On the Quality Properties of Model Transformations: Performance and Correctness

UNIVERSIDAD
DE MÁLAGA

Loli Burgueño

Departamento de Lenguajes y Ciencias de la Computación
University of Malaga

Supervised by
Antonio Vallecillo and Manuel Wimmer

April 2016

To those I love. To those who love me.

El Dr. Antonio Vallecillo Moreno, Catedrático de Universidad del Departamento de Lenguajes y Ciencias de la Computación de la E.T.S. de Ingeniería Informática de la Universidad de Málaga, y el Dr. Manuel Wimmer, profesor perteneciente al Business Informatics Group en la Universidad Tecnológica de Viena,

Certifican que Dña. Dolores Burgueño Caballero, Ingeniera Informática, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada:

On the Quality Properties of
Model Transformations:
Performance and Correctness

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, abril de 2015

Fdo. Antonio Vallecillo
Catedrático de Universidad
Dpto. Leng. y Ciencias de la Computación
Universidad de Málaga

Fdo. Manuel Wimmer
Associate Profesor
Bussiness Informatics Group
Vienna University of Technology

Acknowledgements

This thesis has been supported by the fellowship BES-2012-057064 granted by the Programme for the Training of Researchers of the Ministry of Economy and Competitiveness of Spain and the Spanish research projects TIN2011-23795 and TIN2014-52034-R.

Special Acknowledgements

Mucha gente me ha acompañado durante el desarrollo de esta tesis doctoral, tanto en el ámbito académico como en el personal. Algunos llevan conmigo tanto tiempo que no recuerdo la vida sin ellos. Otros llegaron más tarde y aun así supieron hacerse notar. A todos y cada uno, gracias.

En primer lugar me gustaría mostrar mis agradecimientos a mis directores de tesis, Antonio Vallecillo y Manuel Wimmer. Antonio, gracias por depositar tu confianza en mí y ofrecerme la posibilidad de realizar esta tesis doctoral, espero no haberte defraudado. Gracias por dedicarme parte de tu tiempo aun estando tan ocupado, gracias por tus buenos consejos e ideas y gracias por darme un trato tan agradable. Gracias por tener siempre una sonrisa en la cara y por ser tan entusiasta y optimista, son cosas que se transmiten. Sinceramente, no creo que hubiera podido tener un mejor director de tesis. Manuel, although your stay in Málaga was short (a year and a half is short) and the distance makes the communication difficult, thank you for helping me so much, for always being willing to share your ideas with me, for your advice, for finding the way to work together, for inviting me to Vienna, and a long etcetera. I wish you could have stayed longer in Málaga. We miss you.

También tengo que dar las gracias a todos los miembros de Atenea y en especial a Javi Troya. Javi, gracias por prestarme tanta ayuda y por tu paciencia, sobre todo al principio que era cuando más lo necesitaba.

Gracias también a todos mis compañeros del 3.3.3. por ayudarme en la medida de lo posible, por hacer amenos tantos almuerzos y por los buenos ratos que hemos pasado fuera de las cuatro paredes del laboratorio. Las largas horas de trabajo se hacen más pasajeras cuando la compañía es buena. Gracias a Lola y Jose Luis Reyes por vuestra eficiencia y amabilidad a la hora de resolver los temas administrativos.

I would like to thank Jeff Gray and Eugene Syriani for giving me the chance to spend four months at the University of Alabama working with them. It was a very fruitful stay from which I learnt a lot. At a personal level, thank you, Jeff and Eugene, for treating me so well. Also, thanks to all the members of the group for considering me one of them since the first day. Finally, all my American friends, and specially my three roommates, deserve a special thank you. Thank you for being like my American family and making me feel at home. Without you, nothing would have been the same. Also, I want to thank Gábor Karsai for offering me the possibility to live another experience like this and learn from him and his team at the University of Vanderbilt.

I am very grateful to Martin Gogolla and his team. In the first place, for putting their trust in me and, secondly, for their hard work on our collaborations. It is always a pleasure to work with you.

Quiero dar las gracias a todos mis amigos, a los más cercanos y a los más lejanos, a los que veo con más frecuencia y a los que veo con menos. Gracias por hacerme pasar tantos buenos momentos. Como siempre digo, “no hay que vivir para trabajar sino trabajar para vivir”, y en ese “vivir” es donde vosotros ponéis vuestro granito de arena para que el “trabajar” merezca la pena.

Por último, gracias a toda mi familia por apoyarme y confiar en mí. Gracias a mis hermanas, Petri y Mari Celi por estar siempre ahí, en las buenas y en las malas. Gracias, papá, por desvivirte por mis hermanas y por mí para que nunca nos falte nada, gracias por tu nobleza y por tus infinitos gestos de cariño. Gracias, mamá, por todo lo que haces por mí día a día, por ser un ejemplo a seguir, por los valores que, junto con papá, me has transmitido y me sigues transmitiendo. Gracias también por tus castigos, por tu insistencia y tu paciencia cuando de pequeña no quería estudiar, sin ti ni esta tesis doctoral existiría ni sería la persona que soy a día de hoy.

Abstract

The increasing complexity of software due to continuous technological advances has motivated the use of models in the software development process. Initially, models were mainly used as drafts to help developers understand their programs. Later they were used extensively and a new discipline called Model-Driven Engineering (MDE) was born. In the MDE paradigm, aside from the models themselves, model transformations (MT) are garnering interest as they allow the analysis and manipulation of models. Therefore, the performance, scalability and correctness of model transformations have become critical issues and thus they deserve a thorough study. Existing model transformation engines are principally based on sequential and in-memory execution strategies, and hence their capabilities to transform very large models in parallel and in distributed environments are limited. Current tools and languages are not able to cope with models that are not located in a single machine and, even worse, most of them require the model to be in a single file. Moreover, once a model transformation has been written and executed—either sequentially or in parallel—it is necessary to rely on methods, mechanisms, and tools for checking its correctness.

In this dissertation, our contribution is twofold. Firstly, we introduce a novel execution platform that permits the parallel execution of both out-place and in-place model transformations, regardless of whether the models fit into a single machine memory or not. This platform can be used as a target for high-level transformation language compilers, so that existing model transformations do not need to be rewritten in another language but only have to be executed more efficiently. Another advantage is that a developer who is familiar with an existing model transformation language does not need to learn a new one.

In addition to performance, the correctness of model transformations is an essential aspect that needs to be addressed if MTs are going to be used in realistic industrial settings. Due to the fact that the most popular model transformation languages are rule-based, i.e., the transformations written in those languages comprise rules that define how the model elements are transformed, the second contribution of this thesis is a static approach for locating faulty rules in model transformations. Current approaches able to fully prove correctness—such as model checking techniques—require an unacceptable amount of time and memory. Our approach cannot fully prove correctness but can be very useful for identifying bugs at an early development stage, quickly and cost effectively.

Table of contents

List of figures

List of tables

Acronyms

MDE	Model-Driven Engineering
MDA	Model-Driven Architecture
MT	Model Transformation
M2M	Model-to-Model
M2T	Model-to-Text
T2M	Text-to-Model
MM	Metamodel
HOT	High-Order Model Transformation
CASE	Computer-Aided Software Engineering
OMG	Object Management Group
ATL	ATLAS Transformation Language
QVT	Query/View/Transformation
ETL	Epsilon Transformation Language
EMF	Eclipse Modeling Framework
RQ	Research Question
CC	Constraint Coverage
RC	Rule Coverage
RCR	Relatedness of Constraints and Rules
CT	Classifying Term
USE	UML-based Specification Environment
OCL	Object Constraint Language

Chapter 1
Introduction

There is no doubt that software currently plays an essential role in our society. The needs that we humans have for software are increasing. The more present software is in our daily life, the more we demand from it. Therefore, the problems it has to solve are increasingly complex. Software as we now understand it, i.e., instructions that are executed in digital machines, first appeared in the late 1940s and its instructions were written directly in binary code. Since then, we have placed several abstraction layers on top of the binary code to facilitate the writing of more complex programs—nowadays, it is not only a matter of easing the writing but of making it possible. One of these attempts has led to Model Driven Engineering (MDE).

MDE is an approach for software development that was developed with the intention of manipulating the complexity of large software systems by considering only those aspects that were useful for a specific purpose and leaving out superfluous details. All this is achieved through dedicated models. Models capture the aspects of interest of systems and behave as an abstraction of them, representing reality for a given purpose. Thus, models are simpler, safer and/or cheaper than reality and allow users to deal with the interesting parts of the real systems in a simplified and more focused way. This helps avoid the complexity, danger and irreversibility of real scenarios.

Alongside models, Model Transformations (MT) play a central role in all model-driven software engineering processes [16]. They manipulate these models to accomplish different tasks: not only generating the system implementations from the high-level models, but also for model analysis, software migration and modernization [19] and even for data integration, especially when complex data structures are involved, e.g., in Social Web data management [152].

Although MTs are essential for building systems in the MDE paradigm, we believe they are not yet mature enough to be adopted by industry. For instance, although there are different ways to represent models (such as XML representations, graphs, etc.) and approaches for transforming them (such as textual vs. visual languages, imperative/operational vs. declarative/relational languages, graph transformation languages, etc.) that could be sufficient to solve a wide range of problems, most of them are conceptual solutions and the tool support they offer is only at prototype level, which only permits the transformation of toy models. The problem is not merely a matter of re-implementing tools, in fact it is more serious than that. For instance, relational model transformation languages are computationally expensive because they have to deal with subgraph isomorphism, which is an NP-complete problem. Most of the existing solutions were designed considering only the functional requirements given by the problem to solve, and leaving out some key non-functional requirements such as performance or correctness.

Nowadays, huge amounts of data are handled and the use of the cloud or networks of computers to store it—because a single machine does not fulfil all the requirements (space, availability, etc.)—is needed. Furthermore, engineers and developers are always rushing due to strict deadlines, which means that their processes must be as efficient as possible. In practice, they need to store and handle models with millions of instances, transform these models in a reasonable amount of time and make better use of their current IT infrastructure—networks of distributed computers and multi-core machines. In the meantime, current model transformations engines lack concurrent mechanisms to execute model transformations in parallel and they do not provide the ability to transform models that are larger than the size of the RAM memory of the machine in which the model transformation is being executed.

Apart from the problems that one might find when executing a model transformation, another reason that hinders the acceptance of MDE is that MTs are hard to test and debug. Therefore, it is important to be able to count on mechanisms to prove afterwards that the model transformation execution was correct and there are no mistakes in the implementation that might affect the result. Most MT languages and engines can be seen as a black box that executes a piece of code and obtains certain results but there is no way to know the steps it followed. This is a line of research that has already been started in which some initial solutions have been proposed [1, 143]. There are mechanisms available to fully prove MT correctness but they require so much time or effort on the part of the developer to be applied that may not be worth it in many cases. In those cases, approaches like Tracts [56] that certify the correctness—instead of formally verifying it—provide useful solutions. Tracts can be seen as a specific kind of model transformation contracts [7, 33]. They are suitable for specifying model transformations and count on tool support for checking, in a black-box manner, that a given implementation behaves as expected. Nevertheless, the Tracts approach may cause developers to discard it as an option because it does not provide information as to where the problem is located in the implementation. This lack of traceability mechanisms between implementations and specifications is a serious problem when the MT's size and complexity grow in such a way that manual debugging is no longer possible.

This chapter is structured as follows. In Section 1.1 we present the motivation behind this work, Section 1.2 shows our contributions to the state of the art and Section 1.3 explains the structure of this thesis.

1.1 Motivation and Challenges

Model Driven Engineering raises the level of abstraction in the software development process and allows developers and engineers to deal with complex problems using simpler solutions. Nevertheless, the afore-mentioned weaknesses of model transformations may prevent the extended adoption of the MDE paradigm. These weaknesses have motivated our work, presented in this thesis. Specifically, this thesis addresses the following two research questions:

RQ1 *Is it possible to provide a concurrent approach and the appropriate mechanisms to support the parallel and distributed execution of MTs?*

RQ2 *Is it possible to build traceability mechanisms between implementations and specifications of MTs?*

These research questions have been carefully studied throughout the course of this dissertation.

1.2 Contribution

The research carried out in order to respond to this thesis' research questions has resulted in two main contributions:

(i) a model transformation engine called LinTra that transforms models—that might be distributed over a set of machines—in parallel and,

(ii) a light-weight testing approach that can be used as a first step towards identifying bugs in model transformation implementations at an early stage, quickly and cost effectively.

These two contributions comprise several (sub-)contributions. With respect to LinTra, we have developed a Java-based execution platform that makes use of the Linda coordination language and provides the possibility of executing model transformations in parallel even when models do not fit inside the computer RAM memory. This platform lets the developer choose between two execution modes: out-place and in-place. An out-place execution of a model transformation involves two kinds of models: input models that are read-only and output models that are created from scratch. By contrast, in-place model transformations do not create new models but rather evolve the input models. LinTra also provides the functionality to execute, in parallel, model transformation chains, which are transformations in which the output of a transformation is the input of the following. Current model transformation languages that do not support concurrency or distribution could be compiled to LinTra so that model transformations written in that language could be executed in parallel, also allowing the transformation of larger models.

With regard to our testing approach, as rule-based model transformation languages are the most popular languages, our first contribution is a light-weight and static approach for locating faulty rules in model-to-model transformations. This is used in conjunction with Tracts in such a way that when Tracts identify that there is a mistake in a model transformation, our approach is able to point to where the problem might be in the implementation, i.e., it provides a traceability mechanism between implementations and specifications of MT's. Since our approach is not an exhaustive testing technique nor it can fully prove correctness, we also provide a method to identify whether or not a given transformation is suitable for our testing approach. Our second contribution is the extension of the model-to-model testing approach to model-to-text and text-to-model transformations.

1.3 Outline

The remaining dissertation chapters are structured as follows:

Chapter 2. Background

We present some concepts, technologies and tools that have served as the basis of this dissertation. The MDE methodology is presented, together with its main concepts which are models, metamodels and model transformations. The Linda coordination language, on which our parallel model transformation engine is based, is also presented. Finally, we present Tracts, a black-box testing mechanism for model transformations, and a simple case study that shows how Tracts are used.

Chapter 3. Parallel Out-place Model Transformations

This chapter presents a novel Java-based execution platform that offers a set of core features for the parallel execution of out-place transformations that can be used as a target for high-level transformation language compilers. It is inspired by the concepts and principles of the Linda coordination language, and the use of data parallelism to achieve parallelization. This platform also provides mechanisms for parallel execution of model transformation chains—where there are several transformations to be executed and the output of a transformation is the input of the following transformation. The evaluation presented in this chapter, in which we have compared our engine against several state-of-the-art model transformation engines, reports significant gains in performance and scalability.

Chapter 4. Parallel In-place Model Transformations

This chapter complements Chapter 3 by permitting the parallel execution of in-place model transformations. It discusses the fundamentals of in-place model transformations in light of their parallel execution and provide LinTra with an in-place execution mode.

Chapter 5. Testing Model-to-Model Transformations

This chapter presents a light-weight and static approach for locating faulty rules in Model-to-Model (M2M) transformations, based on matching functions that automatically establish the alignments between specifications and implementations using the metamodel footprints. It is implemented for the combination of Tracts and ATL and is supported by the corresponding toolkit. The chapter ends with an evaluation of the accuracy and limitations of the approach and identifies kinds of transformations which are most suitable for validation with the proposed approach and use mutation techniques to evaluate its effectiveness.

Chapter 6. Extending Tracts for Model-to-Text and Text-to-Model Transformations

While Chapter 5 presents an approach for testing Model-to-Model transformations, this chapter presents a mechanism to reuse that approach for model transformations for which there is no explicit model representation in one of their domains (input or output), i.e., for Model-to-Text (M2T) and Text-to-Model (T2M) transformations. We reduce the M2T or T2M transformation specification problem into an equivalent M2M transformation specification problem by representing the textual part conforming to a generic text metamodel. Two case studies demonstrate its applicability. The chapter ends with the application of the approach to evaluate the code generation capabilities of several existing UML tools.

Chapter 7. Conclusions and Future work

This chapter summarizes the contributions explained in the different chapters and highlights the contributions of our work. Furthermore, we detail the main publications derived from these contributions and discuss the lessons we have learnt. Finally, we outline our prospective lines of future work.

Appendix A. Similarity Matrixes

As we mentioned, Chapter 5 provides a mechanism to identify whether or not a transformation is suitable for applying the testing approach it presents. That mechanism generates a table, which is called a similarity matrix. The similarity matrixes to which Chapter 5 refers are presented in this appendix.

Appendix B. Resumen

The content of this appendix is the same as in the Abstract but is written in Spanish, i.e., it summarizes this dissertation.

Appendix C. Conclusiones y Contribuciones

This appendix reports our conclusions, the list of contributions of this dissertation and discusses future lines of work in Spanish.

Chapter 2
Background

2.1 Model-Driven Engineering

In the field of software engineering, abstractions are a key element for success. Abstraction enables understanding and/or analyzing complex domains of concern, such as programs, software systems, and their application domains, which contain a plethora of detail. In this regard, a model is a simplified and generalized representation of a real world system or concept created to facilitate its understanding.

Model-Driven Engineering is a methodology that advocates the use of models as first class entities throughout the software engineering life cycle. 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.

2.1.1 History

Over the past five decades, software engineers have been creating abstractions that help them program, focusing only on their design intent and leaving out details from the underlying computing environment such as CPU, memory, etc. and their complexities. For instance, languages such as C (released in the early 1970s) raised the level of abstraction over assembly languages so that programmers did not need to worry about low level details related to memory position access. Similarly, early operating system platforms, such as OS/360 (released in 1967) and Unix (originally developed in 1969), shielded developers from the complexities of programming directly with hardware devices [125].

Historically, Computer-Aided Software Engineering (CASE) tools developed in the 80s were considered to be the first tools to support MDE. These tools aimed to provide a graphical means of simplifying software development, whilst also generating implementation artifacts. However, they lacked standardization.

In the past two decades, the advances in programming languages and platforms have raised the level of software abstractions available to developers. Examples of this are object-oriented languages such as C++, Java, or C#, which offered a higher level of abstraction than Fortran or C. However, they still had a distinct computing-oriented focus that was a problem when the size of software as well as its complexity increased. When we talk about complexity we mean both accidental and essential complexity. Accidental complexity is caused by the specific solution that the engineer developed and the problems that that solution might carry. On the other hand, the essential complexity, named by Brooks et al. in [17], is given by the problem to be solved itself.

New problems have appeared, related to the semantic gap between the software design and its implementation—which requires an unacceptable number of lines of code. This leads to the fact that developers need to pay attention to so many programming details that it becomes difficult to focus on strategic architectural issues such as system correctness and performance.

Model-Driven Engineering is a relatively new methodology that applies lessons learnt from earlier attempts to develop higher-level platform and language abstractions. MDE tools also help detect and prevent many errors throughout the software development life cycle.

2.1.2 Models and Metamodels

A key concept in model-driven approaches is that of models. Ludewig claims in [95] that they were not invented but rather we have been using them since we have existed. Therefore, it is difficult to find a consensus of what they are, or in other words to find, a definition for the concept of *“model”*. Endless discussions have proved that there is no common understanding of them. Nevertheless, most people seem to support the idea that the particular strength of models is based on the idea of abstraction and promotion of simpler models with a greater focus on the problem space. This combined with executable semantics elevates the total level of automation possible.

According to Stachowiak a model needs to possess three features [130]: *(i)* mapping: a model is a representation of an original, *(ii)* reduction: not all the properties of the subject are mapped onto the model, *(iii)* pragmatic: a model needs to be usable in place of the original with respect to some purpose.

The Object Management Group (OMG) developed a set of standards called Model-Driven Architecture (MDA), thereby creating the foundation for this advanced architecture-focused approach. In different documents, the OMG gives different definitions. In [104] it is defined as the representation of a part of the functionality, structure and/or behavior of a system. In [105], the OMG defines a model as the description or specification of a system and its environment defined for a specific purpose. Finally, in [107] the OMG states that a model captures a view of a physical system, with a specific purpose. The purpose determines what is to be included in the model and what is irrelevant. Consequently, the model describes those aspects of the physical system which are relevant to the model's purpose, and at the right level of abstraction.

From a software engineering perspective, engineers build models principally to better understand the useful characteristics of an existing or desired system and its environment, to predict the characteristics of a system by analyzing its models, to communicate their understanding and design intent to others and to specify the implementation of the system among others. Apart from the definition and the purpose of models and modeling, there is a need to identify the main functions of models. According to Gérard and Selic in a keynote[1] given in 2010, a model must have the following characteristics in order to be useful: *(i)* purposeful, *(ii)* abstract, *(iii)* understandable, *(iv)* accurate, *(v)* predictive and *(vi)* cost-effective.

Related to models there are metamodels. A metamodel is a model that is used to describe another model. It specifies the concepts of the language, the relationships between these concepts, the structural rules that restrict the possible elements in the valid models and those combinations between elements with respect to the domain semantic rules. As a metamodel is also a model, the term "meta" is therefore relative—depending on the perspective, a model is either a model or a metamodel.

Each model is described in the language defined by its metamodel, so there is a conformance relation between a model and its metamodel. A metamodel

[1]http://www.artist-embedded.org/docs/Events/2010/FESA/slides/1_Keynote_Gerard+Selic.pdf

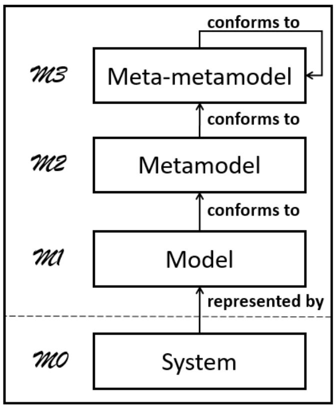


Fig. 2.1 Organization in four layers proposed by the OMG.

is in itself a model and, consequently, it is written in the language defined by its meta-metamodel. The recursive process for defining models which conform to models at a higher level of abstraction ends when a level where a model conforms to itself, is reached.

The OMG supports the four-level architecture, called Meta-Object Facility (MOF), that was illustrated by Bézivin in [14] and presented in Figure 2.1. The ***M0*** layer refers to the system in the real world. A model represents those systems at level ***M1***. This model conforms to its metamodel defined at level ***M2*** and the metamodel itself conforms to the meta-metamodel at level ***M3***. Nevertheless, OMG's standard is currently being challenged by multilevel modeling [5, 89]. Multilevel modeling tries to overcome the limitation of only four meta-levels by allowing an arbitrary number of meta-levels. This results in the concept of *clabjet* which is a model element that has properties of classes and objects. In a multilevel architecture, this dual type/instance nature makes some metamodeling facilities available at each meta-level, which can be beneficial in some situations.

A key difference between a software engineer and other engineers is that the medium in which models are built is very different. Software engineers share the same medium which is the computer, while for other engineers it could be buildings, bridges, aeroplanes, and so on. This unique feature of

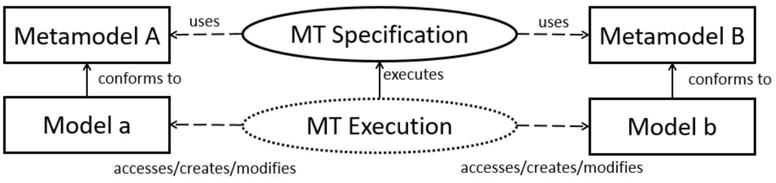


Fig. 2.2 Overview of the elements involved in a MT.

software allows automatic transformations to be defined capable of generating implementations from higher level models. This is something which is much more expensive in other disciplines. Consequently, the purpose of MDE is to make the implementation of systems as automatable as possible, achievable thanks to model transformations.

2.1.3 Model Transformations

In the field of Model-Driven Engineering, a Model Transformation (MT) allows a model to be manipulated and transformed. In the same way that there is no universal definition for the concept of model, there is no universal definition for the concept of model transformation. For instance, a highly extended definition of model transformation is the one given by Kleppe et al. [79] which states that "a transformation is the automatic generation of a target model from a source model, according to a transformation definition". On the other hand, we have tried to be more general and in [143] we state that "a model transformation is an algorithmic specification of the relationship between two or more models, and more specifically, of the mapping from one model to another". Fig. 2.2 illustrates an overview of the main concepts involved in a model transformation. There are two metamodels and two models, both of which conform to their respective metamodels. The model transformation is defined with respect to the metamodels and is executed on specific models. As we have said, this is extensible to other domains (metamodels) and models or there may be only one metamodel (this would be the case of inplace model transformations).

