance Based on the Eclipse Modeling Framework. *Automated Software Engineering*, 20(2):141–184, 2013.

[AVS+14] Hani Abdeen, Dániel Varró, Houari A. Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-Objective Optimization in Rule-based Design Space Exploration. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE)*, pages 289–300, 2014.

[BAHT15] Kristopher Born, Thorsten Arendt, Florian Heß, and Gabriele Taentzer. Analyzing Conflicts and Dependencies of Rule-Based Transformations in Henshin. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 165–168, 2015.

[BBCI+13] Alexander Bergmayr, Hugo Brunelière, Javier Luis Canovas Izquierdo, Jesus Gorronogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela, and Manuel Wimmer. Migrating Legacy Software to the Cloud with ARTIST. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 465–468, 2013.

[BBL76] Barry W. Boehm, John R. Brown, and Myron Lipow. Quantitative Evaluation of Software Quality. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 592–605, 1976.

[BBSG11] Slim Bechikh, Lamjed Ben Said, and Khaled Ghédira. Searching for Knee Regions of the Pareto Front Using Mobile Reference Points. *Soft Computing*, 15(9):1807–1823, 2011.

[BCJM10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, pages 173–174, 2010.

[BCS09] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. JMT: Performance Engineering Tools for System Modeling. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[BDDO04] Jürgen Branke, Kalyanmoy Deb, Henning Dierolf, and Matthias Osswald. Finding Knees in Multi-Objective Optimization. In *Proceedings of the 8th International Conference on Parallel Problem Solving from Nature (PPSN)*, pages 722–731, 2004.

[BED14] Dominique Blouin, Yvan Eustache, and Jean-Philippe Diguet. Extensible Global Model Management with Meta-Model Subsets and Model Synchronization. In *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages (GEMOC)*, pages 43–52, 2014.

[BET10] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Lifting Parallel Graph Transformation Concepts of Model Transformation based on the Eclipse Modeling Framework. *Electronic Communications of the European Association of Software Science and Technology*, 26:1–19, 2010.

[Béz05] Jean Bézivin. On the Unification Power of Models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[bFKLW12] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. Search- Based Detection of High-Level Model Changes. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pages 212–221, 2012.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, pages 273–280, 2001.

[BGH+14] Anne Berry, Alain Gutierrez, Marianne Huchard, Amedeo Napoli, and Alain Sigayret. Hermes: A Simple and Efficient Algorithm for Building the AOC-Poset of a Binary Relation. *Annals of Mathematics and Artificial Intelligence*, 72(1-2):45–71, 2014.

[BGKS14] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. Model Checking of CTL-Extended OCL Specifications. In *Proceedings of the 7th International Conference on Software Language Engineering (SLE)*, pages 221–240, 2014.

[BGWK14] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. JUMP-From Java Annotations to UML Profiles. In *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 552–568, 2014.

[BIG16] BIG: Business Informatics Group. Moliz: Model Execution. `http://modelexecution.org`, 2016. Accessed March 2016.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf H. Reussner. Model-Based Performance Prediction with the Palladio Component Model. In *Proceedings of the 6th International Workshop on Software and Performance (WOSP)*, pages 54–65, 2007.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf H. Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[BLM13] Luca Berardinelli, Philip Langer, and Tanja Mayerhofer. Combining fUML and Profiles for Non-Functional Analysis Based on Model Execution Traces. In *Proceedings of the 9th International Conference on the Quality of Software Architectures (QoSA)*, 2013.

[BMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[BMP09] Simona Bernardi, José Merseguer, and Dorina C. Petriu. A Dependability Profile within MARTE. *Software and Systems Modeling*, 10(3):313–336, 2009.

[Bor15] Francis Bordeleau. Why and Where Do We Need Model Execution? In *Proceedings of the 1st International Workshop on Executable Modeling (EXE)*, pages 1–2, 2015.

[BR03] Christian Blum and Andrea Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[BSC+14] Islem Baki, Houari A. Sahraoui, Quentin Cobbaert, Philippe Masson, and Martin Faunes. Learning Implicit and Explicit Control in Model Transformations by Example. In *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 636–652, 2014.

[BTWV15] Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, 2015.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, pages 359–364, 2002.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A New Symbolic Model Verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, pages 495–499, 1999.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CCGT09] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. *Journal of Software*, 4(9):943–958, 2009.

[CD08] Massimiliano Caramia and Paolo Dell'Olmo. *Multi-Objective Management in Freight Logistics: Increasing Capacity, Service Level and Safety with Optimization Algorithms*, chapter Multi-Objective Optimization, pages 11–36. Springer London, 2008.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework*. Springer Berlin Heidelberg, 2007.

[CDREP10] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: A Bidirectional and Change Propagating Transformation Language. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE)*, pages 183–202, 2010.

[CGdL11] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Generic Model Transformations: Write Once, Reuse Everywhere. In *Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 62–77, 2011.

[CGdL14] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. A Component Model for Model Transformations. *IEEE Transactions on Software Engineering*, 40(11):1042–1060, 2014.

[CGM08] Jesús Cuadrado and Jesús García Molina. Approaches for Model Transformation Reuse: Factorization and Composition. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 168–182, 2008.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Proceedings of the 17th International Conference on Automated Software Engineering (ASE)*, pages 267–270, 2002.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[Cla08] Edmund M. Clarke. *25 Years of Model Checking: History, Achievements, Perspectives*, chapter The Birth of Model Checking, pages 1–26. Springer, 2008.

[CLR+09] Zhenbang Chen, Zhiming Liu, Anders P. Ravn, Volker Stolz, and Naijun Zhan. Refinement and Verification in Component-Based Model-Driven Design. *Science of Computer Programming*, 74(4):168–196, 2009.

[CLV07] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer US, 2nd edition, 2007.