Model Transformations can be classified according to different criteria:

- **Directionality**: *Unidirectional* transformations are those that are defined and executed in just one direction, i.e., establishing which is/are the source and target metamodel(s) and model(s). Typical unidirectional MT languages are ATL [75], QVT Operational [OMG], etc. *Bidirectional* [68] the transformation can be executed either forwards (from source to target) or backwards (from target to source). The most extended bidirectional MT language is QVT Relations [58]. *Direction neutral* transformations are those for which the direction has not been established and they only define the relationship between the metamodels/models. An example of this kind of MT is what we call transformation models and are defined by means of OCL expressions in [69].

- **Metamodels involved in the MT**: *Exogenous* transformations are transformations defined between different metamodels while *endogenous* transformations are transformations between models that conform to the same metamodel.

- **Number of Models Involved**: *Out-place* MTs create model elements in a model based on properties of another model. Contraryly, *in-place* MTs only involve one model being evolved. Note that exogenous transformations are always out-place and that in-place transformations are a type of endogenous transformation.

- **MT Language**: There are different types of MT languages: *declarative*, *imperative* and *hybrid* which combine declarative and imperative parts.

- **Type of MT**: Text artifacts might be involved in one of the domains of model transformations which result in a further two kinds of MTs. *Model-to-Model* transformations where only models are involved as input(s) and output(s), *Model-to-Text* transformations where text artifacts are generated from a model or a set of models, and *Text-to-Model* transformations where models are created from text artifacts/repositories—for instance, MTs that reverse engineering code into models.

Listing 2.1 Example of Linda pseudocode.

```
write("circumference", 3, 47, 53)
write("circumference", 7, 20, 21)
write("square", 5, 20, 30)
read(?, ?, 20, ?)
```

A particular kind of MT is the high-order model transformation (HOT) where a model transformation is itself a model or a so-called transformation model [15]. HOTs conform to a metamodel which is part of the model transformation language's definition, i.e., they are transformations which have other transformations as input and/or output.

2.2 Linda Coordination Language

Linda is a coordination model that uses a shared memory space as the only mean of communication among parallel processes. This model is implemented as a coordination language for parallel and distributed processing. It was first proposed by David Gelernter at Yale University in the mid-1980s [51] and in recent years there has been a resurgence in interest in it, particularly with regard to Java implementations of Linda [148, 149].

In distributed memory systems, such as networks of workstations, the shared memory, which is called *tuple space*, is usually distributed among the processing nodes. Independent from the implementation strategy employed, the tuple space is structured as a bag of tuples. An example of a tuple with four fields is `("circumference", 3, 47, 53)`, where 3 is the radius, and 47 and 53 indicate the position (x and y coordinates) of the circumference represented by this tuple. Another example is `("square", 5, 20, 30)` which represents a square whose side length is 5, whose position on the X-axis is 20 and 30 on the Y-axis.

Linda provides operations, called primitives, to place tuples into tuple spaces (`write` operations) and to retrieve tuples from them (`read` operations). Read operations can be either blocking or non-blocking. A piece of Linda code with examples of these operations is shown in Listing 2.1.

The specification of the tuple to be retrieved makes use of an associative matching technique whereby a subset of the fields in the tuple have their values specified. In our example, the read operation defines a pattern that matches all the tuples whose position on the X-axis is 20. Therefore, the tuples written in the second and third lines are retrieved.

As a coordination language, the Linda primitives were conceived to be integrated with a programming language, which is called the host language. There are different Linda implementations for different host languages such as C-Linda [4] for C and JavaSpaces [96] for Java. Listing 2.2 shows a piece of Java code that, using the Linda implementation JavaSpaces, is able to read and write circumferences into the tuple space. For representing the circumferences a class implementing the `Entry` interface is needed (lines 2–13). The main program, after the configuration of the tuple space (lines 21–23), writes two circumferences into the tuple space (lines 25-29) and then reads the one that has radius 3 (lines 31-34).

2.3 Model Transformation Contracts. Tracts

2.3.1 Specifying Transformations with Tracts

Tracts were introduced in [56] as a specification and black-box testing mechanism for model transformations. They provide modular pieces of specification, each one focusing on a particular transformation scenario. Thus each model transformation can be specified by means of a set of Tracts, each one covering a specific use case—which is defined in terms of specific input and output models and how they should be related by the transformation. In this way, Tracts allow partitioning the full input space of the transformation into smaller, more focused behavioral units, and to define specific tests for them. Commonly, what developers are expected to do with Tracts is to identify the scenarios of interest (each one defined by a Tract) and check whether the transformation behaves as expected in these scenarios.

In a nutshell, a Tract defines a set of constraints on the *source* and *target* metamodels, a set of *source-target* constraints, and a *test suite*, i.e., a collection

Listing 2.2 Example of the JavaSpaces implementation of Linda for Java.

```java
/** Circumference class */
import net.jini.core.entry.*;
public class Circumference implements Entry {
    public int radius, x, y;
    public Circumference() {}
    public Circumference(int radius, int x, int y) {
    this.radius = radius; this.x = x; this.y = y; }
    public void setRadius(int r) { this.radius = r; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public String toString() {
        return("Circumference␣("+radius+",␣"+x+",␣"+y+")"); }
}

/** Main Program */
import JavaSpacesUtils.SpaceAccessor;
import net.jini.core.lease.*;
import net.jini.space.JavaSpace;
public class HelloWorld {
  public static void main(String[] args) {
      SpaceAccessor newSpaceAccessor = new
        SpaceAccessor("file␣containing␣details␣of␣the␣space␣to␣use");
      JavaSpace space = newSpaceAccessor.getSpace();

      Circumference c = new Circumference(3, 47, 53);
      space.write(c,null,Lease.FOREVER);

      Circumference c = new Circumference(6, 47, 70);
      space.write(c,null,Lease.FOREVER);

      Circumference template = new Circumference();
        template.setRadius(3);
      Circumference result =
            (Circumference)space.read(template,null,Long.MAX_VALUE);
      System.out.println(result);
  }
}
```

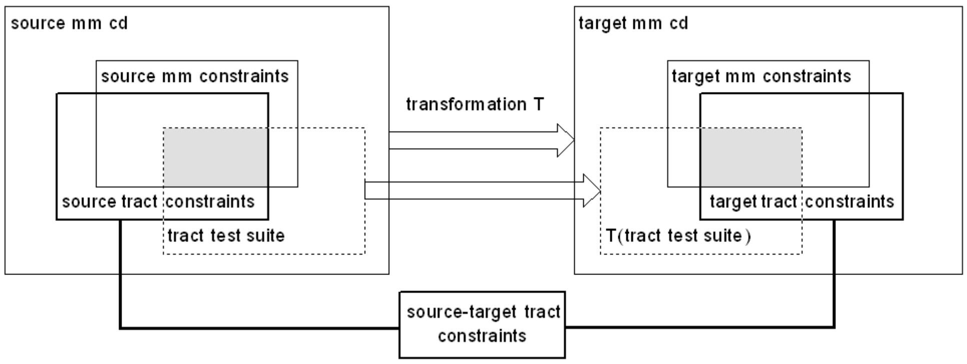


Fig. 2.3 Building Blocks of a Tract [56].

of source models. The constraints serve as "contracts" (in the sense of contract-based design [100]) for the transformation in some specific scenarios, and are expressed by means of OCL invariants. They provide the *specification* of the transformation. Test suite models are pre-defined input sets of different sorts, to exercise an implementation of the transformation (they may be not only be positive test models, satisfying the source constraints, but also negative test models, used to know how the transformation behaves with them).

Fig. 2.3 depicts the main components of the Tracts approach: the source and target metamodels, the transformation T under test, and the transformation contract, which consists of a Tract test suite and a set of Tract constraints. In total, five different kinds of constraints are present: the source and target models are restricted by general constraints added to the language definition, and the Tract imposes additional source, target, and source-target Tract constraints for a given transformation. In the drawing, `mm` stands for metamodel, and `cd` is a short for class diagram.

If we assume a source model M as being an element of the test suite and satisfying the source metamodel and the source Tract constraints given, the Tract essentially requires the result $T(M)$ of applying transformation T to satisfy the target metamodel and the target Tract constraints, and the tuple $< M, T(M) >$ to satisfy the source-target Tract constraints.

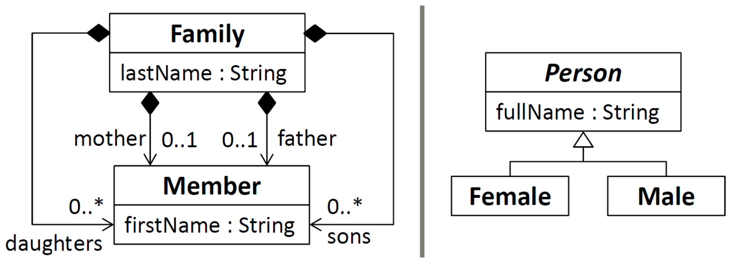


Fig. 2.4 The Family and Person metamodels.

To demonstrate how to use Tracts, we introduce the simple transformation example *Families2Persons* (the complete example is available on our project website [23]). The source and target metamodels of this transformation are shown in Fig. 2.4.

For this example, one Tract (Listing 2.3) is developed to consider only those families which have exactly four members (mother, father, daughter, son). The first constraint states that all families in the source model have exactly one daughter and one son. The second and third constraints state that all mothers and daughters are transformed into female persons. Constraint $C4$ mandates that all fathers and sons should be transformed into male persons. Constraints $C5$ and $C6$ state, respectively, that all female and male objects in the target model come from the corresponding object in the source model. Then, $C7$ checks that the size of the source and target models correspond. Finally, $C8$ checks that all names in the target model are neither the empty String nor undefined. Note that although some of the constraints could have been written using similar expressions (e.g., $C2$, $C3$, $C4$), we decided to express them using different styles for illustration purposes, and also to be able to differentiate them in our analyses.

Listing 2.3 Tracts for the *Families2Persons* case study.

```
-- C1: SRC_oneDaughterOneSon
Family.allInstances->forAll(f|f.daughters->size=1 and f.sons->size=1)

-- C2: SRC_TRG_Mother2Female
Family.allInstances->forAll(fam|Female.allInstances->exists(f |
    fam.mother.firstName.concat('␣').concat(fam.lastName)=f.fullName))
```

```

-- C3: SRC_TRG_Daughter2Female
Family.allInstances->forAll(fam|Female.allInstances->exists(f |
    fam.daughters->exists(d |
        d.firstName.concat('␣').concat(fam.lastName)=f.fullName)))

-- C4: SRC_TRG_FatherSon2Male
Family.allInstances->forAll(fam|Male.allInstances->exists(m |
    fam.father.firstName.concat('␣').concat(fam.lastName)=m.fullName
    xor fam.sons->exists(s |
        m.firstName.concat('␣').concat(fam.lastName)=s.fullName))

-- C5: SRC_TRG_Female2MotherDaughter
Female.allInstances->forAll(f|Family.allInstances->exists(fam |
    fam.mother.firstName.concat('␣').concat(fam.lastName)=f.fullName
    xor fam.daughters->exists(d |
        d.firstName.concat('␣').concat(fam.lastName)=f.fullName)))

-- C6: SRC_TRG_Male2FatherSon -- analogous to C5

-- C7: SRC_TRG_MemberSize_EQ_PersonSize
Member.allInstances->size=Person.allInstances->size

-- C8: TRG_PersonHasName
Person.allInstances->forAll(p|p.fullName <> '' and
    not p.fullName.oclIsUndefined())
```

Concerning the kinds of constraints defined, $C1$ represents a pre-condition for the transformation, $C2 - C7$ define constraints on the relationships between the source and target models, i.e., constraints that should be ensured by the transformation, and finally, $C8$ represents a post-condition for the transformation. Note that this approach is independent from the model transformation language and platform finally used to implement and execute the transformation.

2.3.2 Implementing Transformations with ATL

Given a specification of an MT, a *model transformation language* may be selected to implement the transformation. The ATLAS Transformation Language (ATL) [75] is a common choice. ATL was designed as a hybrid model transformation language containing a mixture of declarative and imperative constructs for defining uni-directional transformations. An ATL transformation is mainly composed by a set of *rules*. A rule describes how a subset of the target model should be generated from a subset of the source model. A rule consists of an *input* pattern (henceforth also referred to as left-hand side)—having an optional *filter* condition—which is matched on the source model and an *output* pattern (henceforth also referred to as right-hand side) which produces certain elements in the target model for each match of the input pattern. OCL expressions are used to calculate the values of target elements' features, in the so-called *bindings*. Given the metamodels in Fig. 2.4, a possible implementation in ATL is shown in Listing 2.4.

This implementation comprises two helper functions (whose definition is not shown in the listing) and two rules. One of the helpers is used to decide whether a member is female or not, and the second is used to compute the family name of a family member. Then, the first rule, $R1$, transforms male members (note the use of the helper `isFemale()` to filter the corresponding source objects) into male persons and computes their `fullName` attribute. Rule $R2$ is analogous, but for female family members.

For illustration purposes, we have chosen this simple example. Nevertheless, ATL allows the development of more complex transformations that require advanced characteristics such as the capability to create references in the target model and the use of the `resolveTemp` operation, which makes possible to point to any of the target model elements that will be generated from a given source model element. Model transformations with these characteristics have been implemented and are presented as part of our case studies in the following chapters.

2.3.3 Testing Transformations with Tracts

By running the transformation implementation for each model of the test suite and checking the target as well as the source-target constraints for the resulting input model and output model pairs, the validation of the transformation with respect to the constraints is achieved. The output of this validation phase is a test report documenting each constraint validation for the given input and

Listing 2.4 *Families2Persons* ATL Transformation.

```
module Families2Persons;
create OUT: Persons from IN: Families;

helper context Families!Member def:isFemale:Boolean=
  if not self.familyMother.oclIsUndefined() then
    true
  else
    if not self.familyDaughter.oclIsUndefined() then
      true
    else
      false
    endif
  endif;

helper context Families!Member def:familyName:String=
  if not self.familyFather.oclIsUndefined() then
    self.familyFather.lastName
  else
    if not self.familyMother.oclIsUndefined() then
      self.familyMother.lastName
    else
      if not self.familySon.oclIsUndefined() then
        self.familySon.lastName
      else
        self.familyDaughter.lastName
      endif
    endif
  endif;

rule Member2Male { -- R1
 from
  s: Families!Member (not s.isFemale)
 to
  t: Persons!Male(fullName<-s.firstName+'␣'+s.familyName)
}

rule Member2Female { -- R2
 from
  s: Families!Member (s.isFemale)
 to
  t: Persons!Female(fullName<-s.firstName+'␣'+s.familyName)
}
```

Listing 2.5 Test result for the *Families2Persons* example.

```
-----------------------------------------
-- Results for src_model001
-----------------------------------------
C1: SRC_oneDaughterOneSon: OK
...
C4: SRC_TRG_FatherSon2Male: KO
    Instances of src_model001 violating the constraint
    Set(Member001, Member002, ...)
...
```

output model pairs. An example report for the *Families2Persons* example for an input test model called `src_model001` produced by the *TractsTool* [26, 21] is shown in Listing 2.5. This model is composed of 1250 model elements (250 families, each one with one father, one mother, one son and one daughter), and was generated by an ASSL [54] procedure (cf. [26]).

In order to fix the transformation implementation to fulfil all constraints, the alignments between the transformation rules and the constraints are crucial in order to track the actual faults in the transformation rules from the observed constraint violations. While for the given example this may be achieved by just looking at the constraints and the rules (actually $R2$ misses the white space in the String concatenation), for larger examples automation support is essential due to the complexity of model transformations. Even in this example the alignment between the rules and the constraints is not trivial, and this is precisely where our proposed approach comes into play.

Chapter 3
Parallel Out-place Model Transformations

A wide range of different transformation languages already exists, each of them comprising different characteristics [121]. However, the increasing size and complexity of models are challenging the existing model transformations languages and engines, whose performance and scalability need to be significantly improved as the industry is progressively adopting model-driven techniques [83].

In fact, current model transformation engines are mostly based on sequential and in-memory execution strategies and thus they have limited capabilities to transform very large models in acceptable time.

This hinders the benefits of using models and model transformations in different application domains that use huge models, including biology, medicine and sociology.

At the same time, parallel computing has become increasingly important as chipmakers are putting more and more processor cores on individual chips—which are mainly wasted if sequential engines are used. Similarly, distributed algorithms are gaining attention as computer communications are getting much faster, cheaper and more reliable, and the Cloud is taking over.

In this chapter we present an approach to achieve parallel and distributed execution of transformations, providing the performance and scalability required to transform very large models in distributed environments. We introduce the LinTra approach and its Java implementation, jLinTra, which are based on the Linda [52] coordination language, and the use of data parallelism to achieve parallelization. LinTra offers concurrency and distribution mechanisms using the well known principles of separation of concerns [42], permitting concurrent access to distributed data in a transparent way. In LinTra, distribution is achieved using the *blackboard* [29] distributed shared memory approach, which also provides an abstraction over existing Java-based data space platforms. Scalability is addressed by using data management middleware platforms to implement the blackboards, which are able to deal with very large volumes of distributed data in an efficient way. Finally, the *master-slave* pattern [29] is used for achieving data parallelism.

The contribution of this chapter is fourfold. First, we present a novel Java-based execution platform called jLinTra for the parallel execution of out-place transformations that may also be used as a target for high-level transformation language compilers. Second, we provide a mapping of model transformation concepts into the LinTra framework. In particular, we define the representation of models and metamodels and how those models are stored over a set of machines using a blackboard approach. Third, we demonstrate the performance and scalability of this platform by reporting the results of running a model transformation test set using different Java middleware platforms for presenting models, and by comparing it against several state-of-the-art model transformation engines, including sequential and parallel ones. Finally, we discuss some implementation solutions for dealing with models that do not fit in memory or which are distributed over several machines, using highly distributed, scalable NoSQL databases [120] as underlying technologies.

The structure of this chapter is as follows. Section 3.1 introduces the LinTra framework, how model transformations are embedded in this framework and jLinTra's features for out-place transformations. Then Section 3.2 focuses on the execution of transformation chains where the output of a transformation is the input of the following. In Section 3.3 jLinTra is evaluated by using several case studies where we investigate the execution performance of LinTra with respect to different Java-based middleware platforms used to store and retrieve models, and we compare jLinTra with other execution engines. Finally, in Section 3.4 we discuss related work and Section 3.5 summarizes the chapter.

3.1 LinTra and its Java Implementation jLinTra

LinTra is a framework that allows the parallel execution of out-place model transformations, regardless of whether the models are located in a single machine or distributed over a set of nodes. We base our transformation approach on Linda [52], the mature coordination model for parallel processes that we introduced in Section 2.2.

Fig. 3.1 shows the architecture of the LinTra approach. For running transformations on such architecture, we explored how model transformations fit into the Linda framework and we made the distinction between two independent layers. The middleware layer contains the concrete Linda implementation, while the jLinTra layer on top of it comprises the model transformation written in Java and the models and metamodels representations. We also decided how trace links are encoded to allow for efficient retrieval, and how the transformation rule execution is distributed over the available computational resources (machines, cores, etc.).

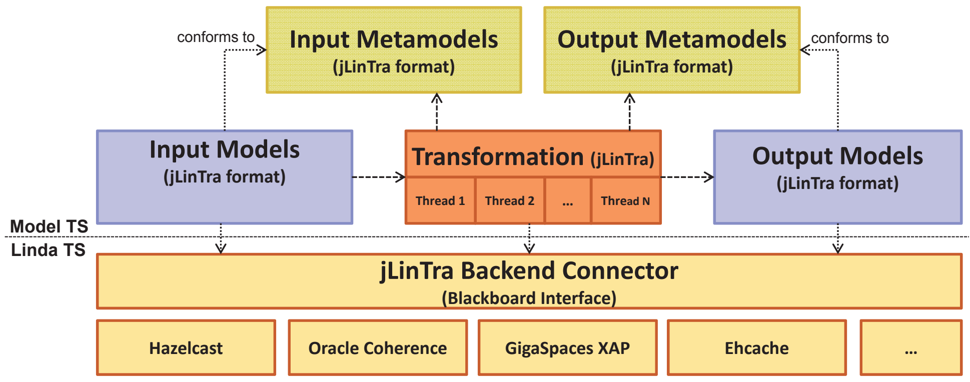


Fig. 3.1 LinTra architecture

3.1.1 Linda and Existing Implementations

There is a wide variety of pure Linda implementations written in different languages such as JavaSpaces [96] and TSpaces [91] in Java, C-Linda [4] in C, Rinda [127] in Ruby and PyLinda[1] in Python.

In addition, there are other mature software solutions for data management based on in-memory data grids (IMDG) or on distributed caches that are not used as Linda implementations but that provide similar functionality and even more. They are a specific kind of NoSQL databases called key-value caches. In particular, they (*i*) scale-out because every node (computer) adds its CPU and RAM to the cluster which can be used by all the nodes; (*ii*) can store big data and enable fast access to it as it is manipulated in main memory; (*iii*) permit dynamic scalability as nodes can dynamically join other nodes in a grid (cluster); (*iv*) enable elastic main memory as every node adds its own RAM memory to the cluster's memory pool; (*v*) implement fault-tolerance mechanisms without data loss, and (*vi*) implement a programming model to access the cluster as if it was a single machine. Some of these data management solutions are Hazelcast, Oracle Coherence, GigaSpaces XAP, Ehcache and Infinispan, to mention a few. In Section 3.3 we present a brief description for each particular solution we have worked with.

3.1.2 Building a Common Interface: The Blackboard Metaphor

According to Linda [52], the data storage is called *tuple space* (or *blackboard*). This tuple space can be thought of as a distributed shared memory that follows the Blackboard architecture pattern [29].

Different Linda implementations provide different interfaces to access the blackboard. To make the jLinTra model transformations independent from the concrete Linda implementation, we have defined an interface reusing the Linda primitives to read and write elements, adapting them to our needs. In particular, we use identifiers for referring to model elements, and thus we provide specific methods to read and write them using these identifiers.

[1]https://code.google.com/p/pylinda/

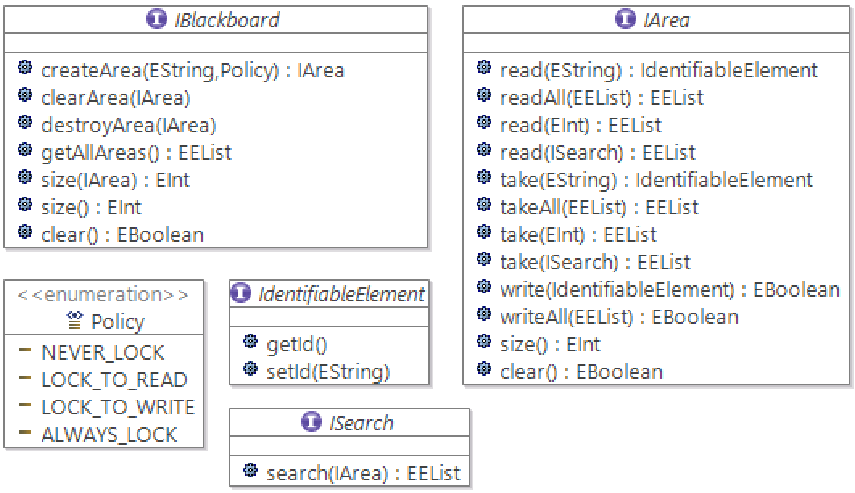


Fig. 3.2 LinTra interface.

We also permit partitioning the tuple space in *areas*. Finally, the interface provides methods to allow users to search for elements in the blackboard.

Fig. 3.2 shows the interfaces we have defined to access the blackboard. Following the Linda approach, the jLinTra implementation is not aware of how the distribution is done, nor the synchronization mechanisms needed for providing concurrency to the solution. Both concepts are transparent to the jLinTra model transformations, and the middleware layer takes care of them.

Focusing first on interface `IBlackboard`, we assume that the *blackboard* is composed of different areas (of type `IArea`) having each one an specific access policy. `LOCK_TO_READ` policy means that no more than one thread can access at the same time the area to read (or read and delete) an element, thus the thread accessing takes the token while the rest of the threads trying to read are blocked until the *token* is released. `LOCK_TO_WRITE` policy implies that at most one thread can access the area to write an element simultaneously. `ALWAYS_LOCK` combines the two previous policies whereas `NEVER_LOCK` means that all threads can freely access the area. These policies are internally managed by the LinTra platform, depending on the kind of transformation (e.g., regular or chained) and also to implement some internal processes, such as the assignment of identifiers—for which a private area in the blackboard is used. Users do not need to care about these policies.

Interface `IBlackboard` is shown in Listing 3.1. It offers methods to create, clear and destroy areas dynamically. It also offers the possibility to obtain a collection with the areas available with `getAllAreas()`. Method `size(IArea area)` returns the number of elements stored in the given area given and `size()` returns the number of elements stored in the blackboard (which is equivalent to the sum of the size of all the areas belonging to it). `clear()` deletes all the areas from the blackboard and their elements.

Listing 3.1 IBlackboard interface

```
public interface IBlackboard extends Serializable {
  public enum Policy{NEVER_LOCK, LOCK_TO_READ, LOCK_TO_WRITE,
      ↪ALWAYS_LOCK};
  public IArea createArea(String name, Policy p);
  public boolean clearArea(IArea area);
  public boolean destroyArea(IArea area);
  public Collection<IArea> getAllAreas();
  public int size();
  public int size(IArea area);
  public boolean clear();
}
```

In our approach, we consider that every element stored in the tuple space is an object with a unique identifier of type `String`, and thus, it must implement the interface `IdentifiableElement` shown in Listing 3.2.

Listing 3.2 IdentifiableElement interface

```
public interface IdentifiableElement extends Serializable {
  public String getId();
  public void setId(String id);
}
```

Regarding the interface `IArea` that is presented in Listing 3.3, its method `read(String id)` reads without deleting and returns the element with identifier `id` or null if the element does not exist in the area. Method `readAll(Collection<String> ids)` reads without deleting and returns the collection of elements whose identifiers are contained in `ids`, while method `read(int n)` reads `n` elements from the area and method `read(ISearch searchMethod)` receives as parameter a search method implementing interface `ISearch`—which requires to have a method called `search(IArea)`. This search method establishes the criteria for which elements are retrieved from the area. Equivalent to the *read* methods, the *take* methods have a similar behavior with the only difference that they delete the elements from the area. Methods `write(IdentifiableElement elem)` and `writeAll(Collection<IdentifiableElement> elems)` write the given elements into the area, `size()` returns the number of elements in the area and `clear()` removes all the elements stored in the area.

Listing 3.3 IArea interface

```java
public interface IArea extends Serializable {
  public IdentifiableElement read(String id);
  public Collection<IdentifiableElement> readAll(Collection<String> ids
      ↪);
  public Collection<IdentifiableElement> read(int n);
  public Collection<IdentifiableElement> read(ISearch searchMethod);
  public IdentifiableElement take(String id);
  public Collection<IdentifiableElement> takeAll(Collection<String> ids
      ↪);
  public Collection<IdentifiableElement> take(int n);
  public Collection<IdentifiableElement> take(ISearch searchMethod);
  public boolean write(IdentifiableElement elem);
  public boolean writeAll(Collection<IdentifiableElement> elems);
  public int size();
  public boolean clear();
}
```

To illustrate how the previously mentioned search method can be implemented and how it works, Listing 3.4 provides a possible implementation. Assuming that identifiers represent integers, it obtains the set of elements whose identifiers are in the range given by `min` and `max`. Another possible implementation of the search method could retrieve elements by type. This decision may have an impact on the performance. We recommend to keep the search method as simple as possible—i.e., avoid unnecessary accesses to the area and complex computations.

For clarity, in all the listings we have omitted that the methods throw BlackboardException when an Exception occurs.

Listing 3.4 Search method

```java
public class SearchByIdRange implements ISearch {
  int min, max;
  public SearchRange(int min, int max){
    this.min = min; this.max = max; }
  public Collection<IdentifiableElement> search(IArea area) {
    List<IdentifiableElement> elems =
                                new LinkedList<IdentifiableElement>();
    for (int i=min; i<=max; i++){
      IdentifiableElement e = area.read(Integer.parseInt(i));
      if (e!=null) { elems.add(e); }
    }
    return elems;
  }
}
```

3.1.3 Models and Metamodels in LinTra

In order to represent metamodels and models in Java so that they can be used by jLinTra, we need to identify the mappings between the metamodeling concepts and Java. In our approach we have worked with Eclipse Modeling Framework (EMF) models and thus we have built a bridge between Ecore (i.e., the metamodeling language of EMF) and jLinTra. Every class in an Ecore-based metamodel is mapped to a Java class that implements the `Serializable` and `IdentifiableElement` interfaces. Attributes belonging to the Ecore classes become Java fields, as well as the references that store the identifiers of the target element(s). As we shall later see, this is an important design decision in order to be able to write and execute transformation rules more independently than using explicit object pointers in Java. However, it also introduces additional challenges, e.g., when it comes to navigating between objects. Single inheritance is represented by Java inheritance and multiple inheritance is simulated with single inheritance and interface implementations. Java classes also need a constructor that receives as arguments the values of the attributes and references, and the getter and setter methods for all its fields. Models in jLinTra are composed by the set of Java objects that instantiate the Java classes. Note that, although we have implemented the bridge between Ecore and jLinTra, we do not provide support for all the features of EMF such as its operations (e.g. eContainer(), eContent(), etc.).

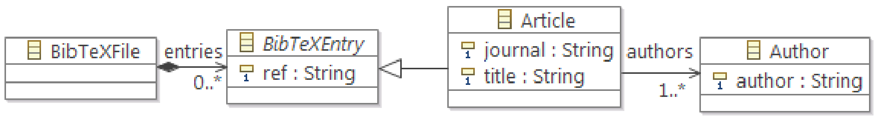


Fig. 3.3 `BibTeXML` metamodel excerpt.

Listing 3.5 Corresponding Java code for meta-class Article

```java
public class Article extends BibTeXEntry
                    implements IdentifiableElement,Serializable{
  private static final long serialVersionUID = 1L;
  String id, journal, title;
  String[] authorsIds;
  public Article(String id, String ref, String journal, String title,
          String[] authorsIds){
    super(ref);
    this.id = id;
      this.journal = journal;
      this.title = title;
      this.authorsIds = authorsIds;
  }
  public String getId() { return id; }
  public void setId(String id) { this.id = id; }
  ...
}
```

Let us use the case study `BibTeXML2DocBook` available at the ATL Zoo [61], in particular, the `BibTeXML` metamodel, to present how the generated Java code looks like. Fig. 3.3 shows an simplified excerpt of the original metamodel and Listing 3.5 presents a fragment of the generated jLinTra code for its class `Article`.

The bridge between EMF and jLinTra has been implemented in Java, and is able to work in parallel to optimize the load and translation of EMF models into their Java representation in memory, and viceversa.

3.1.4 Traceability

Traceability is normally needed when executing an out-place model transformation because the creation of an element might require information about some other elements previously transformed, or that will be transformed in the future [124]. This means that there might be dependencies that can affect the

	Input Id(s)	Output Id(s)
1..1	1	1_ruleName
1..N	1	{1_ruleName.1, 1_ruleName.2, 1_ruleName.3}
N..1	{1, 2, 3}	1-2-3_ruleName
N..N	{1, 2}	{1-2_ruleName.1, 1-2_ruleName.2, 1-2_ruleName.3}

Table 3.1 Example uses of trace function

execution performance, e.g., when one element needs to create a relationship to an element that has not yet been created.

LinTra does not store information about the traces explicitly. Traceability is implicitly implemented using a bidirectional function that receives as its parameter the object identifier (or all the object identifiers in case that the match comprises more than one element) in the source model, and returns the identifier of the target element(s)—regardless of whether the target elements themselves have already been created or not. Being a bidirectional function, it can be applied in the opposite direction to obtain the source identifiers of any target element.

Elements have identifiers, of type *String*. In the source model, these identifiers correspond to sequential integers. For creating the identifiers of target elements, we need to distinguish four different cases depending on the cardinality of the relation they have with corresponding source elements. Cardinality 1..1 is the simplest one; in this case every element is transformed into another element, keeping its identifier followed by the name of the rule that transforms it. We need to attach the rule name as the matches might not be exclusive. In the case of $1..N$ cardinality, more than one output element is generated for every input element, and thus the integers specifying the order in which the elements are created are attached to the output elements' identifiers. The opposite case is $N..1$, where all the input identifiers are attached at the beginning separated by dashes $(-)$. Finally, the $N..N$ case is the combination of $1..N$ and $N..1$ situations. For illustration purposes, Table 3.1 shows an example of some input identifiers and their corresponding output identifiers.

In jLinTra, there is a class in charge of creating and resolving identifiers. By way of illustration, the piece of code corresponding to the cardinalities $1..N$ and $1..N$ is presented in Listing 3.6.

The `create` method is called by a rule at the same moment that it is creating a target element in order to obtain its identifier. For instance, let us assume that the rule "A-to-B" is creating a single target element from an element whose identifier is 5, the identifier of the target element is given by `TraceFunction.create(5, "A-to-B")`. On the other hand, the `resolve` method is called when a target element has to reference another target element. As we mentioned in the previous subsection, references are represented by storing the identifiers of the referenced elements. The `resolve` method provides the identifier of the referenced element given the identifier of the source element from which it is created and the name of the rule that transforms it. Let us assume that a new target element is being created and it has to reference the target element that is transformed by the rule "C-to-D" from a source element with identifier 20. That new target element references the element whose identifier is given by `TraceFunction.resolve(20, "C-to-D")`. Note that the target element with the identifier given by the `resolve` method may or may not have been created yet.

Listing 3.6 TraceFunction class

```java
public class TraceFunction {
    // Methods for cardinality 1..1
  private static String f(String id, String ruleName) {
    return id+"_"+ ruleName;
  }
  public static String create(String id, String ruleName) {
    return f(id, ruleName);
  }
  public static String resolve(String id, String ruleName) {
    return f(id, ruleName);
  }
    // Methods for cardinality 1..N
  private static String f(String id, int pos, String ruleName) {
    return id+"_"+ ruleName + "." + pos;
  }
  public static String create(String id, int pos, String ruleName) {
    return f(id, pos, ruleName);
  }
  public static String resolve(String id, int pos, String ruleName) {
    return f(id, pos, ruleName);
  }
    ...
}
```

3.1.5 Master-Slave Configuration

To run the model transformation, jLinTra uses the Master-Slave design pattern [29]. The blackboard is also used by the master and the slaves to communicate with each other.

The master partitions the input model and creates what we call *jobs*, which are subsets of the input model. Those jobs are written in the shared memory, in an area called `TODO`. Slaves access that area to get jobs. Given the way in which the references are represented and the nature of the traceability function, we do not need to take into account how the models are partitioned or distributed.

The size of the jobs is critical for establishing the model partitioning. There are several tradeoffs that must be considered when deciding the size of the jobs.

- Given that our models are not currently preprocessed, the master does not know how models are organized. Then, the bigger the jobs are, the more homogeneous the submodels will be (with respect to the type of elements that they contain), and therefore the more homogeneous the work of the slaves will be.
- Access to the blackboard is normally costly, and then bigger jobs means less accesses.
- The memory available for each thread is limited, and thus care must be taken so that jobs fit into that memory.
- Smaller jobs permit better distribution of the tasks among the available machines and slaves.

These decisions can have a significant impact on the performance of the transformation. Given that there is no solution that suits all purposes, in jLinTra the size of the jobs is a parameter given to the transformation in a configuration file. In this way, users can experiment with different options to find which is the optimal one for their transformation. Nevertheless, in the future we plan to investigate how the size of the jobs can be optimized for each particular case making use of the techniques used in distributed systems.

The code that the slaves execute is shown in Listing 3.7. Every slave reads *jobs* from the `TODO` area (lines 6 and 7) and for each job, it searches in the corresponding area for the model elements included in the job (line 8) and applies the transformation to them, e.g., the one shown in Listing 4.3.

Listing 3.7 Slave Implementation

```
 1 public class Slave implements Runnable, ISlave {
 2     //Definition of variables
 3     int threadId;
 4     ...
 5     public Slave(int threadId, ITransformation transfoImpl,
 6         IArea todoArea, IArea outArea) { ... }
 7     public void run() {
 8         Job job = askForWork();
 9         while (job != null) {
10           Collection<IdentifiableElement> inElems = job.getArea()
11             .read(new SearchByIdRange(job.getMinID(),job.getMaxID()));
12         Collection<IdentifiableElement> out = transfoImpl
13             .transform(inElems);
14         outputArea.writeAll(out);
15         job = askForWork();
16       }
17     }
18 }
```

An additional benefit of this architecture is that it permits process distribution: the slaves can be executed in different machines. In this way, it can maximize the use of available processing resources in a flexible and scalable manner.

3.1.6 jLinTra Transformation Definitions By-Example

Let us use again the `BibTeXML2DocBook` case study to illustrate how transformations are written in jLinTra, and in particular a simplified version (shown in Listing 3.8) where every `Article` is transformed into a `Section`, every `Author` is transformed into a `Paragraph`, and every `Section` created from an `Article` is linked to the corresponding `Paragraphs` created from the `Authors` of that `Article`.

Listing 3.8 Example transformation excerpt

```
 1 module BibTeX2DocBook;
 2 create OUT : DocBook from IN : BibTeX;
 3 rule article2section {
 4   from
 5     a : BibTeX!Article
 6   to
 7     s : DocBook!Section (
 8       title <- a.title,
 9       paras <- a.authors
10     )
11 }
12 rule author2paragraph {
13   from
14     a : BibTeX!Author
15   to
16     p : DocBook!Para (
17       content <- a.author
18     )
19 }
```

As Listing 4.3 shows, transformations written in jLinTra consist of iterating over the elements received as parameter (line 4) and for every element, the corresponding filters are applied in the form of "`if`" instructions (lines 5 and 10), to check if the element needs to be transformed, and how. In case they are matched, the appropriate output elements are created. Note that this is a very simple example where there are only two 1..1 rules. Further examples are available from our project website [27] also involving more complex rules.

Listing 3.9 Example transformation excerpt

```
1 public class BibTeXML2DocBook implements ITransformation {
2     public Collection<IdentifiableElement> transform(
            Collection<IdentifiableElement> elems) {
      List<IdentifiableElement> out =
            new LinkedList<IdentifiableElement>();
      for (IdentifiableElement e : elems) {
        if (e instanceof Article) {
          Article a = (Article) e; out.add(article2Section(a));
        }
        else if (e instanceof Author){
          Author a = (Author) e;  out.add(author2Paragraph(a));
        }
      }
      return out;
    }
    private Section article2Section(Article a){
     return new Section(
      TraceFuntion.create(a.getId(),Rules.Art2Sec), a.getTitle(),
      TraceFunction.resolveAll(a.getAuthorsIds(),RuleNames.Auth2Par));
    }
    private Paragraph author2Paragraph(Author a) {
     return new Paragraph(
      TraceFunction.create(a.getId(),Rules.Auth2Par),a.getAuthor());
    }
}
```

Note as well the use of the `TraceFunction` class not only to store the traces but also to resolve the references to other elements regardless of whether they have already been transformed or not. This is how relationships between transformation rules are naturally managed in our approach.

3.1.7 Distributed Models

One of the benefits of the Linda approach lies on its independence from data size and distribution, given that it uses a shared-memory architecture. Such separation of concerns is also key in LinTra, which then permits dealing with model storage and distribution in an independent manner. In fact, our architecture clearly separates those aspects (see Fig. 3.1).

We have studied different technological solutions for implementing the data management layer. In the first case we have one in-memory solution, when the models fit into the computer memory, and we want just to use the parallel features of jLinTra. It uses the Java `HashMap` collection type to implement the Tuple spaces (i.e., the *blackboard*) and all its areas. This is the technological solution we have used to compare the performance of jLinTra with existing model transformation engines (ATL, QVT-O, ETL, etc.) since all of them only support in-memory implementations.

Our approach also supports dealing with models that do not fit in memory, or that are distributed over several machines, using in-memory data grids that can be connected to distributed, scalable NoSQL databases [120] as underlying technologies. Examples of NoSQL databases include Cassandra, neo4j, FoundationDB and MongoDB. These new database technologies achieve scalability through horizontally distributing data, and replace normalized data models, strong data consistency guarantees, and SQL queries with schema-less data models, weak consistency guarantees, and proprietary APIs.

We tested five different commercial solutions for key-value in-memory data grids and/or caches that permit distribution and connection to NoSQL databases:

- **Oracle Coherence**[2] is an in-memory data grid from Oracle.
- **Hazelcast**[3] is an in-memory open source data grid based on Java.
- **Ehcache**[4] is an open source distributed cache.
- **GigaSpaces XAP** (eXtreme Application Platform)[5] is an in-memory computing software platform provided by GigaSpaces.
- **Infinispan**[6] is a open source data grid platform and key-value data store.

Table 3.2 shows the results of running the same jLinTra transformation on different Java-based data management solutions for several models of increasing size. The number in the leftmost column indicates the number of

[2]http://www.oracle.com/technetwork/middleware/coherence/overview/index.html
[3]http://hazelcast.com/
[4]http://ehcache.org/
[5]http://www.gigaspaces.com/xap-in-memory-computing-event-processing/Meet-XAP
[6]http://infinispan.org/

No. elements	**HashMap**	**Coherence**	**Hazelcast**	**Ehcache**	**XAP**	**Infinispan**
0.1×10^6	0, 138	2, 133	19, 654	0, 299	8, 757	0,189
0.2×10^6	0, 138	2, 971	39, 335	0, 424	16, 542	0,313
0.5×10^6	0, 385	8, 298	99, 740	0, 276	38, 796	0,877
1.0×10^6	0, 969	16, 164	300, 105	0, 795	78, 810	1,688
1.5×10^6	1, 732	26, 701	451, 701	1, 817	121, 527	3,094
2.0×10^6	3, 034	35, 561	590, 760	4, 431	159, 862	5,353
2.5×10^6	5, 105	44, 142	724, 658	11, 811	177, 273	6,536
3.0×10^6	6, 990	56, 144	870, 705	14, 280	–	9,527
3.5×10^6	7, 975	75, 321	1016, 626	20, 329	–	13,202

Table 3.2 Data management middleware comparison

model elements in the input model. Since we are only interested in how the read and write operations perform, we have applied the identity transformation (more precisely, we used the IMDb Movie Database transformation presented in Section 3.3.2). All execution times are shown in seconds. Cells with a dash "–" mean that the model cannot be transformed due to a memory allocation problem.

Among them, Infinispan was the one that offered more features. In addition, its integration with the LevelDB database[7] was easy and provided us with all the functionality we required for implementing persistence (i.e., disk storage) and distribution for large models. Although a detailed performance comparison between the different NoSQL databases is out of scope for this work and therefore left for future work, our initial experiments show that the key-value data stores are the solutions which perform best and they all present similar performance. Hence, in this thesis we have used Infinispan (with LevelDB as persistent database) to implement the blackboard layer of jLinTra in case an in-memory solution was not enough to store and transform models. Otherwise, the Java HashMap implementation of the blackboard is used.

[7]http://leveldb.org/

3.2 Model Transformation Chains

As more complex problems are tackled in industry, the number of transformations involved in MDE solutions has increased. In fact, most real-world MDE scenarios do not involve a single transformation from one source model to one target model, but multiple model transformations organized in chains, with the output of some transformations serving as input to others [146]. Thus, smaller transformations can focus on specific concerns, are easier to develop and maintain, and together constitute a more modular, extensible and maintainable architecture. Integrating them into chains is no longer an issue, with domain specific languages for specifying and executing model transformation chains [118]. The number of transformations in a chain depends on the domain and in the particular application, but they can normally range between 5 and 12 in industrial projects [48]. Hence the importance of considering the parallel execution of transformation chains.

Given the architecture of our platform, implementing the parallel execution of chained transformations is rather natural. It was a matter of extending our approach with (*i*) synchronization mechanisms between the transformations, and (*ii*) pairs of element identifiers.

Synchronization between the different transformations is naturally implemented by the use of the master-slave pattern and by the way in which jobs are assigned to slaves. Thus, the slaves in charge of implementing the second transformation will wait until they have jobs to do. These will be generated by the master of the second transformation as soon as the output elements of the first transformation are produced. Unlike regular model transformations—where a complete source model is available at the beginning of the transformation—transformation chains involve *streaming* models [37]—i.e., those whose elements are not all present in disk or memory, but rather arrive as one or more continuous data streams—that cause dependencies when a rule needs to access an element that is not available yet. jLinTra uses the synchronization mechanisms that Java provides. When a slave finds a dependency, it invokes the wait() method and all the resources are released for the use of other slaves until the master invokes the notify() method informing that there are new elements are available in the model. The master and slaves of a transformation know when the model has been totally loaded because an EOF flag indicates when all the elements are already available.

Regarding the pairs of element identifiers, we previously mentioned that model elements have unique identifiers, which were used to implement the traces in an efficient manner by means of a bidirectional function. For that, the identifiers of the output elements had a special form too (see, e.g., Table 3.1). In order for these elements to become the input elements of another transformation, while still maintaining the tracing information, we need to assign them other identifiers, which allow them to act as source elements of the next transformation. Thus, our transformations always generate two identifiers for all output elements. The one explained in Section 3.1.4 plus a new one representing an integer which is the one used in case the target model needs to be used as source of another transformation. A Hashtable is also generated with the two identifiers, in order to optimize the search for elements using their first identifier. Such a table is also stored in the blackboard, as another artifact of the transformation itself.

3.3 Evaluation and Performance Analysis

In this section, we discuss the performance and scalability of jLinTra by performing a set of case studies [90] based on a set of exemplar transformations. The discussion follows the guidelines for conducting empirical explanatory case studies by Roneson and Hörst [119]. Detailed information on the metamodels and input models used in these examples (number of elements, file size on hard disk, etc.), and on the transformations themselves, is available from our project's website.

3.3.1 Research Questions

We have defined two research questions that compare the performance and scalability of jLinTra with respect to state-of-the-art sequential transformation engines and emerging parallel transformation engines, one about the parallel execution of transformation chains, and a final one about the effect on the performance when models are stored in disk, and not in memory. More specifically, we aimed at answering the following research questions (RQs):

RQ1. *How does jLinTra perform compared to existing sequential execution engines?* One main goal of jLinTra is to improve the performance and scalability of current sequential execution engines. Thus, we evaluate the achieved speedup compared to such approaches.

RQ2. *How does jLinTra perform compared to other emerging parallel execution engines?* jLinTra is also compared against other existing parallel execution engines w.r.t. performance and scalability.

RQ3. *How does jLinTra model transformation chains perform?* The performance of running a model transformation chain in jLinTra is compared to the performance of running the same transformation sequentially one after the other.

RQ4. *How is the performance of jLinTra affected when models do not reside in memory?* jLinTra provides an abstraction from data management middleware solutions, permitting transparent access to data independently from where it resides (in-memory, on-disk, distributed). It is important to evaluate the costs of dealing with models that do not reside in memory (because of their size or their origin) and how this affects the performance of jLinTra model transformations in terms of speed degradation, scalability, etc.

3.3.2 Case Studies

This section describes five examples that have been used to evaluate jLinTra and compare it with other model transformation languages and engines. These examples were chosen to capture different relevant features of model matching, navigation and element traceability involved in most commonly used model transformations.

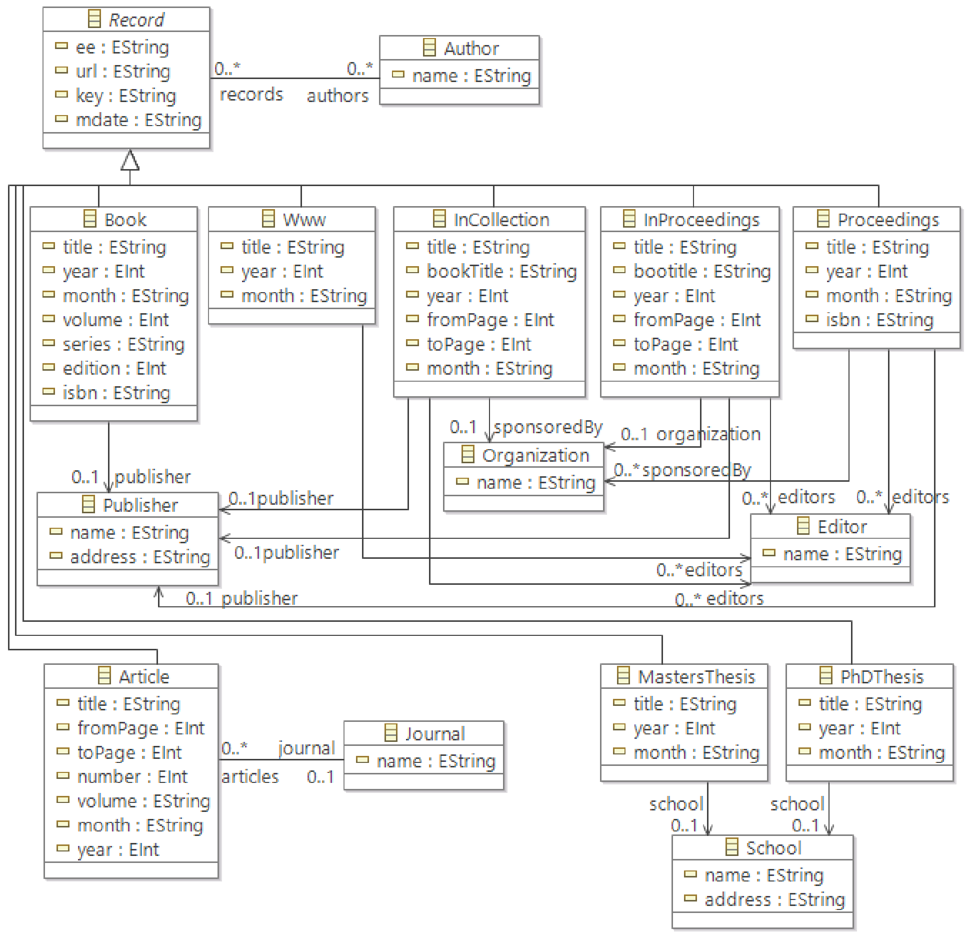


Fig. 3.4 DBLP Metamodel.