[CM78] Joseph P. Cavano and James A. McCall. A Framework for the Measurement of Software Quality. *ACM SIGSOFT Software Engineering Notes*, 3(5):133–139, 1978.

[CM09] Jesús Sánchez Cuadrado and Jesús García Molina. Modularization of Model Transformations Through a Phasing Mechanism. *Software and Systems Modeling*, 8(3):325–345, 2009.

[Coh88] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 2nd edition, 1988.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

[DA95] Kalyanmoy Deb and Ram Bhushan Agrawal. Simulated Binary Crossover for Continuous Search Space. *Complex Systems*, 9(2):115–148, 1995.

[DAPM02] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[Dar59] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, London, 1859.

[DG01] Kalyanmoy Deb and Tushar Goel. A Hybrid Multi-Objective Evolutionary Approach to Engineering Shape Design. In *Proceedings of the 1st International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, pages 385–399, 2001.

[DGM11] Mauro Luigi Drago, Carlo Ghezzi, and Raffaela Mirandola. Towards Quality Driven Exploration of Model Transformation Spaces. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 2–16, 2011.

[DGM15] Mauro Luigi Drago, Carlo Ghezzi, and Raffaela Mirandola. A Quality-Driven Extension to the QVT-Relations Transformation Language. *Computer Science - Research and Development*, 30(1):1–20, 2015.

[DJ12] Kalyanmoy Deb and Himanshu Jain. Handling Many-Objective Problems Using an Improved NSGA-II Procedure. In *Proceedings of the 7th World Congress on Evolutionary Computation (CEC)*, pages 1–8, 2012.

[DJ14] Kalyanmoy Deb and Himanshu Jain. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2014.

[DJVV14] Joachim Denil, Maris Jukss, Clark Verbrugge, and Hans Vangheluwe. Search-Based Model Optimization Using Model Transformations. In *Proceedings of the 8th International Conference on System Analysis and Modeling (SAM)*, pages 80–95, 2014.

[dLV02] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In *Proceedings of of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 174–188, 2002.

[DM05] Antinisca Di Marco. *Model-based Performance Analysis of Software Architectures*. PhD thesis, University of L'Aquila, 2005.

[DMM03] Kalyanmoy Deb, Manikanth Mohan, and Shikhar Mishra. A Fast Multi-objective Evolutionary Algorithm for Finding Well-Spread Pareto-Optimal Solutions. KanGAL Report 2003002, Indian Institute of Technology Kanpur, 2003.

[dOB11] Márcio de Oliveira Barros. Evaluating Modularization Quality as an Extra Objective in Multiobjective Software Module Clustering. In *Proceedings of the 3rd International Symposium on Search Based Software Engineering (SSBSE)*, pages 267–267, 2011.

[Dor92] Marco Dorigo. *Optimization, Learning and Natural Algorithms (in Italian).* Phd thesis, Politecnico di Milano, 1992.

[Dro95] R. Geoff Dromey. A Model for Software Product Quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.

[dS15] Alberto Rodrigues da Silva. Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[DZT12] Francisco Durán, Steffen Zschaler, and Javier Troya. On the Reusable Specification of Non-functional Properties in DSLs. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE)*, pages 332–351, 2012.

[Ecl] Eclipse Foundation. ATL Transformations. `http://www.eclipse.org/atl/atlTransformations/`. Accessed March 2016.

[Ecl11] Eclipse Foundation. JET2: Java Emitter Templates. `http://projects.eclipse.org/projects/modeling.m2t.jet`, 2011. Accessed March 2016.

[Ecl14] Eclipse Foundation. EMF Refactor. `http://www.eclipse.org/emf-refactor/`, 2014. Accessed March 2016.

[Ecl15] Eclipse Foundation. Atlas Transformation Language – ATL. `http://eclipse.org/atl`, 2015. Accessed March 2016.

[Ecl16a] Eclipse Foundation. Acceleo. `http://projects.eclipse.org/projects/modeling.m2t.acceleo`, 2016. Accessed March 2016.

[Ecl16b] Eclipse Foundation. Eclipse Modeling Framework (EMF). `https://eclipse.org/modeling/emf/`, 2016. Accessed March 2016.

[Ecl16c] Eclipse Foundation. Moka Overview. `https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`, 2016. Accessed March 2016.

[Ecl16d] Eclipse Foundation. Xbase. https://wiki.eclipse.org/Xbase, 2016. Accessed March 2016.

[Ecl16e] Eclipse Foundation. XPand. http://projects.eclipse.org/projects/modeling.m2t.xpand, 2016. Accessed March 2016.

[Ecl16f] Eclipse Foundation. Xtext - Language Engineering Made Easy! https://www.eclipse.org/Xtext/, 2016. Accessed March 2016.

[Edg81] Francis Ysidro Edgeworth. *Mathematical Psychics*. P. Keagan, London, 1881.

[ES11] A. E. Eiben and S. K. Smit. Parameter Tuning for Configuring and Analyzing Evolutionary Algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[EWZ14] Dionysios Efstathiou, James R. Williams, and Steffen Zschaler. Crepe Complete: Multi-Objective Optimisation for Your Models. In *Proceedings of the 1st International Workshop on Combining Modelling with Search- and Example-Based Approaches (CMSEBA)*, pages 25–34, 2014.

[FBL+13] Martin Fleck, Luca Berardinelli, Philip Langer, Tanja Mayerhofer, and Vittorio Cortellessa. Resource Contention Analysis of Cloud-based System through fUML-driven Model Execution. In *Proceedings of the 5th International Workshop Non-functional Properties in Modeling: Analysis, Languages and Processes (NiM-ALP)*, pages 6–15, 2013.

[FHH03] Deji Fatiregun, Mark Harman, and Robert M. Hierons. Search Based Transformations. In *Proceedings of the 5th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 2511–2512, 2003.