DBLP—Model queries

The first example uses the complete DBLP database[8] as source model. It has 5, 654, 916 elements when stored as a model. Its metamodel is shown in Fig. 3.4. This is an example of model queries over a large model. This case study defines four different transformations, covering four types of queries which exercise different accesses to model elements. Those transformations

[8]http://dblp.uni-trier.de/xml/

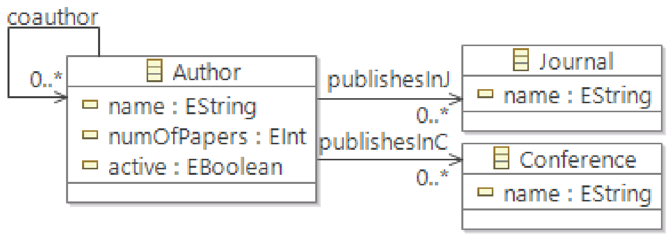


Fig. 3.5 AuthorInfo Metamodel.

extract information using the `AuthorInfo` metamodel as transformation target (Fig. 3.5).

- Find all the authors that have published at the International Conference on Model Transformation (ICMT) conference and their number of papers.
- Find if those ICMT authors are still publishing (active) or if they are inactive (*active* means that they have published in the last 5 years).
- Find the conferences where people who stopped publishing at ICMT are now publishing.
- Find all journals where people who are actively publishing at the Information & Software Technology (IST) journal (i.e., have published something in the last 10 years) are also publishing.

IMDb Movie Database—Model copy and traversal

The second example uses the "Movie Database" (IMDb) proposed in the Transformation Tool Contest (TTC) 2014 [70], whose metamodel is shown in Fig. 3.6.

The first transformation is the identity, which checks how fast the complete model graph can be traversed and copied. The second one copies all the elements in the input model (movies, actors and actresses—3.5 million elements) and finds all pairs of people who played together in at least in three movies. This second transformation involves navigating the source elements

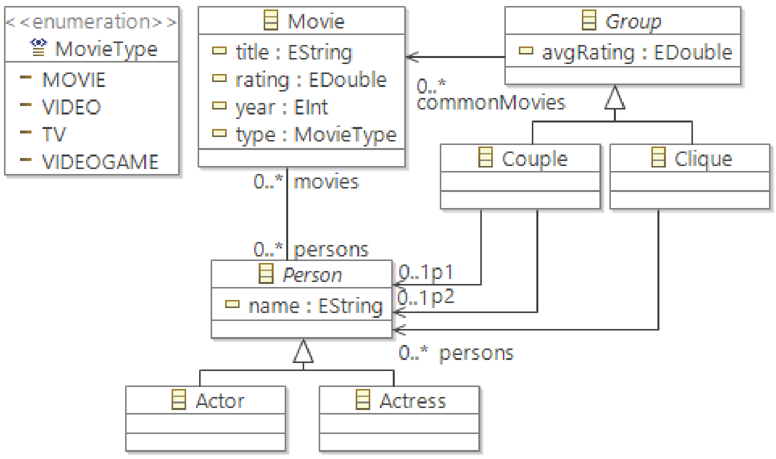


Fig. 3.6 IMDb Metamodel.

before transforming the elements (this model transformation uses same source and target metamodels).

In this case we run the transformations over a set of 9 different models, emulating different sizes of the database model to check how different model transformations engines scale up (from 100, 000 elements to the complete model with 3.5 million elements).

Java Refactoring—Model modification

This case study is taken from the 2015 edition of TTC[9]. This is an example of program transformation rules for code refactoring, where all the `@Singleton` annotations are removed from Java programs and their implicit behavior is replaced with the actual Java code they represent. More precisely, given an annotated Java program, all classes annotated with the `@Singleton` keyword must be modified as follows: the annotation is removed; all constructors are set to private; a public and static variable named `instance` whose type coincides with the class type is created; and a `getInstance` method is created for each constructor that initializes the variable `instance` in case it was not already initialized, and then returns it. Each `getInstance` method has the

[9]http://www.transformation-tool-contest.eu/solutions_refactoring.html

Listing 3.10 Code to be refactored

```
@Singleton
public class ContextDataFilter extends ViewerFilter {
   private String pattern;
   public ContextDataFilter(String pattern){
      this.pattern = pattern;
   }
}
```

Listing 3.11 Refactored code

```
public class ContextDataFilter extends ViewerFilter {
   private static ContextDataFilter instance;
   private String pattern;
   private ContextDataFilter(String pattern){
      this.pattern = pattern;
   }
   public ContextDataFilter getInstance(String pattern){
      if (instance==null){
         instance = new ContextDataFilter(pattern);
      }
      return instance;
   }
}
```

same parameters as the corresponding constructor. This is an example where strong dependencies between the transformation rules exist.

An illustrated example of a Java class annotated is shown in Listing 3.10 while Listing 3.11 shows the code after having applied the transformation.

The input models are obtained from Java code using MoDISCO [19]. The Java metamodel has a total of 125 classes from which 15 are abstract, 166 relationships among them and 5 enumeration types. As source model we have selected the complete Eclipse project, containing $4,357,774$ entities. In order to assess how the transformation scales up with this kind of input, we generated 11 smaller sample source models (with subsets of the Eclipse project) ranging from $100,000$ elements to the complete model.

Java to Prefuse Graph—Model transformations

This case study is taken from the model visualization domain. Tools for the analysis of large models that use visualization techniques require efficient

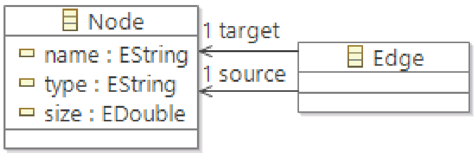


Fig. 3.7 Prefuse Graph Metamodel.

mechanisms for transforming models, since a change in a model normally means having to rework all artifacts.

In our case studies, starting from a model of a large Java program, we transform it into a graph of the Prefuse visualization tool[10], where every class is represented by a `Node` and every Java attribute whose type is another class as an `Edge`. The size of every node depends on the number of outgoing edges, and the color depends on the type of Java class: red if the class is abstract, blue if it is final, and green if it is a regular class. The target metamodel is shown in Fig. 3.7.

For the source models, we have used once again the source models and submodels from the complete Eclipse project. The transformation produces $26,426$ elements.

Java to Prefuse Graph and then to Reduced Graph—Model transformation chains

Apart from model-to-model transformations, we wanted to explore the use of parallel techniques in model transformation chains, as stated in RQ3. For that we will use a dedicated example. It uses the previous Java-to-Graph transformation as the initial transformation of a chain that reduces the original graph in order to become easier to understand and to manipulate by final users. Normally, the resulting graph after applying the Java-to-Graph transformation is still too large, and in this example a second transformation (Graph-to-ReducedGraph) is applied to the resulting model that reduces the graph. It does so by removing the nodes that have five or less outgoing edges. The corresponding edges are also removed so that no dangling edge exists in the

[10]http://prefuse.org/

final graph. For instance, the graph given for whole Eclipse project has $26,426$ elements and this second transformation reduces it to $2,936$ elements. We will refer as Java-to-Graph-to-ReducedGraph to the chained transformation composed of the two individual transformations: Java-to-Graph and then Graph-to-ReducedGraph. We will discuss the nature of this composition (it could be either sequential or in parallel) later in Section 3.3.4.

3.3.3 Setup

Transformation languages. We have selected several transformation languages to compare against jLinTra: QVT-O, ATL, parallel ATL (p-ATL), ATL and ETL. First, QVT-O [OMG] is an imperative language with a similar syntax to Java. We used the QVT-O execution engine[11] currently supported in Eclipse. ATL [75] is a hybrid language containing a mixture of declarative and imperative constructs. ATL supports three different execution engines: the sequential ATL engine provided as part of the standard distribution package (ATL); a sequential ATL EMF Transformation Virtual Machine with advanced features, called EMFTVM [147] (ATL-VM); and a parallel ATL engine [137] in which the parallelization is automatically performed by the transformation engine. Finally, the Epsilon Transformation Language (ETL) is a hybrid model transformation language developed atop the Epsilon model management infrastructure [82].
Measurement method. For all the approaches we have only taken into account the time of the transformation execution, i.e., we do not consider the time used for loading the models into memory or the time taken to serialize them to the disk. We launched the transformations using a Java program that invokes the model transformations using the APIs provided by the evaluated languages for running transformations programmatically, and registered the computation times using the `System.currentTimeMillis()` Java method.

The execution results shown in the following tables are computed as the average value after executing each transformation 20 times for every input model, having discarded the first 5 executions as the VM has a warm-up phase,

[11]https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

where the results are not optimal (it is worth mentioning that all the engines need a warm-up phase, which means that the first transformation always takes longer than the following ones; for this reason, we first warmed up the Java virtual machine by running another transformation whose result was discarded, and only considered the results after the warm-up phase). Outliers were also considered. In fact, during the measurement phase to compute the resulting values we identified a very small number of outliers (less than .05% in total). These executions were discarded for computing the average values.
Measurement environment. We executed all the transformations on a machine whose operating system is Ubuntu 12.04 64 bits with 11.7 Gb of RAM and 2 processors with 4 hyperthreaded cores (8 threads) of 2.67GHz each. The Eclipse version is Luna. The Java version is 8, where the JVM memory as been increased with the parameter *-Xmx11000m* in order to be able to allocate larger models in memory. The version of QVT-O we have used is 3.4.0 and for ATL it is 3.6.0. There are more than one virtual machines offered for ATL, among which we have selected the default one and EMFTVM in its version 3.6.0. Regarding the p-ATL, we have used the only available prototype[12]. The version of ETL is 1.2.0.

3.3.4 Results

Results concerning RQ1 and RQ2.

Tables 3.3 to 3.7 show the results obtained for the different transformation runs, answering RQ1 and RQ2 (examples 3.3.2 to 3.3.2). Execution times are shown again in seconds. Cells marked with dashes "−" indicate that both the input and output models did not fit into memory.

Figures 3.8-3.12 shows the graphical representation of the times presented in Tables 3.3 to 3.7. The X-axis represents the number of model elements expressed in millions (except in the first chart), and the Y-axis represents the time taken by the transformation in seconds. Paying individual attention to each case study, we can see that jLinTra is the engine that achieves the

[12]http://web.emn.fr/x-info/atlanmod/index.php?title=Parallel_ATL

	DBLP					
	ATL	**ATL-VM**	**QVT-O**	**ETL**	**p-ATL**	**jLinTra**
DBLPv1	14,304	25,821	78,891	5148,403	13,312	11,004
DBLPv2	15,129	36,974	90,014	5165,910	14,191	11,060
DBLPv3	30,585	38,696	44,403	5177,700	30,967	12,147
DBLPv4	30,885	27,187	44,359	5195,066	29,645	17,691

Table 3.3 Results for the DBLP case study.

	IMDb-Identity					
No. elements	**ATL**	**ATL-VM**	**QVT-O**	**ETL**	**p-ATL**	**jLinTra**
0.1$\times 10^6$	1,707	0,884	2,922	1,047	1,478	0,138
0.2$\times 10^6$	3,559	1,880	5,645	2,145	3,114	0,138
0.5$\times 10^6$	9,557	5,158	14,806	5,852	8,756	0,385
1.0$\times 10^6$	22,036	11,009	33,535	14,465	21,019	0,969
1.5$\times 10^6$	35,026	18,471	52,796	27,438	34,414	1,732
2.0$\times 10^6$	46,957	27,985	71,044	36,965	48,917	3,034
2.5$\times 10^6$	61,586	36,274	92,435	48,332	63,871	5,105
3.0$\times 10^6$	75,968	47,496	125,451	71,360	76,967	6,990
Complete	94,765	73,858	170,280	78,825	89,487	7,975

Table 3.4 Results for the IMDb-Identity transformation.

	IMDb-FindCouples					
No. elements	**ATL**	**ATL-VM**	**QVT-O**	**ETL**	**p-ATL**	**jLinTra**
0.1$\times 10^6$	21,257	16,086	198,822	44,346	21,949	1,149
0.2$\times 10^6$	38,818	27,407	345,627	74,958	39,801	2,092
0.5$\times 10^6$	103,374	91,268	1406,258	24,284	113,283	6,534
1.0$\times 10^6$	288,549	377,605	8071,292	225,684	290,359	17,225
1.5$\times 10^6$	541,643	960,045	27077,018	1488,706	560,301	33,115
2.0$\times 10^6$	952,579	1925,698	–	2240,208	997,052	51,811
2.5$\times 10^6$	1496,316	4026,594	–	3606,062	1476,207	76,139
3.0$\times 10^6$	2373,767	7727,816	–	5433,614	2261,381	122,971
Complete	3522,418	10674,105	–	7293,901	3240,343	192,316

Table 3.5 Results for the IMDb-FindCouples transformation.

lowest execution time in the DBLP case study, followed by p-ATL and ATL. The execution times are similar in this case because parallel ATL follows a rule-based design which means that every rule is executed in a thread. Although the input models are large, transformations are very small, they

	Java Refactoring					
No. elements	**ATL**	**ATL-VM**	**QVT-O**	**ETL**	**p-ATL**	**jLinTra**
0.1$\times 10^6$	14,912	14,415	3,26	16,047	1,793	0,163
0.2$\times 10^6$	45,767	36,651	6,56	35,872	3,869	0,28
0.5$\times 10^6$	240,75	188,771	16,823	113,607	15,487	0,771
1.0$\times 10^6$	947,133	710,489	32,635	318,412	21,133	2,422
1.5$\times 10^6$	2088,758	1552,695	50,086	606,131	31,209	3,911
2.0$\times 10^6$	3826,694	2744,669	74,283	949,207	38,118	4,934
2.5$\times 10^6$	6492,396	4498,775	90,798	2386,584	50,036	5,695
3.0$\times 10^6$	12385,465	11563,646	–	3654,23	67,245	11,786
3.5$\times 10^6$	16382,859	13802,342	–	4742,888	80,386	12,361
4.0$\times 10^6$	16836,713	14414,278	–	4844,773	82,367	14,233
Complete	20148,078	18501,348	–	5792,249	93,441	15,748

Table 3.6 Results for the Java Refactoring transformation.

	Java-to-Graph					
No. elements	**ATL**	**ATL-VM**	**QVT-O**	**ETL**	**p-ATL**	**jLinTra**
0.1$\times 10^6$	0,099	0,134	0,280	0,334	0,065	0,064
0.2$\times 10^6$	0,198	0,215	0,522	0,748	0,124	0,134
0.5$\times 10^6$	0,508	0,408	1,273	2,025	0,295	0,344
1.0$\times 10^6$	0,909	0,705	2,572	4,045	0,580	0,701
1.5$\times 10^6$	1,361	1,006	3,406	6,001	0,805	1,037
2.0$\times 10^6$	1,829	1,309	4,926	8,498	1,057	1,456
2.5$\times 10^6$	2,239	1,601	6,004	10,869	1,394	1,799
3.0$\times 10^6$	2,830	2,449	7,751	11,710	1,686	2,049
3.5$\times 10^6$	3,245	2,733	8,724	14,890	1,881	2,293
4.0$\times 10^6$	3,284	3,211	9,919	16,130	1,933	2,74
Complete	3,720	2,660	11,711	20,835	2,201	3,474

Table 3.7 Results for the Java-to-Graph transformation.

have only one to two rules and thus, the number of processors available does not matter because most of them are simply not used.

jLinTra is also the one that executes the transformation faster in the case of the IMDb identity transformation. The second best performance corresponds to ATL-VM, closely followed by ETL and by ATL and p-ATL. Finally, QVT-O has the worst performance.

In the IMDb FindCouples transformation, jLinTra and QVT-O have again the best and the worst performance, respectively, while p-ATL and ATL are

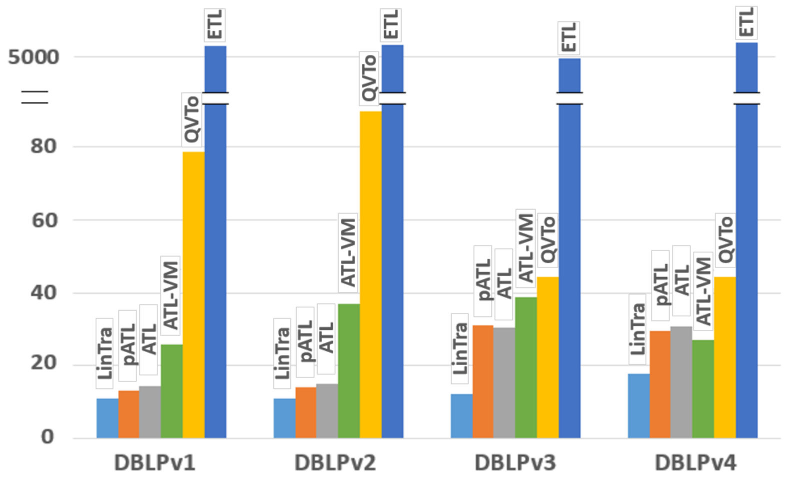


Fig. 3.8 Comparative chart for the DBLP case study.

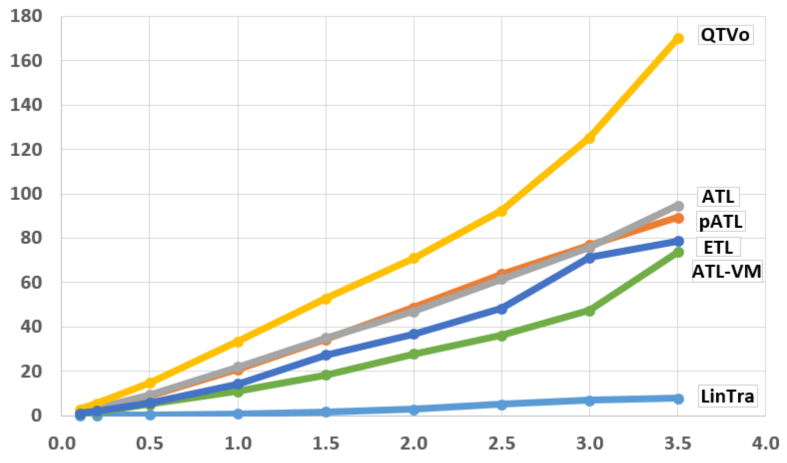


Fig. 3.9 Comparative chart for the IMDb-Identity case study.

followed by ETL and ATL-VM. Although this transformation has more rules than the DBLP transformation, the behavior of ATL and its parallel version is similar again because given the nature of the case study and the input models, one rule needs much more time than the rest of them — which means that most of the time only one core is actively used. Thus, the execution shows a sequential behavior from certain moment onwards.

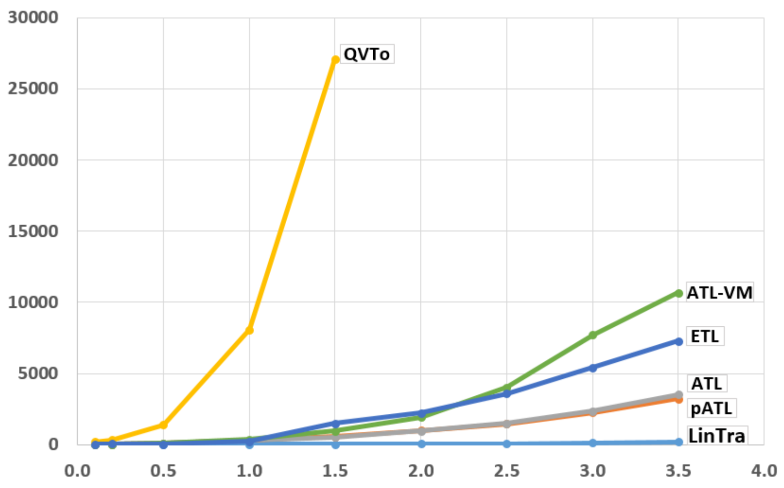


Fig. 3.10 Comparative chart for the IMDb-FindCouples case study.

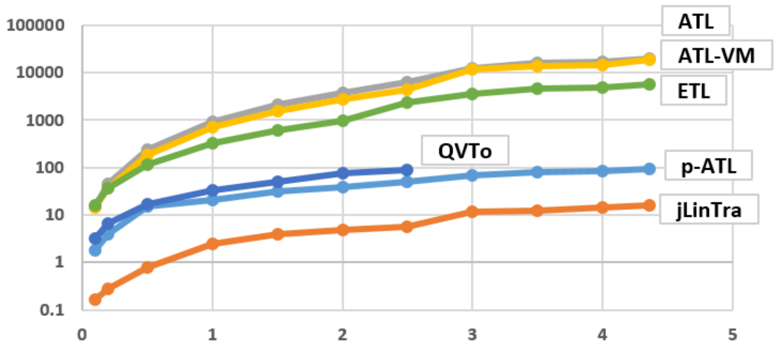


Fig. 3.11 Comparative chart for the Java Refactoring case study.

The Java Refactoring transformation shows with a logarithmic scale that jLinTra beats the rest of the execution engines. It is followed by p-ATL and QVT-O—although QVT-O is not able to transform models with 3 millions of elements or more. The next best option is ETL followed by ATL-VM and ATL which present very similar results.

The Java-to-Graph case study shows a weakness of LinTra. When a transformation needs to navigate through relationships, LinTra needs to access the data layer in each hop to get the corresponding element, given its identifier. In this particular case, the navigation path for the rule that

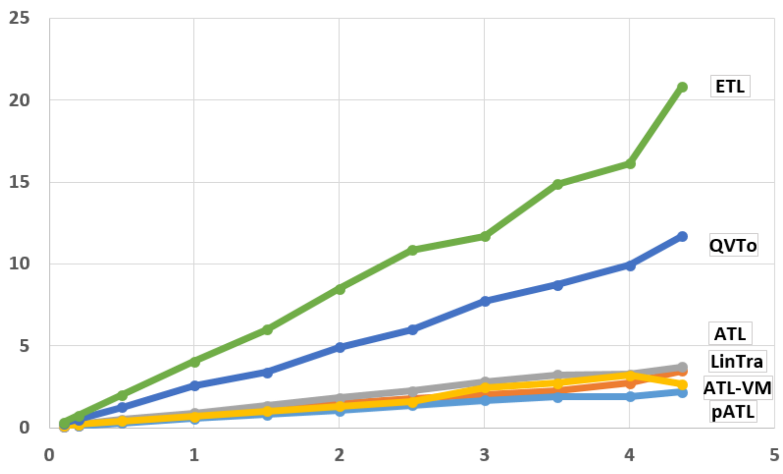


Fig. 3.12 Comparative chart for the Java-to-Graph case study.

transforms classes to nodes needs $1 + depth_{class}$ hops, where $depth_{class}$ is the depth of the class with respect to its root package; and the rule to create the edges from attributes needs $5 + depth_{type} + depth_{absTypeDecl}$ hops where $depth_{type}$ and $depth_{absTypeDecl}$ are the corresponding depths of the attribute type and the class that contains the attribute with respect their root packages. As this transformation requires long navigation paths, the jLinTra performance is affected and p-ATL is slightly faster. The less successful languages are QVT-O and ETL.