[FHH04] Deji Fatiregun, Mark Harman, and Robert M. Hierons. Evolving Transformation Sequences using Genetic Algorithms. In *Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 66–75, 2004.

[Fis74] Peter C. Fishburn. Lexicographic Orders, Utilities and Decision Rules: A Survey. *Management Science*, 20(11):1442–1471, 1974.

[Fog62] Lawrence J. Fogel. Toward Inductive Inference Automata. In *Proceedings of the 2nd International Federation for Information Processing (IFIP)*, pages 395–399, 1962.

[Fog66] Lawrence J. Fogel. *Intelligence Through Simulated Evolution*. John Wiley & Sons, 1966.

[Fow99] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FP09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley, 2009.

[FR07] Robert B. France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Proceedings of the International Workshop on the Future of Software Engineering (FOSE)*, pages 37–54, 2007.

[FSB13] Martin Faunes, Houari A. Sahraoui, and Mounir Boukadoum. Genetic-Programming Approach to Learn Model Transformation Rules from Examples. In *Proceedings of the 6th International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 17–32, 2013.

[FTKW16] Martin Fleck, Javier Troya, Marouane Kessentini, and Manuel Wimmer. Model Transformation Modularization as a Many-Objective Optimization Problem. 2016. Submitted for Review to *IEEE Transactions on Software Engineering*.

[FTW15] Martin Fleck, Javier Troya, and Manuel Wimmer. Marrying Search-based Optimization and Model Transformation Technology. In *Proceedings of the 1st North American Symposium on Search Based Software Engineering (NasBASE)*, pages 1–16, 2015. Preprint available at `http://martin-fleck.github.io/momot/downloads/NasBASE_MOMoT.pdf`.

[FTW16a] Martin Fleck, Javier Troya, and Manuel Wimmer. MOMoT - Marrying Search-based Optimization and Model Transformation Technology. `http://martin-fleck.github.io/momot/`, 2016. Accessed March 2016.

[FTW16b] Martin Fleck, Javier Troya, and Manuel Wimmer. Objective-Driven Model Transformations. 2016. Revision Submitted to *Journal of Software: Evolution and Process*.

[FTW16c] Martin Fleck, Javier Troya, and Manuel Wimmer. Towards Generic Modularization Transformations. In *Companion Proceedings of the 15th International Conference on Modularity, 1st International Workshop on Modularity in Modelling (MOMO)*, pages 190–195, 2016.

[FTW16d] Martin Fleck, Javier Troya, and Manuel Wimmer. Transformation Chain for Ecore models. `http://martin-fleck.github.io/momot/casestudy/generic_modularization/`, 2016. Accessed March 2016.

[GDM14] Hamid Gholizadeh, Zinovy Diskin, and Tom Maibaum. A Query Structured Approach for Model Transformation. In *Proceedings of the 3rd Workshop on the Analysis of Model Transformations (AMT)*, pages 54–63, 2014.

[GGKdL14] Antonio Garmendia, Esther Guerra, Dimitrios S. Kolovos, and Juan de Lara. EMF Splitter: A Structured Approach to EMF Modularity. In *Proceedings of the 3rd Workshop on Extreme Modeling (XM)*, pages 22–31, 2014.

[GHH14] Martin Gogolla, Lars Hamann, and Frank Hilken. On Static and Dynamic Analysis of UML and OCL Transformation Models. In *Proceedings of the 3rd Workshop on the Analysis of Model Transformations (AMT)*, pages 24–33, 2014.

[GK10] Joel Greenyer and Ekkart Kindler. Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. *Software and Systems Modeling*, 9(1):21–46, 2010.

[GL85] David E. Goldberg and Robert Lingle Jr. Alleles, Loci, and the Traveling Salesman Problem. In *Proceedings of the 1st International Conference on Genetic Algorithms (ICGA)*, pages 154–159, 1985.

[Glo86] Fred Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers and Operations Research*, 13(5):533–549, 1986.

[GM93] Robert Godin and Hafedh Mili. Building and Maintaining Analysis-level Class Hierarchies Using Galois Lattices. In *Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 394–410, 1993.

[GPC05] Marcela Genero, Mario Piattini, and Coral Calero. *Metrics For Software Conceptual Models*. Imperial College Press, 2005.

[Gra92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.

[GRK13] Pieter Van Gorp, Louis M. Rose, and Christian Krause, editors. *Proceedings of the 6th Transformation Tool Contest (TTC)*, volume 135 of *Electronic Proceedings in Theoretical Computer Science*, 2013.

[Had16a] David Hadka. *Beginner's Guide to the MOEA Framework*. CreateSpace Independent Publishing, 2016.

[Had16b] David Hadka. MOEA Framework User Guide, Version 2.8. `http://moeaframework.org/`, 2016. Accessed March 2016.

[Har07] Mark Harman. The Current State and Future of Search Based Software Engineering. In *Proceedings of the Workshop on the Future of Software Engineering (FOSE)*, pages 342–357, 2007.

[Hec06] Reiko Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006.

[Hev07] Alan R. Hevner. The Three Cycle View of Design Science. *Scandinavian Journal of Information Systems*, 19(2):87–92, 2007.

[HHJZ09] Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On Language-Independent Model Modularisation. *Transactions on Aspect-Oriented Software Development*, 6:39–82, 2009.

[HHP02] Mark Harman, Robert M. Hierons, and Mark Proctor. A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization. In *Proceedings of the 4th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1351–1358, 2002.

[HJ98] Michael Pilegaard Hansen and Andrzej Jaszkiewicz. Evaluating the Quality of Approximations to the Non-Dominated Set. Technical report, Technical University of Denmark, 1998.

[HJ01] Mark Harman and Bryan F. Jones. Search-based Software Engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[HJZA07] Jakob Henriksson, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Reuseware - Adding Modularity to Your Language of Choice. *Journal of Object Technology*, 6(9):127–146, 2007.

[HM79] Ching-Lai Hwang and Abu Syed Md. Masud. *Multiple Objective Decision Making — Methods and Applications: A State-of-the-Art Survey*. Springer Berlin, 1979.

[HMD97] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica - A Modeling Language for Global Optimization*. MIT Press, 1997.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[HMTY10] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Proceedings of the International Summer School on Empirical Software Engineering and Verification (LASER)*, pages 1–59, 2010.

[Hol62] John H. Holland. Outline for a Logical Theory of Adaptive Systems. *Journal of the ACM*, 9(3):297–314, 1962.

[Hol75] John Henry Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1975.

[Hol83] Richard C. Holt. *Concurrent Euclid, the Unix* System, and Tunis*. Addison-Wesley, 1983.

[Hol92] John Henry Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[HT07] Mark Harman and Laurence Tratt. Pareto Optimal Search Based Refactoring at the Design Level. In *Proceedings of the 9th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1106–1113, 2007.

[ISO11] ISO: International Organization for Standardization. Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733`, 2011. ISO/IEC 25010:2011, Accessed March 2016.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.

[JBF11] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model Driven Language Engineering with Kermeta. In *Proceedings of the 4th International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 201–221, 2011.

[JDJ+06] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala Latvala, and Ivan Porres. Model Checking Dynamic and Hierarchical UML State Machines. In *Proceedings of the 3rd Workshop on Model Design and Validation (MoDeVa)*, 2006.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the Satellite Events of the 8th International Conference on on Model Driven Engineering Languages and Systems (MoDELS)*, pages 128–138, 2006.

[KAB02] Ivan Kurtev, Mehmet Aksit, and Jean Bézivin. Technological Spaces: An Initial Appraisal. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications (DOA)*, 2002.

[KBSB10] Marouane Kessentini, Arbi Bouchoucha, Houari A. Sahraoui, and Mounir Boukadoum. Example-Based Sequence Diagrams to Colored Petri Nets Transformation Using Heuristic Search. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA)*, pages 156–172, 2010.

[KC00] Joshua D. Knowles and David W. Corne. M-PAES: A Memetic Algorithm for Multiobjective Optimization. In *Proceedings of the 2000 Congress on Evolutionary Computation (CEC)*, pages 325–332, 2000.

[KDDFQS13] Mathias Kleiner, Marcos Didonet Del Fabro, and Davi Queiroz Santos. Transformation as Search. In *Proceedings of the 9th European Conference on Modelling Foundations and Applications (ECMFA)*, pages 54–69, 2013.

[KE95] James Kennedy and Russell C. Eberhart. Particle Swarm Optimization. In *Proceedings of the 4th International Conference on Neural Networks*, pages 1942–1948, 1995.

[KJV83] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC-FSE)*, pages 285–294, 2007.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels.* Addison-Wesley, 2008.

[KLR+12] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Model Transformation By-Example: A Survey of the First Wave. In *Conceptual Modelling and Its Theoretical Foundations*, pages 197–215, 2012.

[KLW13] Marouane Kessentini, Philip Langer, and Manuel Wimmer. Searching Models, Modeling Search: On the Synergies of SBSE and MDE. In *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 51–54, 2013.

[Koz92] John R. Koza. *Genetic Programming.* MIT Press, 1992.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 46–60, 2008.

[KR06] Harmen Kastenberg and Arend Rensink. Model Checking Dynamic States in GROOVE. In *Proceedings of the 13th International Workshop on Model Checking Software (SPIN)*, pages 299–305, 2006.

[Kra07] Jeff Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):36–42, 2007.

[KSB08] Marouane Kessentini, Houari A. Sahraoui, and Mounir Boukadoum. Model Transformation as an Optimization Problem. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 159–173, 2008.

[KSBB12] Marouane Kessentini, Houari A. Sahraoui, Mounir Boukadoum, and Omar Benomar. Search-based Model Transformation by Example. *Software and Systems Modeling*, 11(2):209–226, 2012.

[KSW+13] Angelika Kusel, Johannes Schoenboeck, Manuel Wimmer, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. Reality Check for Model Transformation Reuse: The ATL Transformation Zoo Case Study. In *Proceedings of the 2nd Workshop on the Analysis of Model Transformations (AMT)*, 2013.

[Küh06] Thomas Kühne. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[Kur05] Ivan Kurtev. *Adaptability of Model Transformations*. PhD thesis, University of Twente, 2005.

[Kur08] Ivan Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In *Proceedings of the 3rd International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, pages 377–393, 2008.

[KvdBJ07] Ivan Kurtev, Klaas van den Berg, and Frédéric Jouault. Rule-based Modularization in Model Transformation Languages Illustrated with ATL. *Science of Computer Programming*, 68(3):138–154, 2007.

[KW52] William H. Kruskal and W. Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621, 1952.

[KW06] Alexander Knapp and Jochen Wuttke. Model Checking of UML 2.0 Interactions. In *Proceedings of the Workshops and Symposia of the 8th International Conference on on Model Driven Engineering Languages and Systems (MoDELS)*, pages 42–51, 2006.

[LAD+14] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Model Transformation Intents and Their Properties. *Software and Systems Modeling*, pages 1–38, 2014.

[LKR13] Kevin Lano and Saeed Kolahdouz Rahimi. Case Study: Class Diagram Restructuring. In *Proceedings of the 6th Transformation Tool Contest (TTC)*, pages 8–15, 2013.

[LTDZ02] Marco Laumanns, Lothar Thiele, Kalyanmoy Deb, and Eckart Zitzler. Combining Convergence and Diversity in Evolutionary Multiobjective Optimization. *Evolutionary Computation*, 10(3):263–282, 2002.

[LWG+12] Philip Langer, Manuel Wimmer, Jeff Gray, Gerti Kappel, and Antonio Vallecillo. Language-Specific Model Versioning Based on Signifiers. *Journal of Object Technology*, 11(3):1–34, 2012.