	ATL	**ATL-VM**	**QVT-O**	**ETL**	**p-ATL**
DBLPv1	1,300	2,347	7,169	467,863	1,210
DBLPv2	1,368	3,343	8,139	467,100	1,283
DBLPv3	2,518	3,186	3,656	411,477	2,549
DBLPv4	1,746	1,537	2,507	293,648	1,676
IMDb-Identity	17,350	9,754	27,367	12,311	16,486
IMDb-FindCouples	17,960	33,375	367,936*	29,904	18,123
Java Refactoring	749,678	616,913	17,500	243,059	9,276
Java-to-Graph	1,334	1,179	3,656	5,837	0,808

Table 3.8 Average speed-up of jLinTra w.r.t. the rest of the transformation engines.

Table 3.8 summarizes the results previously discussed by showing the average speed-ups for jLinTra with respect to each engine and model transformation. The speed-up for the engine E_i and the transformation T_j is computed as:

$$\text{speed-up}_{ij} = \frac{\sum_{n=1}^{N} \frac{time(E_i,T_j,M_n)}{time(jLinTra,T_j,M_n)}}{N}$$

N being the number of input models for which the transformation has been executed and M_n the n^{th} model. Cells marked with an asterisk ("*") in Table 3.8 indicate that not all the executions finished because the largest models did not fit into memory.

We can conclude that jLinTra is the one that performs better in all but one case, running an average of 97 times faster than the rest of the model transformation engines in the conducted case studies. The only case in which jLinTra was beaten corresponds to its worst-case scenario, when heavy navigation through relationships is required for each element to transform. And even in this case the only engine that beat jLinTra was p-ATL, also because in this case the number of rules was large in the transformation and hence p-ATL could make use of all the machine cores. In summary, jLinTra is between 1.5 and 749 times faster than ATL; between 1.5 and 616 times faster than ATL-VM; between 2.5 and 367 times faster than QVT-O; between 6 and 467 times faster than ETL, and between 0.8 and 18 times faster than parallel ATL.

Results concerning RQ3.

None of the engines with which we are comparing jLinTra permits the parallel execution of model transformation chains but the transformations must be executed sequentially one after the other. In our case, we can start executing the second transformation as soon as the first one produces elements. This is why we conducted this last experiment, in which we compare the performance of jLinTra executing the two transformations in order (the second one starts its execution once the first one has finished and the intermediate model is available) versus executing them in parallel (the second one starts as soon as there are elements in the intermediate model). Table 3.9 shows the comparison results.

These results show that the execution times are similar no matter if the transformations are executed in parallel or not. This makes sense because LinTra follows a data-parallelism approach which means that as long as there are elements to transform available, all the cores are working on the transformation all the time. So all the computing resources are maximally used all the time.

In fact, there is a very slight increase of time when the two transformations are executed in parallel (1% in average). This is due to the synchronization mechanisms needed to execute the two transformations in parallel. Nevertheless, this extra time is justified when the priority is not the overall time but having results in the output model as soon as possible.

Results concerning RQ4.

One of the issues we address in this chapter is the transformation of very large models that do not fit into a single machine memory. Given that all the previous models fit into our machine memory, we have created synthetic models for the IMDb case study according to the procedure described in the TTC case [70]. It explains that the synthetic models must be built by replicating *N* times a given pattern which has 20 elements and 32 references.

In order to easily see the influence on the access time to a database and the storage latency, we decided to execute the identity transformation, which has a linear complexity. We executed the transformation on our machine until we ran out of memory space. Models with 8 million of elements and less were executed using only RAM memory. Larger models started using the hard disk drive to store the parts that could not fit into memory, using Infinispan with a LevelDB database. The execution times (in seconds) obtained for the different models as well as the amount of hard disk space used are shown in Table 3.10.

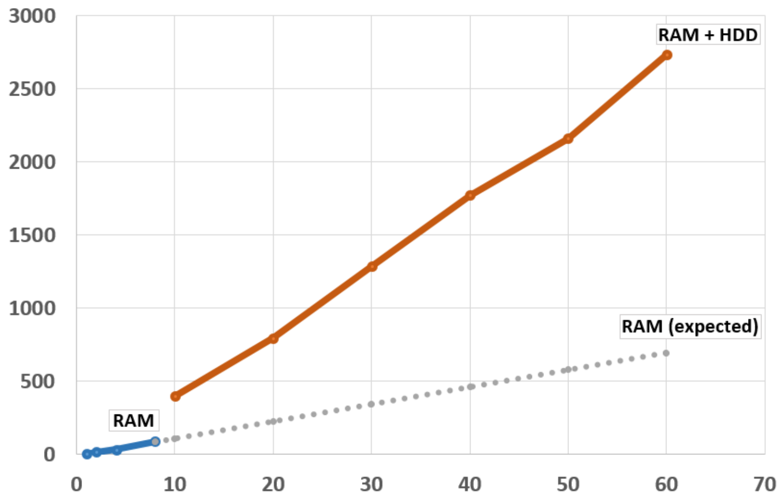


Fig. 3.13 Comparative chart for the IMDb-Identity using RAM memory and hard disk.

Considering only the times obtained for models which needed only RAM memory and applying an interpolation process, the data fit a straight line whose equation is $11.729x - 7.91$ with a coefficient of determination (R^2) of 0.9911 (x represents the size of the models, expressed in millions of elements). We have used that function to predict the execution times we could obtain for the models that do not fit into memory should our machine have more RAM. All the curves are depicted in Fig. 3.13 where the X-axis represents the number of model elements expressed in millions and the Y-axis represents the time taken by the transformation in seconds. Then, we have computed the speed-up between that values and the times obtained experimentally. The results for the models shown in Table 3.10 starting from the model with 10 million of elements are: 3.632, 3.515, 3.687, 3.847, 3.736 and 3.929. The average speed-up is 3.724 which means that the penalty introduced by the hard disk leads to executions 3.7 times slower.

Fig. 3.13 also shows that the disk storage solution is slower, but still linear ($46.48x - 107.08$ with $R^2 = 0.999$). The fact that the introduction of disk does not change the growth model of the runtime function, only changes the constants, is probably the most important aspect for RQ4.

We also wanted to see the impact of using a solid-state drive (SSD) instead of a hard disk drive (HDD). We run the same experiments changing the storage medium and the results showed that the transformations were executed 1.5 times faster when the SSD was used (as expected).

3.3.5 Discussion

Based on the reported results, let's answer the four research questions.

- **Answering RQ1**. The results presented in Subsection 3.3.4 show clearly that jLinTra is always much faster than any sequential engine that we have evaluated. Of course, this is expected because concurrent solutions usually perform faster than sequential ones.

- **Answering RQ2**. The comparison between p-ATL and jLinTra is more interesting as both approaches execute transformations in parallel, although each one uses a different model: data parallelism in jLinTra vs. process parallelism in p-ATL (each rule is executed in one processor). The results obtained in Subsection 3.3.4 show a significant speed-up of jLinTra with respect to p-ATL in all cases but one. Analyzing these speed-ups, we can draw two conclusions. First, and as we expected, the use of navigation paths is more expensive in jLinTra than in p-ATL. This seems to be the only weakness of jLinTra with respect to p-ATL (we also outline the way in which this problem can be addressed later in Section 7.3). In every other case, jLinTra performs better. And second, the size of the input model does not seem to have a significant impact on the performance difference.

- **Answering RQ3**. For practical purposes, jLinTra transformation chains perform basically in the same way when they are executed in parallel as if the transformations are executed sequentially one after the other. That means that no price must be paid when data in the output model is needed as soon as possible, although the output model is not complete.

- **Answering RQ4**. Once the models are large enough to not fit in memory, parts of them are kept in a database in the file storage. The experimental results shown in Subsection 3.3.4 suggest that using a SSD or HDD in combination with RAM memory works well and with only a small delay (between 2.5 and 3.7 times slower respectively), but that all solutions (memory, HDD, SSD) scale equally.

3.3.6 Threats to Validity

In this subsection, we elaborate on several factors that may hinder the validity of our results.

Internal validity—are there factors which might affect the results in the context the case study? Concerning the measurement approach we used in our case study, we have to note that Eclipse is a multi-threaded application. Thus, other ongoing threads within Eclipse could affect our performance measurements. To address this issue, we stopped all additional tasks that might be automatically started, e.g., build processes.

Another threat to validity is the internal representation of the models. For instance, while ATL uses standard EMF, jLinTra uses their own internal format. Thus, there may be differences on how the specifics of EMF are supported and represented.

Finally, we refrained from performing example-specific low-level optimizations that would be possible on the Java code level, in order to compete with ATL and QVT-O in similar conditions.

External validity—to what extent is it possible to generalize the findings for out-place transformations in general? So far, we cannot claim any performance results outside the context of the presented case study. Nevertheless, the evaluation method used in the case study can indeed be applied on other out-place transformation examples as well. Thus, replaying the presented experiments for those transformation cases should enable the possibility of reasoning about the performance of those cases as well by using the provided infrastructure available on our website. However, for transformations going beyond out-place transformations, dedicated evaluation methods and infrastructures may be needed. Finally, the case study may be repeated on other hardware platforms to see, e.g., the impact of the number of cores on the performance.

3.4 Related Work

With respect to the contribution of this chapter, we first elaborate on related approaches which are dedicated to storing and retrieving very large models. Second we discuss closely related work considering the performance of model transformations in general and concerning their parallel execution in particular. Third we discuss different categories of coordination languages and their relation to model transformations. Finally, we relate to other model transformation types going beyond unidirectional out-place transformations.

3.4.1 Persisting Very Large Models

The scalability problems of loading large models represented by XMI documents into memory has been already recognized several years ago. One of the first solutions for EMF models is the Connected Data Objects (CDO)[13] model repository which enables to store models in all kinds of database back-ends such as traditional relational databases or emerging NoSQL databases. CDO supports the ability to store and access large-sized models due to the transparent loading single objects on demand and caching them. If objects are no longer referenced, they are automatically garbage collected. There are also several projects for storing very large EMF models, like MongoEMF[14] and Morsa [45, 46]. Both approaches are built on top of MongoDB. Furthermore, graph-based databases as well as map-based databases are also exploited for model storage such as done in Neo4EMF [9, 59] where also different unloading strategies for partial models are explored [40]. In [35], Clasen et al. elaborate on strategies for storing models in a distributed manner by horizontal and vertical partitioning in Cloud environments. A similar idea is explored

[13]http://projects.eclipse.org/projects/modeling.emf.cdo
[14]http://code.google.com/a/eclipselabs.org/p/mongo-emf

in [41] where different automatic partitioning algorithms are discussed for graph-based models.

Compared to these existing approaches, we use standard data management solutions for storing unstructured information and a Linda-based approach for organizing and accessing the data as we have discussed in Section 3.3.

3.4.2 Transforming Very Large Models

Several lines of research consider the transformation of large models. In this paper, we focus on out-place model transformations running in batch mode or streaming mode. However, to deal with large models, orthogonal techniques may be applied as well. Especially, two scenarios have been discussed in the past in the context of speeding-up model transformation executions, which benefit from alternative execution strategies. First, if an output model already exists from a previous transformation run for a given input model, only the changes in the input model are propagated to the output model. Second, if only a part of the output model is needed by a consumer, only this part is produced while other elements are produced just-in-time. For the former scenario, *incremental* transformations [77, 114, 134] have been introduced, while for the latter *lazy* transformations [138] have been proposed.

Another interesting line of research for executing transformations in parallel is the work on critical pair analysis [66] from the field of graph transformations. This work has been originally targeted to transformation formalisms that do have some freedom for choosing in which order to apply the rules. Rules that are not in an explicit ordering are considered to be executed in parallel if no conflict, e.g., add/forbid conflict (one rule is producing an element which blocks the execution of another rule) or delete/use conflict (one rule is deleting an element which is required to exists for the execution of another rule), is statically computed. However, execution engines follow a pseudo-parallel execution of the rules. But the general notion of critical pairs may be also a valid input for distributing transformation rules. In particular, having non-conflicting transformation rules allows for distributing them without having negative side-effects.

The performance of model transformations is now considered as an integral research challenge in MDE [83]. For instance, Amstel et al. [144] considered the runtime performance of transformations written in ATL and in QVT. In [151], several implementation variants using ATL, e.g., using either imperative constructs or declarative constructs, of the same transformation scenario have been considered and their different runtime performance has been compared. However, these works only consider the traditional execution engines following a sequential rule application approach. One line of work we are aware of dealing with the parallel execution of ATL transformations is [35] where Clasen et al. outlined several research challenges when transforming models in the cloud. In particular, they discussed how to distribute transformations and elaborated on the possibility to use the Map/Reduce paradigm for implementing and distributing model transformations which has been realized in a follow-up work [10]. In addition, Tisi et al. [137] present a parallel transformation engine for ATL. This implementation is used as reference in the evaluation section (cf. Section 3.3) for parallel model transformation engines.

3.4.3 Coordination Models and Languages

A wide variety of models, formalisms and mechanisms were defined in the 90's for describing concurrent and distributed computations based on the concept of *coordination* [52]. The purpose of such models and their corresponding languages was to explicitly deal with the concurrency of cooperation among very large numbers of possibly heterogeneous active entities that comprise a single application, and that can live in distributed settings. There are different approaches to coordination, which can be broadly classified in data-driven and process-driven [109].

From the range of coordination languages available, we realized that the execution of transformation rules mainly depends on the available data in the trace and output models. Thus, rule executions seem to be mostly data dependent. Therefore, we decided to use a data-driven coordination approach instead of a process-driven one (such as the one used for p-ATL, in which each process takes care of a rule [137]). From the data-driven proposals, we decided to use Linda because of its particular features, which make it more suitable for our case. In particular, it has a well-defined set of primitives which are independent of the host language; it naturally implements data parallelism; and it abstracts away all distribution aspects, hence enabling the required separation of concerns requited to tackle the difficult problem of having to deal with concurrency, parallelism and distribution at the same time. Linda (and its many variants) have been successfully used to implement many kinds of parallel applications [109] and there is a wide variety of mature implementations in different languages (cf. Section 3.1.1). However, we are not aware of any other approach using Linda for the parallelization and distribution of model transformations.

3.4.4 Other Model Transformation Types

In this paper we have focused on out-place unidirectional model transformations only. However, there are other kinds of model transformations as well, such as in-place or bidirectional model transformations [38, 98].

In-place transformations are those in which the source and target metamodels coincide. They are normally used when we need to evolve models. For instance, when improving models, e.g., with model refactorings, or executing models, in-place transformations are very appropriate [16]. Although similar in practice, the different semantics of in-place transformations [141] introduce interesting challenges to their concurrent execution, which deserve a separate study. In particular, there are two kinds of in-place model transformation strategies, *non-recursive* and *recursive*, depending on whether recursive matching takes place or not. In recursive matching, the matches of rules are not solely computed based on the initial input model but on the current model state—which may have been modified by previous application of rules. This is the typical strategy followed in graph or rewriting systems, where a set of rules modifies the state of a configuration of objects (representing the model) one by one [117]. In the non-recursive matching strategy, there is one input model which is used to directly compute the output model without considering intermediate steps.

In the case of parallel implementation of recursive in-place transformations there is some work in the field of graph transformations where multi-core platforms are used for the parallel execution of graph transformation rules [11, 72, 84] especially for speeding up the matching phase of the left-hand side of graph transformation rules.

Examples of non-recursive in-place transformations include, e.g., the JavaRefactoring and the Graph-to-ReducedGraph transformations already described in this paper. Of course, this kind of transformation can be implemented in terms of out-place ones, as we have seen above. However, there are some model transformation languages that offer in-place modes too, including ATL, which implements what they call *refining mode* [75, 147]. In a separate paper [22], we also explored this option for LinTra, implementing a non-recursive approach for executing in-place transformations, similar to the ATL refining mode. Our initial results have shown that the LinTra in-place mode significantly outperforms ATL, and that the jLinTra in-place model transformation is on average 1.81 times faster than its out-place version.

Bidirectional model transformations constitute another special kind of model transformations, that can be executed in both directions [131]. They are normally used to keep two (or more) models consistent, since changes in any of them are automatically propagated to the others, resolving potential differences and restoring the overall consistency. Improving the performance of bidirectional transformations, e.g., by their distributed and parallel execution, is considered to be an open challenge [68].

3.5 Summary

In this chapter we have demonstrated the use of the Linda concepts and mechanisms for the parallel execution of out-place model transformations. Based on the results, the jLinTra platform seems quite promising as a base infrastructure for running efficient model transformations when combined with a data parallelism approach to achieve parallelization. We implemented a layer on top of existing Java-based data grids that allows to specify model transformations in Java transparent to the underlying technologies and parallelization strategies. In particular, we showed how to represent metamodels and models for running transformations in parallel as well as how to provide intrinsic traceability for out-place transformations.

The evaluation also shows that current in-memory data grids perform well, i.e., they do not necessarily introduce significant penalties. However, there is a relevant difference in the performance when accessing data. The comparison against other well-known sequential transformation engines also showed a significant speedup and the comparison against a parallel engine using task parallelism showed that data parallelism, as used by jLinTra, seems to work better than process parallelism for the model transformations used in the presented case study.

All in all, the jLinTra solution presented here seems to be able to deal and scale with larger models, and significant outperform current model transformation engines, both sequential and parallel.

	Java-to-Graph-to-ReducedGraph	
No. elements	**Chain in sequence**	**Chain in parallel**
$\mathbf{0.1}\times10^6$	$0,069$	$0,070$
$\mathbf{0.2}\times10^6$	$0,182$	$0,153$
$\mathbf{0.5}\times10^6$	$0,366$	$0,387$
$\mathbf{1.0}\times10^6$	$0,754$	$0,779$
$\mathbf{1.5}\times10^6$	$1,195$	$1,179$
$\mathbf{2.0}\times10^6$	$1,865$	$1,907$
$\mathbf{2.5}\times10^6$	$2,161$	$2,296$
$\mathbf{3.0}\times10^6$	$2,521$	$2,653$
$\mathbf{3.5}\times10^6$	$2,956$	$2,827$
$\mathbf{4.0}\times10^6$	$3,331$	$3,430$
Complete	$3,557$	$3,688$

Table 3.9 Results for the Java-to-Graph-to-ReducedGraph transformation chain.

No. elements	**Time (seconds)**	**Gb. of disk**
$\mathbf{1}\times10^6$	$5,035$	$0,0$
$\mathbf{2}\times10^6$	$17,558$	$0,0$
$\mathbf{4}\times10^6$	$33,859$	$0,0$
$\mathbf{8}\times10^6$	$87,836$	$0,0$
$\mathbf{10}\times10^6$	$397,283$	$5,3$
$\mathbf{20}\times10^6$	$796,720$	$10,8$
$\mathbf{30}\times10^6$	$1268,161$	$16,4$
$\mathbf{40}\times10^6$	$1774,581$	$21,9$
$\mathbf{50}\times10^6$	$2161,152$	$27,4$
$\mathbf{60}\times10^6$	$2733,589$	$33,0$

Table 3.10 Results for IMDb-Identity using RAM memory and HDD.

Chapter 4 Parallel In-place Model Transformations

As presented in Chapter 3, LinTra only permitted out-place model transformations. In this kind of transformations, input and output models often conform to different metamodels and output models are created from scratch. However, there are many situations in which we need to evolve models, instead of creating them anew. For instance, the Java Refactoring case study presented in 3.3 is a concrete case where the input and the output metamodels are the same, and only a few of modifications are needed in the models.

Also, in-place model transformations are the best choice when migrating and modernizing software using Model-Driven Engineering approaches [12, 19], *(i)* software is reverse-engineered to obtain a model representation of the system, *(ii)* modernization patterns are applied on the model level, and *(iii)* the modernized model is translated back into code. Modernization at model level is typically achieved using in-place model transformations, where the initial model is evolved until the final target model is obtained. Models which are reverse-engineered from large systems may be huge, thus high-performing in-place transformation engines are needed. For this reason, we extend our LinTra language with an in-place semantics.

This chapter is structured as follows. Section 4.1 shortly introduces our reference non-recursive in-place semantics. Section 4.2 shows how LinTra realizes its in-place semantics, while Section 4.3 illustrates the benefits of parallel in-place transformations. Finally, Section 4.4 discusses related work before we summarize the work in Section 4.5.

4.1 Background

In-place transformations specify how the input model evolves to obtain the output one, i.e., how the input model has to change. There are two kinds of in-place model transformation strategies, non-recursive and recursive, depending on whether recursive matching takes place or not.

By *recursive matching* we understand that the matches of rules are not solely computed based on the initial input model but on the current model state which probably has been modified by previous application of rules. This is the typical strategy followed in graph or rewriting systems, where a set of rules modifies the state of a configuration of objects (representing the model) one-by-one. Thus, after the application of each rule, the state of the system is changed, and subsequent rules will be applied on the system on this new state. Therefore, the transformation navigates the target model, which is continuously updated by every executed rule.

Regarding *non-recursive matching*, it shares some characteristics with out-place transformations. In this strategy, there is one input model which is used to directly compute the output model without considering intermediate steps.

We chose to follow a non-recursive approach for the LinTra in-place mode. Our decision was also inspired by the ATL *refining mode* [136, 147], used to implement in-place transformations. ATL supports both out-place and in-place modes. In both execution modes, source models are read-only and target models are write-only. This is an important detail that significantly affects the way in which ATL works in refining mode. Indeed, ATL in-place mode does not execute transformations as these are executed in graph or rewriting systems, as explained in detail in [141]. Thus, we follow as well non-recursive matching in LinTra where rules always read (i.e., navigate) the state of the source model, which remains unchanged during all the transformation execution.

4.2 Approach and Semantic Issues

LinTra follows a non-recursive approach for executing in-place transformations, as the ATL refining mode does. In this section we discuss some semantic issues that might occur in rule-based in-place model transformations in general as they are indeed highly relevant for the parallel execution of in-place transformations.

4.2.1 Atomic Transformation Actions

When executing a non-recursive in-place transformation, the first decision concerns the elements for which the transformation does not specify what to do. We could either decide to exclude them from the target model or to include them as they are. In jLinTra we decided for the second option, which implies that if we want to exclude objects in the target model, the transformation will have to explicitly remove them. Thus, after the input model is loaded, and once the transformation phase starts, an initialization phase is needed where the identity transformation is applied so that the target area contains a copy of the input model.

After the model is copied, in the following we explain the three operations that may be applied to it: deletion of elements, creation of new elements, and modification of existing elements.

Elements Deletion. When an element is deleted, the outgoing relationships from such element to others are deleted too, since such information is stored as attributes in the deleted element. However, the situation is different when the deleted element has incoming relationships. In such case, the information about relationships to the deleted element is stored in the attributes of other elements. In this case, we can distinguish two different semantics. Either all the incoming relationships are deleted, for which the engine needs to traverse the whole model searching for relationships pointing to the deleted element, or they are not deleted, causing dangling references and, consequently, an inconsistent model. In the former option, we need to keep track of all the deleted elements, so that the traversal is realized only once as the last step of the transformation. The latter option is useful in order to make the user aware that he/she is removing an element by mistake. LinTra permits both behaviors, since it is aimed at offering a flexible implementation. When the deleted element is the parent of a containment relationships, all its descendant are also deleted recursively.

Elements Creation. If the developer wants to create a new element, he/she has to create the instance and set its attributes and relationships. In case of bidirectional relationships, there are two alternatives: *(i)* the opposite reference is created automatically, or *(ii)* the creation of the opposite relationship must be explicitly specified by the developer. We permit both behaviors.

Elements Updates. Updating an attribute or an ongoing unidirectional relationship of an element is trivial, since the transformation only has to change the corresponding attribute of the updated element. However, there are again two choices when updating a relationship which is bidirectional, since the previous target element of the relationship would still have a relationship to the updated element unless something is done. Thus, *(i)* the relationship from the previous target element should be automatically removed and a new relationship from the new pointed element to the updated element should be automatically created, or *(ii)* the developer has to specify explicitly in the transformation that the corresponding relationships are removed and created respectively. Again, we permit both alternative behaviors.

4.2.2 Confluence Conflicts

Confluence conflicts typically occur when two rules are applied to the same part of the model and they treat it differently [66]. Thus, the resulting model may vary depending on the order in which those rules are applied. The application of a rule can conflict with the application of another rule in four different ways. Let us explain them for the ATL refining mode which acts as blueprint for the LinTra in-place transformation strategy. For the explanations, let us imagine a transformation for reverse engineering Java code.

Update/Update. Imagine that a rule sets the public variables to private and capitalizes the name of the ones that are private. This case is not a problem for the confluence of non-recursive in-place transformations since only the source model provided by the user is read—the changes done by the rule that changes the visibility are not visible to the rule that capitalizes the names of the variables. On the contrary, if a rule sets the visibility of the variables to private and another rule sets them to public, the transformation may not be confluent. A possible way to prevent this situation is to force the precondition of the rules to be exclusive, which leads to non-overlapping matches. This was the solution adopted by ATL concerning the declarative part. Nevertheless, it is easy to fool ATL by using the imperative part, which is executed after the declarative part of the rule.

Delete/Update. Suppose that a rule sets the visibility of the variables to private and another rule removes all the variables. The situation is similar to the second case we presented for the conflict *Update/Update*. The two rules are a conflicting pair, thus the language should prevent this situation from happening or should establish the behavior of the transformation. Again, it is possible to produce this case in ATL by using the imperative part to set the visibility and writing a declarative rule that removes the variables. Both rules are executed so that the visibility is changed and the variables are removed. As a result, the variables are not present in the resulting model. Apparently, the objects are removed in a later execution phase, after having done all the updates and creations specified in the declarative and imperative parts.

Produce/Forbid. Imagine that a rule adds a variable to a class and another rule removes all the empty classes (classes with no variables) from the model. The first rule is producing an additional structure that is forbidden by the precondition of the second rule. Once again, the order in which the rules are executed influences the result. This time, if we try to implement this transformation with ATL using the imperative part of a rule to add the variables and a declarative rule to remove the empty classes, both rules are applied but the transformation does not fulfil the purpose for which it was written (since only the source model is read). As a result, the classes are removed but the newly created variables remain in the model without any container.

Delete/Use. This conflict appears when a rule deletes elements that produce a match with another rule. Thus, it is the opposite case to *Produce/Forbid*. Depending on the order in which the rules are executed, the transformation is able to execute a higher or lower number of rules.

We have illustrated the conflicts that may appear between rules and how ATL tries to solve them using non-overlapping matches, how they can be avoided or produced, and which is the final result of the execution. Enforcing to have non-overlapping rules is not the only solution; another possibility is to statically detect the conflicting rules using the critical pair analysis approach [99], and subsequently, to deal with the conflicts making use of layers which is also implicitly done in ATL by using different phases in the transformation execution.

As jLinTra is realized as an internal transformation language embedded in Java, we have opted for not imposing any restriction. Thus, our solution is completely flexible with respect to rule executions. The idea is that high-level model transformation languages (such as ATL [75], ETL [82], or QVT-O [OMG]) are automatically compiled to jLinTra. In case that the critical pair analysis is needed, it can be done statically during the compilation process from the high-level model transformation language to LinTra.

4.3 Evaluation

To evaluate our approach we performed an experimental study concerning a transformation which, in reverse engineered Java applications, removes all the comments, changes the attributes from public to private and creates the getters and setters.

4.3.1 Research Questions

The study was performed to quantitatively assess the quality of our approach by measuring the runtime performance of the transformations. We aimed to answer the following research questions (RQs):

1. *RQ1—Parallel vs. sequential in-place transformations*: Is the parallel execution of in-place transformations faster in terms of execution times compared to using the state-of-the-art sequential execution engines? And if there is a positive impact, what is the speedup with respect to the used number of cores for the parallel transformation executions?

2. *RQ2—Parallel in-place vs. parallel out-place transformations*: Is the parallel execution of in-place transformations faster in terms of execution time compared to using their equivalent out-place transformations?

4.3.2 Experiment Setup

To evaluate our approach, we have used the same Java models we used in Section 3.3.2.

We apply an extended version of the `Public2Private` transformation—the original one is available in the ATL Zoo [61]—that changes the visibility of every `public` variable to `private` and creates the corresponding getter and setter methods. In addition, the transformation also removes all the comments contained in the code. All artifacts can be downloaded from our website [27].

Let us show the effects of this transformation with an example. Listing 4.1 shows the Java code that declares a class called `MyClass`, a public attribute `name` and the class's constructor. The code contains some comments too. After applying the transformation, the Java code that the model represents should look like the fragment presented in Listing 4.2.

An excerpt of the code corresponding to the rules in jLinTra is shown in Listing 4.3. As stated in Section 3.1, every slave is in charge of transforming a chunk of the model. For efficiency reasons the changes are made permanent once the whole chunk has been transformed. In order to keep the temporary

Listing 4.1 Code to be refactored

```java
public class MyClass {
    //Declaration of variable called name
    public String name; /* This variable contains the name */
    public MyClass() {
        /* Description @param ... */
        ...
    }
}
```

Listing 4.2 Refactored code

```java
public class MyClass {
    private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public MyClass(){ ... }
}
```

changes the structures `deletedElems`, `modifiedElems` and `createdElems` (lines 2, 8 and 9) are needed.

We have run all our experiments on a machine whose operating system is Ubuntu 12.04 64 bits with 11.7 Gb of RAM and 2 processors with 4 hyperthreaded cores (8 threads) of 2.67GHz each. We discuss the results obtained for the different transformations after executing each one 10 times for every input model and having discarded the first 5 executions as the VM has a warm-up phase where the results might not be optimal. The Eclipse version is Luna. The Java version is 8, where the JVM memory has been increased with the parameter *-Xmx11000m* in order to be able to allocate larger models in memory.

4.3.3 Performance Experiments

The in-place transformation described before has been implemented and executed in jLinTra and in ATL, for which we have used the EMFTVM [147]. We have also developed an out-place transformation version in jLinTra in order to compare its performance with the proposed in-place version. Table 4.1 shows in its left-most column the number of entities of the source models of

Listing 4.3 jLinTra transformation

```java
if (ie instanceof Comment){
    //Delete Comment
    deletedElems.add(ie);
} else if (ie instanceof FieldDeclaration){
    String modId = ((FieldDeclaration) ie).getModifier();
    Modifier mod = (Modifier) srcArea.read(modId);
    String visibility = mod.getVisibility();
    if (visibility.equals(PUBLIC)){
        // Modify visibility
        mod.setVisibility(PRIVATE); modifiedElems.add(mod);
        ...
        // Create getters and setters
        createdElems.add(...);
    }
}
```

the transformation. The second, third, and fourth columns correspond to the execution times (in seconds) obtained for ATL and jLinTra (using the in-place and out-place modes), respectively. Note that we have only taken into account the time of the execution of the transformation, meaning that we do not consider the time used for loading the models into memory, nor the time used to serialize them to the disk. The fifth column presents the speedup of jLinTra with respect to ATL. We can see that the speedup is not constant: it grows with the size of the model, reaching a value of 955.23 for the complete model, meaning that value that jLinTra is 955.23 times faster than ATL for this concrete case. Finally, column six shows the speedup of the in-place and out-place modes of LinTra, where we can see that the in-place model transformation is on average 1.81 times faster than its out-place version.

We already mentioned in Section 4.2 that an initialization phase where the input model is copied to the target area is needed. However, if we moved that process to the loading phase so that both the source and target areas were loaded at the same time, we would only pay a minimum price (an overhead of 5% in the loading phase) and the performance in the transformation phase would be improved reaching a speedup of 3.89 w.r.t. the out-place mode and speedup of 1,195 w.r.t. ATL.

	ATL	**LinTra**		*Speedups*	
No. elements	**EMFTVM**	**In-place (LI)**	**Out-place (LO)**	*LI–EMFTVM*	*LI–LO*
0.1$\times 10^6$	2.40	0.11	0.19	21.23	1.72
0.2$\times 10^6$	12.04	0.29	0.36	41.88	1.25
0.5$\times 10^6$	65.06	0.73	0.98	89.06	1.34
1.0$\times 10^6$	371.41	1.29	2.38	287.34	1.84
1.5$\times 10^6$	1042.41	2.06	2.61	506.71	1.27
2.0$\times 10^6$	2030.82	2.99	5.63	678.16	1.88
2.5$\times 10^6$	2952.46	3.92	9.64	754.14	2.46
3.0$\times 10^6$	4156.69	5.13	8.82	809.92	1.72
3.5$\times 10^6$	5527.96	6.26	13.77	883.37	2.20
4.0$\times 10^6$	6737.97	7.57	15.20	890.70	2.01
Complete	7238.70	7.58	17.18	955.23	2.27

Table 4.1 Execution results and speedups.

Regarding the gain of in-place MTs in LinTra w.r.t. the number of cores involved in the transformation, the speedups of using only one core w.r.t. using four, eight, twelve and sixteen are 1.19, 1.62, 1.97, 3.24, respectively.

We also planned to execute and compare this transformation with the original ATL virtual machine. However, although it supports the refining mode it does not support the imperative block, which is applied in the particular transformation used in this study.

Regarding the out-place transformation developed in LinTra, it explicitly specifies that all elements that are not modified must be copied, together with their properties. The out-place transformation counts on $3,302$ lines of Java code (we generated the code for the identity transformation using Xtend[1] and adapted the corresponding code to fit the needs of the `Public2Private` transformation), while the in-place transformation has only 194 lines.

For answering the two research questions stated above, we can first conclude that the parallel execution of in-place transformations reduces the execution time compared to using sequential execution and that the execution time can be further improved by adding more cores. Second, for typical in-place transformation problems, parallel in-place transformation executions are more efficient than executing their equivalent out-place transformations.

[1]https://eclipse.org/xtend/

4.3.4 Threats to Validity

In this section, we elaborate on several factors that may jeopardize the validity of our results.

Internal validity—Are there factors which might affect the results of this experiment? The performance measures we have obtained directly relate to the experiment we have used for the evaluation. Therefore, if we had used different experiments other than the `Public2Private` transformation then the speedups between the executions of the different implementations would have probably been different. Besides, we have generated 11 smaller sample source models. Should we have generated different models, the results in Table 4.1 would have also been different. As another threat, we have decided to use the executions after the 5th one in order to avoid the possible influence of the VM warming-up phase. However, if after the 5th execution the VM has not finished warming up, our results are then influenced. Finally, we are quite confident that we have correctly written the equivalent transformation in ATL due to our expertise with such language. Nevertheless, there may exist tiny differences which may have an influence on the execution times.

External validity—To what extent is it possible to generalize our findings? As a proof of concept, we have compared the execution times of our approach with the ATL implementation executed with the EMFTVM engine. We have chosen ATL for the comparison study because we have enriched LinTra with the same in-place semantics that ATL has. Therefore, since our study only compares LinTra and ATL, our results cannot be generalized for all non-recursive engines.

4.4 Related Work

In this chapter, we have focused on in-place model transformations running in batch mode.

An important line of research for executing transformations in parallel is based on critical pair analysis [66] from the field of graph transformations as discussed in Section 4.2 and as we mentioned in the previous chapter. This work has been originally targeted to transformation formalisms that do have some freedom for choosing the order in which to apply the rules. Rules that are not in an explicit ordering are considered to be executed in parallel if no conflict is statically computed. Although, most existing execution engines follow a pseudo-parallel execution of the rules, but there are already some emerging approaches which consider the execution of graph transformations in a recursive way on top of multi-core platforms [11, 72, 84]. A closer comparison concerning the commonalities and differences of recursive and non-recursive in-place semantics concerning parallelism is considered as a subject for future work.

As we also mentioned in the Related Work of the previous chapter, Clasen et al. and Tisi et al. are working on model transformations on the cloud [35] and a parallel transformation engine for ATL [137] respectively. However, they only consider out-place transformations while we tackled the parallel execution of in-place transformations.

4.5 Summary

We have presented an extension for LinTra that allows the parallel execution of in-place model transformations. We have shown with experiments that the performance is improved w.r.t. other in-place MT engines and that in cases where in-place transformations can be achieved also by means of out-place transformations, the in-place transformations provide better performance and usability.

Chapter 5 Testing Model-to-Model Transformations

In the previous chapters, we have already seen the importance of model transformations in the context of MDE and we have worked on improving their performance. Once we count on a software artifact given by a model transformation, another important property to check is their correctness as the quality of the resulting systems is highly influenced by the quality of the model transformations employed to produce them. However, users of transformations have to deal with the problem that transformations are difficult to debug and test for correctness [8]. In fact, as the size and complexity of model transformations grow, in many cases manual debugging is no longer possible, and there is an increasing need to count on methods, mechanisms and tools for testing their correctness [64, 8].

In general, debugging is readily classified into three parts: the identification of the existence of a problem, the localization of the fault, and the actual correction of the problem [126].

In this chapter, the existence of a problem is detected by the misalignment between the model transformation specification and its implementation. The former specifies the *contract* that determines the expected behavior of the transformation and the context in which such a behavior needs to be guaranteed, while the latter provides the actual behavior of the transformation. If the transformation does not behave as expected, a violation of the contract occurs.

Here we use Tracts [56] for the specification of model transformations, which are a particular kind of *model transformation contracts* [7, 33] especially suitable for specifying model transformations in a modular and tractable manner. Tracts count on tool support for checking, in a black-box manner, that a given implementation behaves as expected—i.e., it respects the Tracts constraints [26].

Once a problem has been found (i.e., a constraint has been violated), we need to locate the fault [67]. One of the major shortcomings of model transformation specification approaches based on contracts is the lack of traceability links between specifications and implementations. In the case a constraint is not fulfilled, the elements involved in the constraint evaluation could provide valuable information to the transformation engineer, but the links to the transformation implementation are not available.

This chapter presents a solution to this problem. It uses a white-box and static analysis to find the location of the model transformation rules that may have caused the faulty behavior. It provides the first step of an iterative approach to model transformation testing, which aims at locating faults as early as possible in the development process. Although this step cannot fully prove correctness, it can be useful for identifying many bugs in a very early stage and in a quick and cost-effective manner [153]. It can also deal with industrial-size transformations without having to reduce them or to abstract away any of their structural or behavioral properties, and it can represent a very valuable first step before diving into more expensive and complex tests (such as model checking, formal validation, dynamic tests, etc. [6, 43, 44, 87, 30, 2]) which represent numerous challenges, mainly because of their inherent computational complexity [8, 7].

An evaluation discussing the accuracy and the limitations of the approach is also provided. The evaluation has been conducted on a number of transformations with the goal of quantitatively assessing the correctness (Are the alignments correct?), completeness (Are there any missed alignments?) and usefulness (How useful is the resulting information to the developer for locating the faults?) of the techniques. Furthermore, we also identify the kinds of transformations which are most suitable for validation with the proposed approach, and provide a test to automatically check this a-priori. Finally, we use mutation techniques to evaluate its effectiveness.

This chapter is organized as follows. Section 5.1 introduces the proposed approach and Section 5.2 discusses how we have implemented it. Section 5.3 is devoted to the evaluation of our proposal, and to analyze its advantages and limitations. Finally, Section 5.4 presents related work before Section 5.5 summarizes the chapter.

5.1 Matching Tables

5.1.1 Motivation and Challenges

As discussed in Section 2.3, Tracts allow us to define constraints for specifying model transformations. Regarding the model transformation implementation, our approach could be applied to any MT language that uses the metamodel footprints, independently if it is a high-level model transformation language (such as ATL [75], ETL [82], QVT-O [OMG], etc.), or an intermediate language that will be compiled to a high-level language (such as QVT Core [OMG] or LinTra). For readability reasons and because it is one of the most popular languages in which model transformations are written, we have chosen ATL.

Having independent artifacts for the specification and implementation of model transformations permits choosing which formalism to use for each level. However, the following questions cannot be answered without a thorough analysis of both artifact types:

- Which transformation rule(s) implement(s) which constraint(s)?
- Are all constraints covered by the transformation rules?
- Are all transformation rules covered by the constraints?

In order to establish the relation between the constraints and the rules that might make them fail, two approaches can be followed: *dynamic* or *static*.

Dynamic approaches are based on a concrete model transformation execution over a model or set of models. The procedure consists of tracking the transformation process and storing information about each executed step and the specific instances. Once the transformation has finished and the failures and the objects that caused them are known, it is necessary to go backwards over the trace information stored during the transformation execution to find the errors. In these approaches, an input model needs to be available to execute the transformation, and the environment where the transformation is to be executed must be provided too.

Static approaches, on the other hand, do not make use of executions. They obtain the relation between the constraints and the rules by means of an algorithm. The only inputs for this process are the transformation implementation and the specification constraints.

Dynamic approaches normally give more precise results, although, as mentioned before, they are dependent on the particular input model and transformation execution, while static ones can compute more general alignments. In this chapter, we target the challenge of finding "guilty" transformation rules following a static approach. Since there is no direct relation between the rules and the constraints (constraints are created independently of any transformation implementation), our work computes for each pair (constraint, rule) the probability that the constraint failure comes from the rule making use of the common denominator that both have: the structural elements belonging to the metamodels.

It can also be considered a white-box approach, because it takes into account the internal structure and details of the tract constraints and of the transformation implementation.

5.1.2 Methodological Approach

Given a set of OCL constraints (from the Tracts) and a set of ATL rules, Fig. 5.1 summarizes the commonalities between them (in the figure, relationships ≪c2≫ and ≪u≫ stand, respectively, for "conforms to" and "uses"). There is also a direct relation between the ATL and the OCL metamodels, because the former embeds the latter. This may simplify the alignments between ATL and OCL, although it is also true that the OCL constraints and the ATL rules are written differently. First, the former impose conditions on the relationship between the source and target models, while the latter describe how the target model should be built from the elements of the source model. Second, specifications and implementations are normally written by different people, at different times, with different goals in mind, and using different styles (e.g., they may use different navigation paths to refer to the same elements, because the starting contexts are not the same, or use different OCL operators

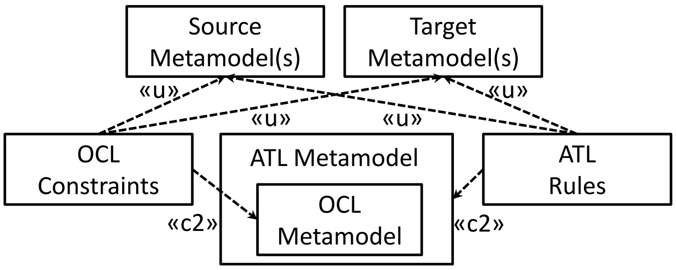


Fig. 5.1 Heterogeneities and Commonalities between Constraints and Rules.

for querying elements). Finally, there are slight differences between OCL and ATL, e.g., ATL introduces additional operations which are of particular interest for transformations and which are not available in OCL. In any case, the OCL constraints and the ATL rules make use of the same source and target metamodels. As we have seen for the *Families2Persons* example in Chapter 2, the same types and features are used in the specification and in the implementation of the transformation. Thus, we use these commonalities to indirectly match the constraints and the rules by matching their footprints concerning the source and target metamodels used.

Our approach focuses on the construction and interpretation of the so-called *matching tables* with the alignments we have discussed before. Thus, our approach builds on the following steps:

1. **Footprint Extraction**. The *structural elements* (henceforth referred to as *footprints* or *types and features*) of both model transformation and constraints are extracted, as explained later in Section 5.1.3.

2. **Footprint Matching**. The footprints extracted in the previous step are compared for each rule and constraint.

3. **Matching Tables Calculation**. The percentage of footprints overlapping, so-called *alignment*, for each transformation rule and constraint is calculated. This information is used to produce the matching tables (cf. Section 5.1.4).

4. **Matching Tables Interpretation**. The resulting tables are analyzed for identifying the guilty rules for each constraint. Guidelines for this analysis, exemplified with a case study, are described in Section 5.1.5.

5.1.3 Footprint Extraction

Now we present how we extract footprints from OCL constraints and ATL rules.

Constraints

There are several possibilities for the footprints extraction of OCL constraints. For example, we could take into consideration all types and features that appear in the OCL expressions, just because they are mentioned. We could even assign weights to these types and features according to their number of occurrences in the constraints, giving less importance (a lower value) to those that appear less often. However, due to the nature of OCL, nesting is necessary to implement correct restrictions in order to isolate the information which is really relevant for our purposes thus, it is important to distinguish between two different kinds of elements that appear in the OCL expressions: those that we want the constraint to refer to, and those which are used for navigation purposes only.

Since metamodels are graphs, OCL expressions are heavily dependent on their *contexts* (i.e., the starting class) [31] and also on the path used to navigate to the final type, which is precisely the one we want the constraint to refer to, i.e., starting from a specific class several navigation paths can lead to the same target class, which is the one that really matters from the constraint perspective, whereas all other classes in the navigation can be considered as mere implementation details. Thus we need to isolate the target features of the constraint from the ones used to reach it. This is why we only consider as relevant the last elements of the OCL expressions. For example, if we have `Family.mother.firstName`, then we will only consider `mother.firstName` whose footprints are `Member` and `Member.firstName`.

When an OCL expression contains operations on collections, we take into account only the types inside the body of the deepest (in the sense of nesting) iterators (*forAll*, *exists*, etc.) to extract just the relevant footprints and not those used for navigation purposes only. After doing some experiments, we realized that this decision helps introduce less noise, i.e., it does not extract types which are not representative for the constraint (since they are used for navigation purposes, and not for model element transformation), what in turn contributes to the modularization and independence of the types extracted in the constraints.

Similarly, primitive types and constants are not considered. Types like *Integer* or *Boolean*, or constants like *true* or *false* can appear frequently, but this does not mean that each appearance provides relevant information for locating a fault. On the contrary, taking them into consideration only introduces more confusion, when precisely our goal is to isolate those elements that are more relevant for locating the faults.

Rules

In this chapter we deal with ATL as proof of concept, although any transformation language based on rules and that uses OCL could be used. For each rule, we obtain the footprints in the left-hand side, right-hand side and imperative part, and build all navigation paths. Then, as in the OCL constraints, we only consider the last part of these paths. Regarding helpers, they can appear in any part of a navigation path. For this reason, when there is a helper in a path, we simply obtain the type it returns. If it is a collection type, we obtain the type of the collection.

We apply the same approach for calls of ATL (unique) lazy rules and called rules. In these cases, we return the type of the first element created by these rules (since this is what ATL actually returns).

With all this, the footprints extracted for the *Families2Persons* example presented in Section 2.3.2 are shown in Table 5.1, for each rule and constraint.

Constraint	Considered Types and Features
C1	Member, Family, Family.daughters, Family.sons
C2	Member, Family, Female, Member.firstName, Family.lastName Female.fullName
C3	Member, Family, Female, Family.lastName, Member.firstName Female.fullName
C4	Member, Family, Male, Member.firstName, Family.lastName Male.fullName
C5	Member, Family, Female, Family.lastName, Female.fullName Member.firstName
C6	Member, Family, Male, Family.lastName, Male.fullName Member.firstName
C7	Member, Person
C8	Person, Person.fullName

Rule	Considered Types and Features
R1	Member, Male, Member.firstName, Male.fullName
R2	Member, Female, Member.firstName, Female.fullName

Table 5.1 Footprints for the *Families2Persons* example.

5.1.4 Footprint Matching and Matching Tables

A tabular representation (called *matching tables*) is used to depict the alignment between constraints and rules. We apply three different matching functions to automatically obtain the values for filling the tabular representations. Each function provides a certain viewpoint on the alignment. This allows us to interpret the results and provides an answer to the questions presented in Section 5.1.1.

In these tables, rows represent constraints and columns represent rules. Each cell links a constraint and a rule with a specific value between 0 and 1. Let C_i be the set of types and features extracted from constraint i and R_j from rule j. Let $|\cdot|$ represent the size of a set.

Matching Tables: Three Different Viewpoints

The *constraint coverage* (CC) metric focuses on constraints. This metric measures the coverage for constraint i by a given rule j. For this metric, the value for the cell $[i, j]$ is given by the following formula.

$$CC_{i,j} = \frac{|C_i \cap R_j|}{|C_i|} \tag{5.1}$$

Since the denominator is the number of types and features in C_i, the result is relative to constraint i and we interpret this value for rule traceability, i.e., to find the rules related to the given constraint. This is, if a constraint fails, the CC table tells us which rule or rules are more likely to have caused the faulty behavior (i.e., be “guilty"). Thus, the CC table is to be consulted by rows.

The *rule coverage* (RC) metric focuses on rules. This metric calculates the coverage for rule j by a given constraint i. We use the RC table to express constraint traceability, i.e., to find the constraints more closely related to a given rule, and therefore it is to be read by columns. The metric is calculated as follows.

$$RC_{i,j} = \frac{|C_i \cap R_j|}{|R_j|} \tag{5.2}$$

The last metric is relative to both constraints and rules, so the RCR table can be consulted by rows and by columns. Thus, it provides information about the relatedness of both rules and constraints, without defining a direction for interpreting the values. The *relatedness* of *constraints* and *rules* (RCR) metric is computed as follows.

$$RCR_{i,j} = \frac{|C_i \cap R_j|}{|C_i \cup R_j|} \tag{5.3}$$

The overlap between the elements extracted from the constraints and the rules gives rise to five different cases which are reflected by the previous metrics. They are depicted in Fig. 5.2 using Venn diagrams.