[LZGS84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[May12] Tanja Mayerhofer. Testing and Debugging UML Models Based on fUML. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1579–1582, 2012.

[May13] Tanja Mayerhofer. Using fUML as Semantics Specification Language in Model Driven Engineering. In *Proceedings of the Satellite Events of the 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 87–93, 2013.

[May14] Tanja Mayerhofer. *Defining Executable Modeling Languages with fUML*. PhD thesis, TU Wien, 2014.

[MB04] Moreno Marzolla and Simonetta Balsamo. UML-PSI: The UML Performance Simulator. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST)*, pages 340–341, 2004.

[MB07] Onaiza Maqbool and Haroon Atique Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition, 2004.

[MDL+14] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In *Proceedings of the 7th*

International Conference on Software Language Engineering (SLE), pages 1–20, 2014.

[MG95] Brad L. Miller and David E. Goldberg. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems*, 9(3):193–212, 1995.

[MG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[MGG+06] Milan Milanovic, Dragan Gasevic, Adrian Giurca, Gerd Wagner, and Vladan Devedzic. Towards Sharing Rules Between OWL/SWRL and UML/OCL. *Electronic Communications of the European Association of Software Science and Technology*, 5, 2006.

[MH97] Laurent Michel and Pascal Van Hentenryck. Localizer: A Modeling Language for Local Search. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP)*, pages 237–251, 1997.

[MH99] Laurent D. Michel and Pascal Van Hentenryck. Localizer: A Modeling Language for Local Search. *INFORMS Journal on Computing*, 11(1):1–14, 1999.

[MH00] Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5(1):43–84, 2000.

[MHF+15] Assaad Moawad, Thomas Hartmann, François Fouquet, Grégory Nain, Jacques Klein, and Johann Bourcier. Polymer - A Model-Driven Approach for Simpler, Safer, and Evolutive Multi-Objective Optimization Development. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 286–293, 2015.

[Mil07] Milan Milanovic. Modeling Rules on the Semantic Web. Master's thesis, University of Belgrade, 2007.

[MJ14] Hamid Masoud and Saeed Jalili. A Clustering-Based Model for Class Responsibility Assignment Problem in Object-Oriented Analysis. *Journal of Systems and Software*, 93:110–131, 2014.

[MKB+14] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 Objectives for Automating Software Refactoring Using NSGA-III. In *Proceedings of the 16th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1263–1270, 2014.

[MKS+15] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-Objective Software Remodularization Using NSGA-III. *ACM Transactions on Software Engineering and Methodology*, 24(3):17:1–17:45, 2015.

[MLK12] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A Runtime Model for fUML. In *Proceedings of the 7th Workshop on Models@run.time (MRT)*, pages 53–58, 2012.

[MLMK13] Stefan Mijatov, Philip Langer, Tanja Mayerhofer, and Gerti Kappel. A Framework for Testing UML Activities Based on fUML. In *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa)*, pages 1–10, 2013.

[MLW13] Tanja Mayerhofer, Philip Langer, and Manuel Wimmer. xMOF: A Semantics Specification Language for Metamodeling. In *Proceedings of the Satellite Events of the 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 46–50, 2013.

[MM06] Brian S. Mitchell and Spiros Mancoridis. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[MM08] Brian S. Mitchell and Spiros Mancoridis. On the Evaluation of the Bunch Search-Based Software Modularization Algorithm. *Soft Computing*, 12(1):77–93, 2008.

[MMR+98] Spiros Mancoridis, Brian S. Mitchell, Chris Rorres, Yih-Farn Chen, and Emden R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC)*, pages 45–52, 1998.

[MMR14] Jose P. Miguel, David Mauricio, and Glen Rodriguez. A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications*, 5(6):31–53, 2014.

[MNB+13] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, Thomas Bruckmann, and Claude Poull. Automated Model-in-the-Loop Testing of Continuous Controllers Using Search. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE)*, pages 141–157, 2013.

[MNM96] Zbigniew Michalewicz, Girish Nazhiyath, and Maciej Michalewicz. A Note on Usefulness of Geometrical Crossover for Numerical Optimization Problems. In *Proceedings of the 5th Annual Conference on Evolutionary Programming*, pages 305–312, 1996.

[Mos89] Pablo Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. Technical Report C3P 826, California Institute of Technology, 1989.

[Mos99] Pablo Moscato. *New Ideas in Optimization*, chapter Memetic Algorithms: A Short Introduction, pages 219–234. McGraw-Hill, 1999.

[MRW77] Jim A. McCall, Paul K. Richarts, and Gene F. Walters. Factors in Software Quality, Volume I, II and III. Technical Report CDRL A003, US Air Force Electronic Systems Divisions and Rome Air Development Center, 1977.

[MSUW04] Stephen. J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, 2004.

[MTR07] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing Refactoring Dependencies Using Graph Transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

[MW47] Henry B. Mann and Donald R. Whitney. On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[OC03] Mark O'Keeffe and Mel Ó Cinnéide. A Stochastic Approach to Automated Design Improvement. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 59–62, 2003.

[OC08] Mark Kent O'Keeffe and Mel Ó Cinnéide. Search-Based Refactoring for Software Maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[ÓCTH+12] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental Assessment of Software Metrics Using Automated Refactoring. In *Proceedings of the 6th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 49–58. ACM, 2012.

[OKK03] Isao Ono, Hajime Kita, and Shigenobu Kobayashi. *Advances in Evolutionary Computing: Theory and Applications*, chapter A Real-Coded Genetic Algorithm using the Unimodal Normal Distribution Crossover, pages 213–237. Springer Berlin Heidelberg, 2003.

[OMG03] OMG: Object Management Group. Common Warehouse Metamodel (CWM) Specification, Version 1.1. `http://www.omg.org/spec/CWM/1.1/`, 2003. OMG Document Number: formal/2003-03-02, Accessed March 2016.