In case (a), each element present in the constraint is contained in the set of elements in the rule: $C_i \subseteq R_j$. Consequently, the value for the CC metric is

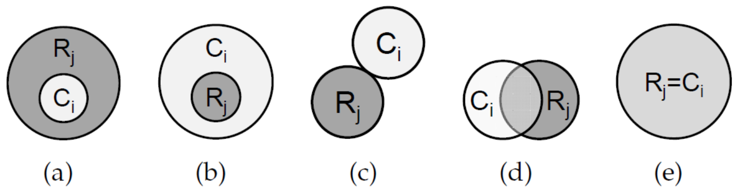


Fig. 5.2 Possible overlaps for C_i and R_j.

1, meaning that the constraint is fully covered by the rule. The other metrics have a value lower than 1.

In case (b), all the elements in the rule are contained in the elements of the constraint, $R_j \subseteq C_i$. RC metric is 1.

In case (c), C_i and R_j are disjoint sets. Thus, the three metrics are 0, which means that the given constraint and the given rule are completely independent.

In case (d), each metric will have a value between 0 and 1. The specific value depends on the size of the sets and on the number of common elements. Thus, the bigger the common part for C_i is, the closer to 1 the value for metric CC will be. Similarly for R_j and metric RC. Regarding the RCR metric, its value only depends on the size of the common part (for a specific size of the footprints); the bigger it is, the closer to 1 the value will be.

In case (e), both constraints and rules have the same elements set, so all metrics are 1.

Considering subtyping. In the three formulas presented above, we consider the intersection $C_i \cap R_j$ as the common elements present in constraint C_i and rule R_j. But we should also take subtyping into account. Its consideration is important because some OCL operators used in the Tract constraints and in the ATL rules (such as `allInstances`) retrieve all instances of a certain class, as well as the instances of all its subclasses, and therefore we can have types in a constraint and in a rule that are not directly related (since they are not the same type), but are related via subtyping (when one type is a sub/super-type of the other). Thus, the fault may be due to a problem not only in a class but also in any of its superclasses. To take this into consideration, we assign a weight to the parent classes, given by the number of its structural features (attributes and references) divided by the number of features of the child class—both sets comprise the class's own features as well as the inherited features from all its superclasses. Thus, the more similar the parent and the child are, the closer to 1 the weight is. Similarly, if the child class incorporates many new features w.r.t. the parent class, the weight assigned to the parent will be closer to 0.

Setting a threshold value. Before going any further, let us explain the need for setting a threshold for cell values in the matching tables. Such a threshold is meant to establish a boundary under which alignments are ignored. It is needed to be able to disregard those situations where a constraint and a rule are minimally related, and thus should not be considered as relevant for locating the fault. Moreover, if a value in a cell is below the threshold in table RCR, then the value in the equivalent cells in the other two tables must be disregarded too, even if their value is above the threshold, to avoid considering irrelevant information.

Fig. 5.3 helps explain this situation. Assume that the elements extracted from constraint C_i are a subset of the elements extracted from rule R_j, as shown in Fig. 5.3(a). In this case, the CC metric for this pair is 1. However, since the set of common elements is very small in comparison with the size of the set of rule elements, the RCR metric is also very small. Despite there being some common elements in the rule and in the constraint, it does not mean that, in this case, the rule is covering the constraint. In most cases where the set of common elements is much smaller than the set of rule elements (even if the CC metric is 1), it is normally because our metamodels are small and the same element may be present in several rules and constraints, and not because there is a relevant relationship. In such cases, when a value is lower than the threshold, we consider that it is not relevant and therefore we do not take it into account. In Fig. 5.3(b), all the elements in the set C_i are also a subset of the elements in rule R_k. The difference lies in the fact that the ratio $|Ci|/|R_k|$ is higher and thus, a relevant value. This means that it is more likely that the rule R_k is implementing the use case that constraint C_i is specifying and, therefore, we should consider the alignment between

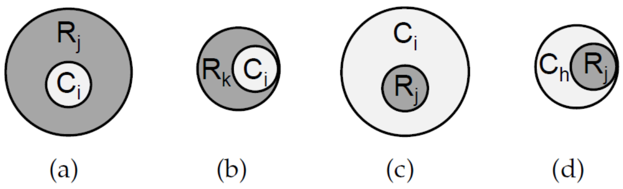


Fig. 5.3 Situations with differently sized rule/constraint footprints.

them as being relevant for our purposes. In fact, in order for the constraint to be properly covered, there should exist a rule that covers the constraint with a large portion. In such a case, the RCR metric would be higher than the threshold and the CC metric shall be considered. Similarly for metric RC, let us suppose that rule R_j is completely covered by constraint C_i, as in Fig. 5.3(c). In this case, the RC metric is 1, since all the elements of R_j are included in C_i. However, as very few elements of C_i are present in R_j, the RCR metric is very small, so the RC metric should not be taken into account. There should exist, consequently, a constraint that has a larger portion of its elements in common with R_j. Fig. 5.3(d) shows an example where the value of RCR is above the threshold and, thus, metric RC is considered.

In summary, this threshold is needed to eliminate the consideration of matches with very low probability, which only cause interferences when looking for the rules that cause the fault. The need for this threshold is based on our experiments with the tables. The current value for the threshold is 0.1. This means that, at least, 10% of the elements that appear in a rule must be present in a constraint in order to consider the CC metric between both. Similarly, at least 10% of the elements in a constraint must be covered by a rule in order to take their RC metric into account. This value has proved to be the most effective threshold for obtaining the highest recall and precision in all the case studies we have analyzed. Research is currently in progress to provide a theoretical justification for such a value. In any case, this value is currently a configuration parameter in our toolkit to allow easy tuning.

Example. Table 5.2 shows the metrics computed for the *Families2Persons* example, presented in Section 2.3. Note that, for a small example like this, the metrics provide information that can be easily interpreted by just looking

	CC		RC		RCR	
	R1	**R2**	**R1**	**R2**	**R1**	**R2**
C1	0.33	0.33	0.25	0.25	0.17	0.17
C2	0.33	0.67	0.50	1.00	0.25	0.67
C3	0.33	0.67	0.50	1.00	0.25	0.67
C4	0.67	0.33	1.00	0.50	0.67	0.25
C5	0.33	0.67	0.50	1.00	0.25	0.67
C6	0.67	0.33	1.00	0.50	0.67	0.25
C7	1.00	1.00	0.50	0.50	0.4	0.4
C8	1.00	1.00	0.50	0.50	0.33	0.33

Table 5.2 *Families2Persons* matching tables.

at the constraints and the rules. The second and third columns express the constraint coverage, the fourth and fifth ones the rule coverage, and the sixth and seventh ones the relatedness.

Matching Tables for *UML2ER*

The *Families2Persons* case study presented so far is a rather small example, although sufficient for demonstrating the basic process of computing the different metrics. Let us analyze here a bigger transformation, namely the *UML2ER* project, from the structural modeling domain. It generates Entity Relationship (ER) Diagrams from UML Class Diagrams.

We have extended the metamodels for the *UML2ER* case study presented in [151]. They are illustrated in Fig. 5.4, and the Tracts we have defined for it are shown in Listing 5.1. Please note the black triangle symbol used in the Listing for the sake of brevity. It is used for marking the place in a constraint (triangle down) that may be extended by another constraint (triangle up). For instance, constraint C2 is extending constraint C1.

Listing 5.1 Tracts for the *UML2ER* case study.

```
-- C1: SRC_TRG_Package2ERModel
Package.allInstances->forAll(p|ERModel.allInstances
->one(e|p.name=e.name [▼]))

-- C2: C1 + Class2EntityType + Nesting
C1[▲] and p.ownedElements-> forAll(class|e.entities
```

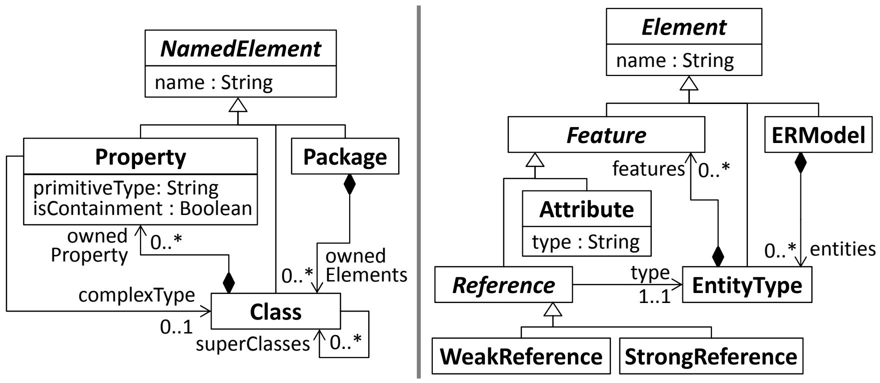


Fig. 5.4 The UML and ER metamodels.

```
->one(entity|entity.name=class.name [▼]))

-- C3: C2 + Property2Feature + Nesting
C2[▲] and class.ownedProperty->forAll(p|entity.
features->forAll(f|f.name=p.name))

-- C4: SRC_TRG_NamedElement2Element
NamedElement.allInstances->size=Element.allInstances->
size

-- C5: SRC_TRG_Package2ERModel
Package.allInstances->size=ERModel.allInstances->size

-- C6: SRC_TRG_Class2EntityType
Class.allInstances->size=EntityType.allInstances->size

-- C7: SRC_TRG_Property2Feature
Property.allInstances->size=Feature.allInstances->size

-- C8: C2 + Property2Attribute + Nesting
C2[▲] and class.ownedProperty->forAll(p|p.primitiveType
<> null implies entity.features->select(f|
f.oclIsTypeOf(Attribute))->one(f|f.name=p.name))

-- C9: C2 + Property2WeakReference + Nesting
C2[▲] and class.ownedProperty->forAll(p|p.complexType
<> null implies entity.features->select(f|f.oclIsTypeOf(
Reference))->one(f|f.name=p.name and p.isContainment
implies f.oclIsTypeOf(WeakReference)))

```

```
-- C10: C2 + Property2StrongReference + Nesting
C2[▲] and class.ownedProperty->forAll(p|p.complexType
<> null implies entity.features->select(f|f.oclIsTypeOf(
Reference))->one(f|f.name=p.name and not p.isContainment
implies f.oclIsTypeOf(StrongReference)))
```

The transformation (shown in Listing 5.2) contains eight rules, where three of them are abstract. There is a large number of inheritance relationships between the rules: $R8, R7 < R6$; $R6, R5 < R4$; $R4, R3, R2 < R1$.

Listing 5.2 *UML2ER* ATL Transformation.

```
module UML2ER;
create OUT : ER from IN : SimpleUML;

abstract rule NamedElement{ --R1
  from s : SimpleUML!NamedElement
  to   t : ER!Element(name <- s.name)
}

rule Package extends NamedElement{ --R2
  from s: SimpleUML!Package
  to   t: ER!ERModel(entities<-s.ownedElements)
}

rule Class extends NamedElement{ --R3
  from s: SimpleUML!Class
  to t: ER!EntityType(features<-s.ownedProperties)
}

abstract rule Property extends NamedElement{ --R4
  from s: SimpleUML!Property
  to   t: ER!Feature ()
}

rule Attributes extends Property{ --R5
  from s: SimpleUML!Property(
            not s.primitiveType.oclIsUndefined())
  to   t: ER!Attribute (type <- s.primitiveType)
}

abstract rule References extends Property{ --R6
  from s: SimpleUML!Property(
            not s.complexType.oclIsUndefined())
  to   t: ER!Reference (type <- s.complexType)
}
```

```

rule WeakReferences extends References{ --R7
  from s: SimpleUML!Property (not s.isContainment)
  to   t: ER!WeakReference
}

rule StrongReferences extends References{ --R8
  from s: SimpleUML!Property (s.isContainment)
  to   t: ER!StrongReference }
```

Tables 5.3 to 5.5 illustrate the corresponding matching tables for the transformation and the given Tracts (the reader should ignore for the moment the square brackets enclosing some numbers). Those cells without a number indicate there is no alignment between the constraint and the rule. The following subsection explains how the information in these matching tables is to be interpreted.

	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
C1	0.25	0.5						
C2	0.2	0.6	0.4			[*0.4*]		
C3	0.25		0.25	0.5	0.38	0.38	0.38	0.38
C4	1.0	1.0	0.75	0.5	0.5	1	0.5	0.5
C5	0.5	1.0						
C6	0.5	0.5	1.0			[*1.0*]		
C7	0.5		0.5	1.0	0.75	0.75	0.75	0.75
C8	0.25		0.17	0.33	0.5	[*0.25*]	[*0.25*]	[*0.25*]
C9	0.28		0.22	0.22	[*0.17*]	0.44	0.33	[*0.22*]
C10	0.28		0.22	0.22	[*0.17*]	0.44	[*0.33*]	0.22

Table 5.3 Matching table using CC metric.

5.1.5 UML2ER Case Study: Pragmatics

Recall that the purpose of the matching tables is to help find the rule(s) that caused the fault when a constraint is not satisfied. To show how these rules are located, let us suppose that we have executed the *UML2ER* transformation for a certain input model and checked the satisfaction of the constraints, something that can be done with our *TractsTool* [25] quite straightforwardly. Let us assume the outcome given by the tool is that constraint $C7$ is not

	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
C1	0.25	0.33						
C2	0.25	0.5	0.5			[*0.33*]		
C3	0.25		0.25	1.0	0.38	0.25	0.5	0.75
C4	0.5	0.33	0.38	0.5	0.25	0.33	0.33	0.5
C5	0.25	0.33						
C6	0.25	0.33	0.5			[*0.33*]		
C7	0.25		0.25	1.0	0.38	0.25	0.5	0.75
C8	0.38		0.25	1.0	0.75	[*0.25*]	[*0.5*]	[*0.75*]
C9	0.63		0.5	1.0	[*0.38*]	0.67	1.0	[*1.0*]
C10	0.63		0.5	1.0	[*0.38*]	0.67	[*1.0*]	1.0

Table 5.4 Matching table using RC metric.

	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
C1	0.13	0.25						
C2	0.11	0.38	0.29			[*0.22*]		
C3	0.13		0.14	0.5	0.21	0.17	0.25	0.3
C4	0.5	0.25	0.25	0.25	0.17	0.25	0.2	0.25
C5	0.17	0.33						
C6	0.17	0.33	0.5			[*0.33*]		
C7	0.17		0.2	1.0	0.3	0.21	0.38	0.5
C8	0.15		0.11	0.33	0.43	[*0.14*]	[*0.19*]	[*0.21*]
C9	0.19		0.18	0.22	[*0.13*]	0.36	0.33	[*0.2*]
C10	0.19		0.18	0.22	[*0.13*]	0.36	[*0.3*]	0.22

Table 5.5 Matching table using RCR metric.

satisfied. In Table 5.3 we can see that there is a complete coverage of $C7$ by rule $R4$ (as mentioned in Section 5.1.4, this table is to be consulted by rows). Consequently, it is very likely that the constraint fails due to this rule, so we should start by checking $R4$. Nevertheless, it does not always mean that $R4$ is the guilty rule. In fact, there are other candidate rules (all of them except $R2$, since the value in cell $C7/R2$ is 0) that could be the cause of the fault. Among them, $R5$, $R6$, $R7$ and $R8$ have the same CC value for constraint $C7$. In order to establish a priority order among these rules, we need to have a look at the RCR metric in Table 5.5. The higher the number in a cell in this table, the higher the priority for the rule to be guilty. The same thing occurs with $R1$ and $R3$, so we need to check their RCR metric for constraint

$C7$. After checking both tables, the error tracking process for constraint $C7$ should follow the sequence of rules: $R4$, $R8$, $R7$, $R5$, $R6$, $R3$, $R1$.

Metric RC can be used to check whether the constraints may offer a full coverage for the complete transformation or not. RC tables (e.g., Table 5.4) are to be consulted by columns: if all the values in a column are 0 or close to 0, it is very likely that the rule represented by such a column is not covered by any constraint. The RC metric is also useful for identifying the constraint that is more probably aligned with a certain rule. As with the CC metric, the higher the value in a cell is, the more likely the constraint represented by such a cell will cover the rule. When there is a draw in this table within a column, the corresponding cell in the RCR table should then be consulted.

5.1.6 Putting the Approach into Context

Once we have an approach to automatically locate the rules of a model transformation that may be the cause of a faulty behavior, it is very important to clarify how and where this approach fits in the overall process of a model transformation development [64]. As in the construction of any other software artifact, we should start with its specification. We believe that specifications should be defined in a modular and iterative manner: it is impossible to build the specifications of any complex system or artifact, and assume they will be complete, accurate and correct without testing them too. This is why in our context we use Tracts, because they allow the software engineer to focus on particular scenarios of special interest, and then build the specifications modularly and progressively.

In turn, the implementation of the model transformation can be built at the same time as the Tracts, or once they have been developed, depending on whether we want the specifications to guide the implementation or just to document its expected behavior. Although in theory the former approach is better, in practice implementations are developed at the same time as specifications (or even before), by different teams, and with different usages in mind (mostly analysis in the case of specifications, and execution in the case of implementations). This is particularly true in the case of specification methods that use precise and formal notations, and require specialized skills.

Once the specifications and the implementation are in place, the debugging process starts [67]. In our view, formal specifications and implementation should be debugged at the same time, assuming that both are complex artifacts and therefore potential subject to errors. The first step would be to discard as soon as possible all small mistakes (in one or the other) in a quick and cost-effective manner, something that can be done with the aid of the appropriate tools [153], before diving into more expensive and complex tests (such as model checking, formal validation, dynamic tests, etc.). And this is precisely where our approach represents a valuable asset.

The first step is to check, using the a-priori applicability test (Section 5.3.4), if our approach will work with the transformation. In the case it is amenable to be analyzed with it, it is a matter of building the matching tables with our toolkit.

The next step is to execute the transformation with the input models provided by the Tract test suites, using the TractsTool environment. In case a constraint is not fulfilled, our tool will provide the list of ATL rules that may have caused the faulty behavior, ordered according to the chances they have of being blamed. The developer can then look for errors on these rules, until one that can explain the constraint violation is found. But it may also be the case that the specifications are wrong, as it is often the case when they have not been tested before (cf. [143]). In any case, what we have now is a tool that is able to uncover, in a quick and easy manner, many of the errors that happen during the early stage of the testing process, and to help locate the rules that cause the faults.

This process will continue until the transformation works, respecting all the Tracts defined for it, which means that the implementation works for (at least) all the constraints and conditions that specify (at this level) its behavior. Then it will be the moment to start going through a more detailed and thorough testing phase, that will help uncover more subtle errors in the transformation—but at a most expensive cost, both time and resource-wise.

5.2 Implementation

In order to extract the footprints of constraints and rules, as well as to build the matching tables, having automation support is essential because this is a rather complex and error-prone task, especially in the case of large model transformations. Fig. 5.5 shows a UML activity diagram that depicts each step of the matching process until the matching tables are obtained.

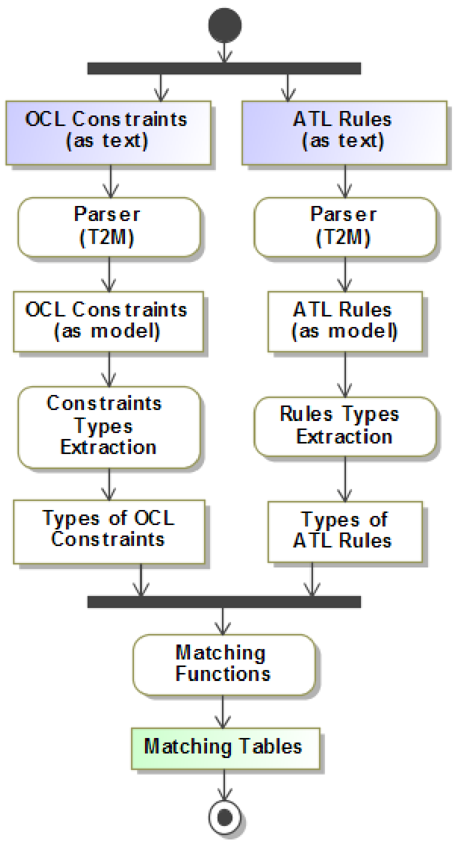


Fig. 5.5 Matching process.

5.2.1 Footprint Extraction from OCL Constraints

The first step is to extract the footprints for each OCL constraint. This is achieved by using the API of the USE (UML based Specification Environment) tool [115].

Firstly, we translate the input and output metamodels to the USE representation by means of a model-to-text transformation. As both the Ecore and the USE meta-metamodels are similar, the translation is quite straightforward. The relevant differences between both languages are the requirement that all relationships must be bidirectional in USE, and its lack of packages. Furthermore, USE only accepts one metamodel and one model, so we have to merge the input and output metamodels. This limitation implies the need to modify the name of each class and association in order to guarantee unique names. We have done so by adding a prefix to the name of the element: `src_` if it belongs to the source metamodel, and `trg_` if it belongs to the target metamodel.

Once both metamodels have been merged into a single file, we add to it the OCL expressions that compose the constraints and load the file into USE. For every OCL expression, USE builds a parse tree representing each subexpression with an explicit node which also provides the return type for each subexpression. To take advantage of this, we have built a small program that uses the aforementioned API. This API allows navigation through the parse tree and extracts the relevant information about the footprints, as explained in Section 5.1.3.

5.2.2 Footprint Extraction from ATL Rules

The first step in the footprints extraction is to inject the textual ATL transformation into a model-based representation. It is done automatically by means of a text-to-model transformation. The obtained model conforms to the ATL metamodel, which is in turn made up of three packages: ATL, OCL and PrimitiveTypes. Then, an ATL transformation (in fact, a so-called Higher-Order Transformation) takes the obtained model, as well as the input and target metamodels of the original transformation, and generates a model with information of the footprints used in each and every rule. We decided to implement a Higher-Order Transformation [135, 145] for extracting the footprints from the ATL rules, because the cost of building and maintaining two individual tools (one for ATL and one for OCL) was less than for developing one common tool.

Focusing on a rule, it is quite straightforward to obtain the footprints of the elements in the left-hand side (LHS, the input part) of the rules as well as those created in the righ-hand side (RHS, the output part). To do so, we need to navigate those objects of type InPattern, OutPattern and Binding of the ATL package[1].

The most challenging part is to extract the types from the OCL expressions. Contrarily to the OCL constraints in USE, ATL does not offer any support nor API to do the extraction. Furthermore, there are slight variations between the versions of OCL used by USE and by ATL concerning predefined types and operations and due to the fact that in ATL the OCL expressions allow references to variables which are bound by the rules. Although those variations do not affect the footprints, they make impossible to apply the same procedure for extracting the footprints from the OCL expressions in USE and the OCL expressions present in ATL. OCL expressions in ATL can be present in the filter part (of the LHS), local variables, the RHS and the imperative part. These textual expressions are built conforming to the OCL package[2] of the ATL metamodel. The extraction of the types in the OCL expressions is a three-step process. In the first step, we only need information of the ATL transformation (expressed as a model, as explained before), while in the second and third steps we need information of the source and target metamodels of the transformation in order to be able to navigate them. An OCL expression can be made up of iterators (in a model level, they are objects of type IteratorExp), such us collect and select. The first step of the footprints extraction consists of taking every OCL expression and removing the iterators. When doing so, from

[1]A snapshot of the ATL package is available from http://atenea.lcc.uma.es/Descargas/ATL.png (the references to the OCL package are not displayed)

[2]A snapshot of the OCL package is available from http://atenea.lcc.uma.es/Descargas/OCL.png (the references to the ATL package are not displayed)

each OCL expression (that may contain iterators), one or more navigation paths are obtained.