[OMG05] OMG: Object Management Group. UML Profile for Schedulability, Performance, and Time, Version 1.1. http://www.omg.org/spec/SPTP/1.1/, 2005. OMG Document Number: formal/2005-01-02, Accessed March 2016.

[OMG08] OMG: Object Management Group. MOF Model to Text Transformation Language, Version 1.0. http://www.omg.org/spec/MOFM2T/1.0/, 2008. OMG Document Number: formal/2008-01-16, Accessed March 2016.

[OMG11a] OMG: Object Management Group. Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), Version 1.3. http://www.omg.org/spec/KDM/1.3, 2011. OMG Document Number: formal/2011-08-04, Accessed March 2016.

[OMG11b] OMG: Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1. http://www.omg.org/spec/MARTE/1.1, 2011. OMG Document Number: formal/2011-06-02, Accessed March 2016.

[OMG13a] OMG: Object Management Group. Concrete Syntax for a UML Action Language: Action Language for Foundational UML (ALF), Version 1.0.1. http://www.omg.org/spec/ALF/1.0.1/, 2013. OMG Document Number: formal/2013-09-01, Accessed March 2016.

[OMG13b] OMG: Object Management Group. UML Testing Profile (UTP), Version 1.2. http://www.omg.org/spec/UTP/1.2/, 2013. OMG Document Number: formal/2013-04-03, Accessed March 2016.

[OMG14a] OMG: Object Management Group. MDA - The Architecture of Choice for a Changing World. http://www.omg.org/mda/, 2014. OMG Document Number: ormsc/14-06-01, Accessed March 2016.

[OMG14b] OMG: Object Management Group. Object Constraint Language, Version 2.4. http://www.omg.org/spec/OCL/2.4, 2014. OMG Document Number: formal/2014-02-03, Accessed March 2016.

[OMG15a] OMG: Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.2. http://www.omg.org/spec/QVT/1.2, 2015. OMG Document Number: formal/15-02-01, Accessed March 2016.

[OMG15b] OMG: Object Management Group. Meta Object Facility (MOF) Core Specification, Version 2.5. http://www.omg.org/spec/MOF/2.5/, 2015. OMG Document Number: formal/2015-06-05, Accessed March 2016.

[OMG15c] OMG: Object Management Group. OMG Unified Modeling Language (OMG UML), Version 2.5. http://www.omg.org/spec/UML/2.5, 2015. OMG Document Number: formal/2015-03-01, Accessed March 2016.

[OMG16] OMG: Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.2.1. `http://www.omg.org/spec/FUML/1.2.1`, 2016. OMG Document Number: formal/2016-01-05, Accessed March 2016.

[Par96] Vilfredo Pareto. *Cours d'Économie Politique*. F. Rouge, Lausanne, 1896.

[PAT12] Dorina C. Petriu, Mohammad Alhaj, and Rasha Tawhid. Software Performance Modeling. In *Proceedings of the 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems Formal Methods for Model-Driven Engineering*, pages 219–262, 2012.

[PHY11] Kata Praditwong, Mark Harman, and Xin Yao. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

[PW07] Dorin Bogdan Petriu and C. Murray Woodside. An Intermediate Metamodel with Scenarios and Resources for Generating Performance Models from UML Designs. *Software and Systems Modeling*, 6(2):163–184, 2007.

[QGC00] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of Java$^{\text{TM}}$ Class Loading. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 325–336, 2000.

[QH11] Fawad Qayum and Reiko Heckel. Search-Based Refactoring Using Unfolding of Graph Transformation Systems. *Electronic Communications of the European Association of Software Science and Technology*, 38, 2011.

[RDV09] Jose E. Rivera, Francisco Duran, and Antonio Vallecillo. A Graphical Approach for Modeling Time-Dependent Behavior of DSLs. In *Proceedings of the 10th Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 51–55, 2009.

[Rec65] Ingo Rechenberg. Cybernetic Solution Path of an Experimental Problem. Library Translation 1112, Royal Aircraft Establishment, 1965.

[Rec73] Ingo Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme Nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog, 1973.

[RLP+14] Shekoufeh Kolahdouz Rahimi, Kevin Lano, Suresh Pillay, Javier Troya, and Pieter Van Gorp. Evaluation of Model Transformation Approaches for Model Refactoring. *Science of Computer Programming*, 85:5–40, 2014.

[RM08] Hajo A. Reijers and Jan Mendling. Modularity in Process Models: Review and Effects. In *Proceedings of the 6th International Conference on Business Process Management (BPM)*, pages 20–35, 2008.

[RN09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 3rd edition, 2009.

[Rot89] Jeff Rothenberg. *Artificial Intelligence, Simulation & Modeling*, chapter The Nature of Modeling, pages 75–92. John Wiley & Sons, 1989.

[RS09] Lily Rachmawati and Dipti Srinivasan. Multiobjective Evolutionary Algorithm with Controllable Focus on the Knees of the Pareto Front. *IEEE Transactions on Evolutionary Computation*, 13(4):810–824, 2009.

[RSA10] Jan Reimann, Mirko Seifert, and Uwe Aßmann. Role-Based Generic Model Refactoring. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 78–92, 2010.

[SBBP05] Olaf Seng, Markus Bauer, Matthias Biehl, and Gert Pache. Search-Based Improvement of Subsystem Decompositions. In *Proceedings of the 7th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1045–1051, 2005.

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2008.

[SBS13] PradyumnKumar Shukla, Marlon Alexander Braun, and Hartmut Schmeck. Theory and Algorithms for Finding Knees. In *Proceedings of the 7th International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, pages 156–170, 2013.

[Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 151–163, 1995.

[See95] Thomas D. Seeley. *The Wisdom of the Hive*. Harvard University Press, 1995.

[Sel07] Bran Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *Proceedings of the 10th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 2–9, 2007.

[She03] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall/CRC, 3rd edition, 2003.

[SHL10] Bernhard Schätz, Florian Hölzl, and Torbjörn Lundkvist. Design-Space Exploration Through Constraint-Based Model-Transformation. In *Proceedings of the 17th International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*, pages 173–182, 2010.

[SHNS13] Hajer Saada, Marianne Huchard, Clémentine Nebut, and Houari A. Sahraoui. Recovering Model Transformation Traces Using Multi-Objective Optimization. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE)*, pages 688–693, 2013.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[SKT13] Jeffery Shelburg, Marouane Kessentini, and DanielR. Tauritz. Regression Testing for Model Transformations: A Multi-objective Approach. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE)*, volume 8084 of *LNCS*, pages 209–223. Springer, 2013.

[SLP10a] Connie U. Smith, Catalina M. Lladó, and Ramón Puigjaner. Performance Model Interchange Format (PMIF 2): A Comprehensive Approach to Queueing Network Model Interoperability. *Performance Evaluation*, 67(7):548–568, 2010.

[SLP10b] Connie U. Smith, Catalina M. Lladó, and Ramón Puigjaner. PMIF Extensions: Increasing the Scope of Supported Models. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 255–256, 2010.

[SRTC14] Daniel Strüber, Julia Rubin, Gabriele Taentzer, and Marsha Chechik. Splitting Models Using Information Retrieval and Model Crawling Techniques. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 47–62, 2014.

[SSB06] Olaf Seng, Johannes Stammel, and David Burkhart. Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems. In *Proceedings of the 8th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1909–1916, 2006.

[SST13] Daniel Strüber, Matthias Selter, and Gabriele Taentzer. Tool Support for Clustering Large Meta-Models. In *Proceedings of the 1st Workshop on Scalability in Model Driven Engineering (BigMDE)*, pages 1–7, 2013.

[ST09] Mark Shtern and Vassilios Tzerpos. Methods for Selecting and Improving Software Clustering Algorithms. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC)*, pages 248–252, 2009.

[SW02] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 2nd edition, 2002.

[Tae03] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proceedings of the 2nd Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, pages 446–453, 2003.

[Tal09] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation.* Wiley Publishing, 2009.

[TBF+15] Javier Troya, Hugo Brunelière, Martin Fleck, Manuel Wimmer, Leire Orue-Echevarria, and Jesús Gorroñogoitia. ARTIST: Model-Based Stairway to the Cloud. In *Proceedings of the Projects Showcase at the Federation on Software Technologies: Applications and Foundations (STAF)*, pages 1–8, 2015.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications ECMDA-FA*, pages 18–33, 2009.

[TKL13] Matthias Tichy, Christian Krause, and Grischa Liebel. Detecting Performance Bad Smells for Henshin Model Transformations. In *Proceedings of the 2nd Workshop on the Analysis of Model Transformations (AMT)*, 2013.

[TMB14] Amjed Tahir, Stephen G. MacDonell, and Jim Buchan. A Study of the Relationship Between Class Testability and Runtime Properties. In *Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 63–78, 2014.

[TRV10] Javier Troya, José Eduardo Rivera, and Antonio Vallecillo. Simulating Domain Specific Visual Models By Observation. In *Proceedings of the Spring Simulation Multiconference (SpringSim)*, pages 1–8, 2010.

[TYH99] Shigeyoshi Tsutsui, Masayuki Yamamura, and Takahide Higuchi. Multi-Parent Recombination with Simplex Crossover in Real Coded Genetic Algorithms. In *Proceedings of the 1st Conference on Genetic and Evolutionary Computation (GECCO)*, pages 657–664, 1999.

[Var06] Dániel Varró. Model Transformation By Example. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 410–424, 2006.

[Vis01] Eelco Visser. A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57(2):109–143, 2001.

[vMvHH11] Robert von Massow, André van Hoorn, and Wilhelm Hasselbring. Performance Simulation of Runtime Reconfigurable Component-Based Software Architectures. In *Proceedings of the 5th European Conference on Software Architecture (ECSA)*, pages 43–58, 2011.

[VP04] Dániel Varró and András Pataricza. Generic and Meta-Transformations for Model Transformation Engineering. In *Proceedings of the 7th International Conference on The Unified Modelling Language: Modelling Languages and Applications (UML)*, pages 290–304, 2004.

[Wig97] Theo A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE)*, pages 33–43, 1997.

[WKR+11] Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Reusing Model Transformations Across Heterogeneous Metamodels. *Electronic Communications of the European Association of Software Science and Technology*, 50, 2011.

[WM97] David H. Wolpert and William. G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[WPP+05] C. Murray Woodside, Dorina C. Petriu, Dorin Bogdan Petriu, Hui Shen, Toqeer Israr, and José Merseguer. Performance By Unified Model Analysis (PUMA). In *Proceedings of the 5th International Workshop on Software and Performance (WOSP)*, pages 1–12, 2005.

[WSD10] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module Superimposition: A Composition Technique for Rule-Based Model Transformation Languages. *Software and Systems Modeling*, 9(3):285–309, 2010.

[WSK+11] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elisabeth Kapsammer. A Survey on UML-Based Aspect-Oriented Design Modeling. *ACM Computing Surveys*, 43(4):1–28, 2011.

[WSKK07] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards Model Transformation Generation By-Example. In *Proceedings of the 40th Hawaii International Conference on Systems Science (HICSS)*, page 285, 2007.

[Yan10] Xin-She Yang. *Nature-Inspired Metaheuristic Algorithms*. Luniver Press, 2nd edition, 2010.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1st edition, 1979.

[YD09] Xin-She Yang and Suash Deb. Cuckoo Search via Lévy Flights. In *Proceedings of the World Congress on Nature and Biologically Inspired Computing (NaBIC)*, pages 210–214, 2009.

[YK96] Yasuo Yonezawa and Takashi Kikuchi. Ecological Algorithm for Optimal Ordering Used By Collective Honey Bee Behavior. In *Proceedings of the 7th International Symposium on Micro Machine and Human Science (MHS)*, pages 249–256, 1996.

[ZK04] Eckart Zitzler and Simon Künzli. Indicator-Based Selection in Multiobjective Search. In *Proceedings of the 8th International Conference on Parallel Problem Solving from Nature (PPSN)*, pages 832–842, 2004.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. TIK-Report 103, Computer Engineering and Networks Laboratory (TIK) and Swiss Federal Institute of Technology (ETH) Zurich, 2001.

[Zsc09] Steffen Zschaler. Formal Specification of Non-Functional Properties of Component-Based Software Systems. *Software and Systems Modeling*, 9(2):161–201, 2009.

[ZTL+03] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M. Fonseca, and Viviane Grunert da Fonseca. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.

Martin Fleck, MSc | CV

Barawitzkgasse 34/4/50 – 1190 Wien – Austria

+43 (1) 588 01 – 188 649 • fleck@big.tuwien.ac.at
www.big.tuwien.ac.at/mfleck

Education

Doctoral Studies in Technical Sciences **Dr.techn.**
TU Wien, Business Informatics Group *Since 2012*
Title: Search-Based Model Transformations
Advisor: Priv.Doz. Mag. Dr. Manuel Wimmer

Master Studies in Software Engineering **MSc**
University of Applied Sciences Upper Austria *2010 – 2012*
Thesis: Profiling Users in Social Networks (written in german)
Supervisor: Dr. Stefan Mitsch
Second Supervisor: Dr. Stephan Lechner
One semester studying at the KTH Royal Institute of Technology, Sweden

Bachelor Studies in Software Engineering **BSc**
University of Applied Sciences Upper Austria *2007 – 2010*
Thesis Part 1: 3D Positioning on Human Model Surfaces (in german)
Supervisor: Dipl.-Ing. Dr. Stephan Dreiseitl
Thesis Part 2: Java Application to Correct Chemical Formulas (in german)
Supervisor: Dr. Masakazu Suzuki
One semester internship at the Institute of Systems, IT and Nanotechnologies, Japan

Professional Experience

Researcher
TU Wien, Business Informatics Group *09/2015 – 03/2016*
Research Interests: MDE, SBSE
Teaching: Model Engineering, Web Engineering

Project Assistant
TU Wien, Business Informatics Group *11/2012 – 09/2015*
European Union FP7 Project ARTIST:
Advanced software-based seRvice provisioning and migraTIon of legacy SofTware
Teaching: Model Engineering, Web Engineering

Tutor
University of Applied Sciences Upper Austria *10/2009 – 06/2011*
Teaching: Relational Databases, SQL, and XML

Developer for Technical Information Systems
Engineering Center Steyr, Austria *07/2009 – 08/2009*
Technologies: Qt, C++

Intern
Institute of Systems, IT and Nanotechnologies, Japan *04/2010 – 06/2010*
Internship during Bachelor Studies
Technologies: Java, Swing

Tutor
University of Applied Sciences Upper Austria *10/2008 – 06/2009*
Teaching: Algorithms and Data Structures

Publications

Peer-Reviewed Journal Papers

[1] Martin Fleck, Javier Troya, Marouane Kessentini, and Manuel Wimmer. Model Transformation Modularization as a Many-Objective Optimization Problem. 2016. Submitted for Review to *IEEE Transactions on Software Engineering*.

[2] Martin Fleck, Javier Troya, and Manuel Wimmer. Objective-Driven Model Transformations. 2016. Revision Submitted to *Journal of Software: Evolution and Process*.

Peer-Reviewed Conference and Symposium Papers

[1] Martin Fleck, Javier Troya, and Manuel Wimmer. Marrying Search-based Optimization and Model Transformation Technology. In *Proceedings of the 1st North American Symposium on Search Based Software Engineering (NasBASE)*, pages 1–16, 2015. Preprint available at `http://martin-fleck.github.io/momot/downloads/NasBASE_MOMoT.pdf`.

[2] Martin Fleck, Javier Troya, and Manuel Wimmer. Search-Based Model Transformations with MOMoT. In *Proceedings of the 9th International Conference on Theory and Practice of Model Transformations (ICMT)*, 2016. Accepted for Publication.

Peer-Reviewed Workshop and Project Showcase Papers

[1] Martin Fleck, Luca Berardinelli, Philip Langer, Tanja Mayerhofer, and Vittorio Cortellessa. Resource Contention Analysis of Cloud-based System through fUML-driven Model Execution. In *Proceedings of the 5th International Workshop Non-functional Properties in Modeling: Analysis, Languages and Processes (NiM-ALP)*, pages 6–15, 2013.

[2] Martin Fleck, Javier Troya, Philip Langer, and Manuel Wimmer. Towards Pattern-Based Optimization of Cloud Applications. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 16–25, 2014.

[3] Martin Fleck, Javier Troya, and Manuel Wimmer. The Class Responsibility Assignment Case. In *Proceedings of the 9th Transformation Tool Contest (TTC)*, 2016. Accepted for Publication.

[4] Martin Fleck, Javier Troya, and Manuel Wimmer. Towards Generic Modularization Transformations. In *Companion Proceedings of the 15th International Conference on Modularity, 1st International Workshop on Modularity in Modelling (MOMO)*, pages 190–195, 2016.

[5] Javier Troya, Hugo Brunelière, Martin Fleck, Manuel Wimmer, Leire Orue-Echevarria, and Jesús Gorroñogoitia. ARTIST: Model-Based Stairway to the Cloud. In *Proceedings of the Projects Showcase at the Federation on Software Technologies: Applications and Foundations (STAF)*, pages 1–8, 2015.