5.2.3 Matching Function

Once we have the types and features used in the constraints (`C`) and the rules (`R`), we apply the matching functions to obtain the measures explained in Section 5.1.4. Algorithm 1 shows the function `intersectionSubtypes` that computes $C_i \cap R_j$ considering subtyping. Given it, Algorithms 2, 3 and 4 present the computation of the values $CC_{i,j}$, $RC_{i,j}$, $RCR_{i,j}$ corresponding to the three metrics. These functions have been implemented in Java. The output of the computation for every pair $[C_i, R_j]$ is represented in a *csv* (*comma-separated value*) format, so that it can be read by spreadsheet-based applications.

```
Input: C, R
Output: v
v = 0 // Find full matches
for c ∈ C do
    if R.contains(c) then
        v = v + 1
        R.remove(c)
    end
end
// Find sub-/supertype matches
for c ∈ C do
    subSuperType = R.containsAny(subSuperType(c))
    if subSuperType <> null then
        v = v + weight(c, subSuperType)
        R.remove(subSuperType)
    end
end
return v
```

Algorithm 1: Function that computes $C_i \cap R_j$

```
Input: C, R
Output: vCC
vCC = intersectionSubtypes(C, R) / size(C)
vRCR = intersectionSubtypes(C, R) / union(C, R)
if vCC > threshold and vRCR > threshold then
    return vCC
end
else
    return 0
end
```

Algorithm 2: Function that computes the CC metric for C_i and R_j

```
Input: C, R
Output: vRC
vRC = intersectionSubtypes(C, R) / size(R)
vRCR = intersectionSubtypes(C, R) / union(C, R)
if vRC > threshold and vRCR > threshold then
    return vRC
end
else
    return 0
end
```

Algorithm 3: Function that computes the RC metric for C_i and R_j

5.3 Evaluation

In this section, we discuss the accuracy and limitations of our approach, and introduce a method for checking if a transformation is amenable to be used with it, based on the concept of *footprint similarity matrix.* To evaluate the accuracy of our approach we performed a case study [90] by following the guidelines for conducting empirical explanatory case studies by Roneson and Hörst [119]. In particular, we report on applying our approach to detect the alignments between Tracts and ATL transformations for four different transformation projects. In addition, we also present the results of a controlled experiment for locating faults in faulty transformations by applying mutations to the four different transformation projects.

```
Input: C, R
Output: vRCR
vRCR = intersectionSubtypes(C, R) / size(union(C, R))
if vRCR > threshold then
    return vRCR
end
else
    return 0
end
```

Algorithm 4: Function that computes the RCR metric for C_i and R_j

5.3.1 Research Questions

The study was performed to quantitatively assess the completeness, correctness, and usefulness of our approach when applied to a real-world scenario. More specifically, we aimed to answer the following research questions (RQs):

1. *RQ1—Correctness*: Are the detected alignments between constraints and rules correct in the sense that all reported alignments are representing real alignments? If our approach reports incorrect alignments, what is the reason for this?

2. *RQ2—Completeness*: Are the detected alignments complete in the sense that all expected alignments are correctly detected? If the set of detected alignments is incomplete, what is the reason for missed alignments?

3. *RQ3—Usefulness*: In those cases where more than one alignment is reported for a constraint or a rule, are the correctly identified alignments outperforming the falsely identified alignments in terms of the calculated similarity value? We provide this additional question, because the first two questions only consider the evaluation of alignments as true/false, but they do not take the weights of the alignments into account.

5.3.2 Case Study Design

Before we present the results of our case study, let us elaborate on its design.

Requirements

As appropriate inputs we require transformation projects that consist of a set of constraints and a set of rules. We also need the source and target metamodels in order to extract the footprints of constraints and rules. Apart from these artifacts, we further require the alignments between the constraints and the rules given by transformation engineers; otherwise, we would not be able to compare the results obtained by our approach with the expected correct set of alignments. To accomplish an appropriate coverage of different scenarios, the transformations should comprise different intrinsic properties, e.g., having different design complexity measures.

Setup

We analyzed the alignments between transformation requirements and implementations in four different real-world transformation projects.

First, and as already presented in Section 5.1.4, we selected the transformation project dealing with the generation of Entity Relationship (ER) Diagrams from UML Class Diagram Models (*UML2ER* for short).

Second, we selected a transformation project that deals with behavioral models. Models conforming to CPL (Call Processing Language) [92] are transformed into models conforming to SPL (Session Processing Language) [28]. The *CPL2SPL* transformation [76] is a relatively complex example available from the ATL zoo [61].

Third, we considered a model transformation project that does not operate on modeling languages but rather on markup languages. More specifically, we considered the *BT2DB* transformation of BibTeX documents into DocBook documents, also available from the ATL zoo. BibTeXML is an XML-based format for the BibTeX bibliographic tool. DocBook, in turn, is an XML-based format for document composition.

Finally, we experimented with a very large transformation called Ecore2Maude (or *E2M* for short) which is used by a tool called e-Motions [116]. It converts models conforming to the Ecore metamodel into models that conform to the

Metric	***UML2ER***	***CPL2SPL***	***BT2DB***	***Ecore2Maude***
ATL LoC	77	348	286	1397
#Elements	86	497	449	2403
#Links	201	1114	1052	5270
#Rules	8	15	9	40
#Helpers	0	6	4	40
#Bindings	5	73	25	329

Table 5.6 Transformation Metrics Overview.

Metric	***UML***	***ER***	***CPL***	***SPL***	***BT***	***DB***	***Ecore***	***Maude***
#Class	4	8	31	77	21	8	18	45
#Atts	3	1	42	33	10	1	31	17
#Refs	4	2	16	62	2	5	34	46
#Inhs	3	6	32	76	31	4	16	38

Table 5.7 Metamodel Metrics Overview.

Maude [36] metamodel, in order to apply some formal reasoning on them afterwards.

Tables 5.6 and 5.7 summarize the main size metrics for the ATL transformations and the corresponding metamodels.

We developed the Tracts for the given transformations. Constraints were written by a member of our team who was familiar with OCL but was unaware of the ATL implementations. They have been written based on the natural language specification of the transformations. For example, the *UML2ER* case study comprises 10 constraints (previously shown in Listing 5.1) of two different kinds: one for comparing the number of instances of certain source and target classes, and one for checking equivalent elements based on containment relationships and value correspondences. There are 16 constraints in the *CPL2SPL* case study, checking that the proper object types in SPL are created from specific object types in CPL. Furthermore, they check that the number of objects in the target model is correct, and that the URIs are correctly created. The 16 constraints in the *BT2DB* case study make sure that the proper book is created for the different possible entries in BibTeX, and that all entries are properly transformed. Finally, for the *E2M* case study, three kinds of constraints have been developed to check that the number of elements in the output model is correct, that the `Operation` entities in the output model have been created from the appropriate input elements, and that from each `Class` entity, the corresponding `Sort` has been created in the target model.

The input data including the Tracts constraints, the ATL transformations, the alignments between them, the results and the accuracy of these four projects (and several others) are available on our project's website [23].

Measures

To assess the accuracy of our approach, we compute the *precision* and *recall* measures originally defined in the area of information retrieval [97]. In the context of our study, precision denotes the fraction of *correctly detected* alignments among the set of *all detected* alignments (i.e., how many detected alignments are in fact correct). Recall indicates the fraction of *correctly detected* alignments among the set of *all actually occurring* alignments (i.e., how many alignments have not been missed). These two measures may also be thought of as probabilities: the precision denotes the probability that a detected alignment is correct and the recall is the probability that an actually occurring alignment is detected. Thus, both values range from 0 to 1.

Precision is used to answer RQ1 and recall to answer RQ2. There is a natural trade-off between precision and recall. Thus, these two metrics may be further combined inside the so-called *f-measure* to avoid having only isolated views on both aspects [97]. To answer RQ3, we use the *utility-average* metric, which serves to reason about the relative difference between false positives and true positives for one row (in the CC and RCR tables) or for one column (in the RC and RCR tables).

To check whether or not our approach is accurate for a given model transformation and a given set of constraints, we have manually obtained the alignments between rules and constraints, reflected in a table called *expected alignment table*. An example is shown in Table 5.8 for the UML2ER

	R1	R2	R3	R4	R5	R6	R7	R8
C1	(×)	×						
C2	(×)	×	×					
C3	×		×	×	(×)	(×)	(×)	(×)
C4	×	(×)	(×)	(×)	(×)	(×)	(×)	(×)
C5	(×)	×						
C6	(×)	×	×					
C7	(×)		×	×	(×)	(×)	(×)	(×)
C8	(×)		×	(×)	×			
C9	(×)		×	(×)		(×)	×	
C10	(×)		×	(×)		(×)		×

Table 5.8 Expected alignments for the *UML2ER* transformation.

transformation where "×" means that there is a direct relation and "(×)" means that there is a relation via inheritance. There is a cross mark, ×, in the cells where there is a direct alignment between constraints and rules, and a cross mark in brackets, (×), when the alignment is due to inheritance relationships between meta-classes or transformation rules (cf. Section 5.1.4). The value of empty cells is 0.

For computing precision and recall, we extract the true-positive values (TPs), false-positive values (FPs) and false-negative values (FNs), with the help of the expected alignment table. A cell contains a TP when *(i)* its value is above the threshold, *(ii)* there is an alignment in the expected alignment table, and *(iii)* the alignment is also identified in the RCR table for the same cell (in the case of CC and RC tables, see Section 5.1.4). There is an FP when our approach identifies that there is an alignment (CC/RC and RCR cell values above the threshold), but the expected alignment table does not indicate so. Finally, there is an FN between a constraint and a rule when our approach identifies that there is no alignment between them and there is a mark in the equivalent cell in the expected alignment table.

From the TP, FP and FN values we compute the *precision*, *recall* and *f-measure* metrics as follows:

$$precision = \frac{TP}{TP + FP} \tag{5.4}$$

$$recall = \frac{TP}{TP + FN} \tag{5.5}$$

$$f\text{-}measure = 2 \times \frac{precision \times recall}{precision + recall} \tag{5.6}$$

The utility-average metric permits reasoning about the relative value difference between FPs and TPs. For example, if there are five alignments in a row in the CC table and four of them are falsely created (which means that there is only one TP and four FPs), but the TP has the highest value, then the four FPs are disregarded because the TP is the first one checked. We have calculated this metric by rows for the CC metric and by columns for the RC metric. The result is the mean of the values obtained in each row/column. As for the RCR metric, since it can be consulted by columns or by rows, we have considered both situations. The utility-average metric, UAM, is computed as follows.

$$UAM = \frac{\sum_{i=1}^{n} u_i}{n} \tag{5.7}$$

where $u_i = 1$ if there are neither FNs nor FPs in the row/column, or there are no FNs and the value of all FPs is less than the value of the TPs; $u_i = 1 - \frac{|F|}{|F|+|TP|}$ if there are no FNs but there are FPs which are bigger than or equal to at least one of the TPs in the row/column (in the formula, TP is the set of all true positives in the row/column, and $F = \{x \in \text{FP} \mid \exists y \in \text{TP}$ with $x >= y\}$); finally, $u_i = 0$ if there are FNs in the row/column.

5.3.3 Results

We now present the results of applying our approach to the four different model transformation projects. A summary of these results is shown in Table 5.9. Detailed results can be found on our project's website. In the matching tables (e.g., Tables 5.3 to 5.5), TPs are shown in normal font, FPs within square brackets, and FNs within curly brackets. These values are obtained by comparing the *expected alignment* tables for the four projects, with the matching tables obtained by our approach.

Metric	UML2ER	CPL2SPL	BT2DB	Ecore2Maude
TPs	46	37	29	11
FPs	9	9	85	3
TNs	-	1	3	-
Precision	0.84	0.80	0.25	0.79
Recall	1.00	0.97	0.91	1.00
F-measure	0.91	0.88	0.40	0.88
Utility average	0.80	0.81	0.60	0.94

Table 5.9 Accuracy of case studies.

As shown in Table 5.9, the values obtained for the precision, recall and f-measure metrics are acceptable in three of the projects: *UML2ER*, *CPL2SPL* and *Ecore2Maude*. With these accuracy results, we can conclude that our approach works well with these projects, since the alignments found statically are quite reliable. Recall is acceptable in all projects, because the number of FNs is low. However, the number of FPs is very high in the *BT2DB* project, resulting in poor precision (0.25). The reasons for this low performance are discussed in next section.

5.3.4 A-priori Applicability Test

After carefully studying the model transformation that scored a low precision of our approach, we discovered that the footprints of its rules were very similar, i.e., they shared many types and features. This led us to introduce a new measure, based on the concept of *footprint similarity matrix* for model transformation rules. A similarity matrix gives us an indication of how rules are related with each other, i.e., the factor of common types/features they share.

Similarity Matrixes

Similarity matrixes have rules in both columns and rows and are consequently symmetric. Thus, we are only interested in half the matrix (excluding the main diagonal, of course). To calculate the fitness for the transformation, we

	R8	**R7**	**R6**	**R5**	**R4**	**R3**	**R2**	**R1**
R1	0	0	0	0	0	0	0	1
R2	0	0	0.2	0	0	0.25	1	
R3	0	0	0.25	0	0.2	1		
R4	0.33	0.25	0.14	0.2	1			
R5	0.2	0.17	0.11	1				
R6	0.14	0.13	1					
R7	0.25	1						
R8	1							

Table 5.10 Similarity Matrix for the Rules in *UML2ER*.

extract the average and standard deviation of half the table. The lower both values are, the fewer footprints rules have in common, and thus, the higher the chance for a successful application of our approach is. We recommend to apply our approach on model transformations where the mean and standard deviation of the similarity matrix for rules are below 0.15. Otherwise, the accuracy of the results is not good enough (precision would be normally below 0.7).

Table 5.10 presents the similarity matrix for the *UML2ER* case study. Both metrics, the mean and standard deviation, have a value of 0.1 which means that rules are *separated enough*, and thus our approach works well because there is no confusion possible when establishing the alignments between the constraints and the rules.

If we compute the similarity matrixes for our other case studies we obtain that, for instance, for the *CPL2SPL* case study, the mean of the similarity matrix is 0.08, and the standard deviation is 0.1. In this case, the precision of the results is 0.8. However, the *BT2DB* transformation shows quite different values. The mean is 0.41 and the standard deviation is 0.24. Consequently, it is difficult to distinguish among them when looking for the “guilty rule”, and this results in the occurrence of many false positives in the matching tables. If we look at the ATL transformation, we find the explanation for such a high value. Since the target metamodel is rather small, many rules create objects of the same target types. For example, 8 rules out of 9 create `Paragraph` elements, and 33% of the rules contain a `TitleEntry` element in their input part.

We have automated the process for obtaining the footprints of any ATL transformation, as well as the computation of the similarity matrixes. With this, we have obtained the similarity matrixes for the transformations in the ATL zoo, in order to investigate the applicability of our approach. A summary of these results is available in Table 5.11 where every row represents a case study. Apart from the mean and deviation, for each case study, we show whether our approach is advisable to be applied (✓) or not (×). All similarity matrixes obtained, as well as the software that computes them, are available in Appendix A and on our project's website. Out of the 41 model transformations studied, the mean and standard deviation turned out to be below 0.15 in 21 of them, which means that our approach is perfectly fit for use with around half of the transformations. The threshold that we used for the mean and the standard deviation of the similarity matrix, 0.15, is to ensure that precision is above 0.8.

It is important to note that this fitness test ensures good results (since the transformation rules are separated enough to be distinguishable by our proposed approach), but it may be that the fitness test scores low and still our approach works well because of the way in which the constraints are written. In any case, there is no guarantee that our approach is fit for use when the applicability test provides results below 0.15.

We also discovered that the number of rules in the transformations has no impact in the applicability of our approach. In fact, the number of rules used in the set of transformations studied ranged from 3 up to 40. As an example, the similarity matrix of a small transformation (*PetriNet2PathExp*, 3 rules) gave bad results, while the one obtained from the largest transformation (*Ecore2Maude*, 40 rules) gave good results. Contrarily, we obtained adverse results for another large transformation (*R2ML2XML*, 55 rules), while we got good results for small transformations (such as *PetriNet2Grafcet*, 5 rules). We have applied the *Pearson correlation coefficient*, a measure of the linear correlation between two variables, on the results, when the first variable is the number of rules in the transformations and the second is the mean obtained

Transformation	**No. of Rules**	**Mean**	**Deviation**	**Advisable**
ATL2Problem	18	0.35	0.13	×
ATOM2RSS	3	0.05	0.05	✓
ATOM2XML	8	0.33	0.08	×
BibTeX2DocBook	9	0.41	0.24	×
CPL2SPL	16	0.07	0.13	✓
Ecore2USE	14	0.1	0.17	✓
Grafcet2PetriNet	5	0.14	0.07	✓
HTML2XML	30	0.21	0.11	×
IEEE14712MoDAF	13	0.03	0.05	✓
KM32OWL	16	0.13	0.14	✓
KM32Problem	16	0.44	0.15	×
Measure2Table	6	0.31	0.37	×
Measure2XHTML	22	0.07	0.11	✓
MySQL2KM3	11	0.2	0.29	×
OCL2R2ML	37	0.09	0.12	✓
OWL2XML	24	0.51	0.16	×
PathExp2PetriNet	3	0.15	0.04	✓
PathExp2TextualPath	5	0.37	0.44	×
PetriNet2Grafcet	5	0.14	0.07	✓
PetriNet2PathExp	3	0.28	0.11	×
PetriNet2PNML	4	0.17	0.05	×
PetriNet2XML	5	0.54	0.11	×
PNML2PetriNet	5	0.28	0.12	×
PNML2XML	4	0.72	0.17	×
R2ML2RDM	69	0.11	0.14	✓
R2ML2XML	55	0.26	0.13	×
R2ML2WSDL	14	0.07	0.14	✓
RDM2R2ML	56	0.1	0.13	✓
RDM2XML	39	0.32	0.14	×
RSS2ATOM	3	0.05	0.05	✓
RSS2XML	4	0.37	0.15	×
UML2ER	8	0.09	0.11	✓
WSDL2R2ML	17	0.06	0.11	✓
WSDL2XML	20	0.36	0.15	×
XML2ATOM	10	0.15	0.06	✓
XML2MySQL	6	0.12	0.1	✓
XML2PetriNet	5	0.29	0.06	×
XML2PNML	5	0.25	0.19	×
XML2RSS	9	0.14	0.07	✓
XML2WSDL	19	0.14	0.08	✓
XSLT2XQuery	7	0.07	0.14	✓

Table 5.11 Summary of Similarity Matrixes.

from the similarity matrixes. The obtained value was -0.13, meaning that this dependence is minimal.

Concept	Mutation Operators	Concept	Mutation Operators
Matched Rule	Addition Deletion Name Change	Filter	Addition Deletion Condition Change
In/Out Pattern Element	Addition Deletion Type Change Name Change	Binding	Addition Deletion Feature Change Value Change

Table 5.12 Possible Mutations for ATL Transformations (from [13]).

5.3.5 Experimenting with Faulty Transformations

So far, we have illustrated our approach with correct model transformations. However, given that it has been devised to detect errors in faulty transformations, it is essential to test its effectiveness when the transformations are indeed faulty.

Setup. For this reason we have used mutation analysis [74] to systematically inject faults into model transformations [101], and then used our approach to locate the bugs. The purpose of a mutated transformation is to emulate a transformation that contains bugs, and then see if our approach detects them.

To define the possible mutations of ATL transformations, we use the list of transformation change types presented in [13], which are summarized in Table 5.12. For more information on the precise mutations and the results obtained for the case studies presented in this paper we kindly refer to [140].

Example. As an example, we have applied the following mutations for the *CPL2SPL* transformation mentioned above:

1. Addition of an OutPatternElement in $R1$, which results in the creation of unexpected additional elements in the target model.

2. Modification of the feature of a binding in $R3$, resulting in incorrectly initialized features in the target model.

Mutation	Constraints Violated	Guilty Rule Located?	Number of Steps
CPL2SPL_1	C1	✓	1
	C2	✓	1
	C3	✗	-
	C11	✓	1
CPL2SPL_2	C4	✓	1
CPL2SPL_3	C5	✓	1
	C6	✓	1
	C14	✓	1
CPL2SPL_4	C12	✓	1
CPL2SPL_5	C15	✓	2
CPL2SPL_6	C5	✓	3
	C13	✓	3
CPL2SPL_7	C10	✓	1

Table 5.13 Summary of mutations and fault localization results (CPL2SPL project).

3. Modification of the condition of the filter in $R5$, changing the amount of produced target model elements.

4. Modification of a binding and addition of OutPatternElement in $R6$, thus producing more target model elements.

5. Deletion of a binding and an OutPatternElement, along with its binding, in $R8$; emulating the circumstance in which a transformation produces not enough target elements.

6. Addition of a filter in $R9$, making the application of the rule more restricted, thus creating less elements in the target model.

7. Feature modification in a binding and deletion of a binding in $R11$, resulting in wrongly assigned values and missing values in the target model.

Measures For each mutation, we collect: *(i)* the constraints violated when the mutation is applied; *(ii)* if the user was able to find the guilty rule using our approach; and *(iii)* the number of steps needed for finding the guilty rule. By number of steps we mean the number of rules that the user needs to check in order to find the one that was mutated (including that one).

Results. The results in Table 5.13 show that all mutations were detected by our approach for the given example. Each mutation caused one or more constraints to fail, and the guilty rule was correctly identified for all constraints but one ($C3$). This happened because of false negatives, given that the relation between rule *CPL2SPL_1* and constraint $C3$ was quite loose. However, the mutation caused several constraints to fail and our approach was able to identify the mutated rule in the rest of the cases, so the guilty rule was eventually identified.

The overall results obtained for all four projects, described in our technical report [140], show similar effectiveness. We injected a total of 21 mutations, causing 48 constraints to fail. All mutants were killed, i.e., all guilty rules were correctly identified by our approach. Only for three constraints that failed we could not identify the rule causing it but, in all cases, these rules caused the violation of several constraints, and the guilty rule was already identified as the one responsible for the violation of a different constraint that failed with the same mutation, such is the case with $C3$ in *CPL2SPL_1*, so the guilty rule was eventually identified. Regarding how many rules need to be checked before identifying the guilty one, our proposed approach needed an average of 1.78 rules to be checked.

5.3.6 Threats to Validity

In this subsection, we elaborate on several factors that may jeopardize the validity of our results.

Internal validity—Are there factors which might affect the results of this case study? The quality of the data appearing in the matching tables, as well as the usefulness and accuracy of these, are crucial for the internal validity due to three main factors. First, the Tracts need to be manually defined. If they do not contain valuable restrictions, then the matching tables are not useful. Defining constraints is not a trivial task, and the person responsible for doing so needs to have knowledge of OCL, of the transformation to check, and of what should be checked.

Second, the way in which footprints are extracted is crucial for building the tables. As explained in Section 5.1.3, there may be very long navigation paths expressed in OCL both in the Tracts and in the rules. From them, we extract the types and features discarding some elements because they are not considered as relevant by giving a higher priority to the results than to the paths used in the computations.

Third, in order to study the accuracy of our tables, we have manually defined the expected alignment tables. Should we have failed to properly identify these alignments, the value of precision and recall would have been incorrectly calculated. In any case, they were written by a member of the team and double-checked by another, in order to minimize this risk. We have also made some assumptions in the implementation of our approach. For instance, we have chosen 0.1 as the threshold value for considering alignments relevant, as mentioned in Section 5.1.4. We also decided not to take constants and primitive types into account (Section 5.1.3). Although our experiences have shown that these decisions seem to be correct, they need to be further validated with more experiments and case studies.

Fourth, different styles of Tracts definition may have an effect on the outcomes. As mentioned in Section 5.3.2, the Tracts constraints were written by a member of our team. Of course, should they had been written by other people, or by the developers of the transformations themselves, the results presented here may have been slightly different. Here we assumed the underlying hypothesis that the constraints and rules are more heterogenous if they are developed by different persons, thus resulting in a more difficult matching problem.

Finally, concerning the experiment with faulty transformations, we relied on the state-of-the-art of mutation operators for model transformations, but further operators may be required in the future to deal with more fine-grained OCL expression mutations. Thus, these additional operators may have an impact on the results gained in our experiments.

External validity—To what extent is it possible to generalize the findings? As a proof of concept of our approach, we have extracted the matching tables for model transformations written in the ATL language. The metamodel of ATL comprises, amongst others, a package for OCL. Currently, the footprint extraction operates on this representation, and thus, it works only for ATL transformations. Nevertheless, it would be possible to reuse parts of the ATL footprint extraction for other rule-based transformation languages that also integrate OCL as a sublanguage. Another threat to external validity would be considering further features of model transformations, such as reflection [85]. Finally, our studies are focussing for out-place transformation scenarios, and thus, additional studies are needed for in-place transformation scenarios. As part of our future work we plan to investigate these issues, and also try to define a minimal set of requirements on the kinds of specification notations and implementation languages which are amenable to be directly addressed by our approach.

5.4 Related Work

The need for systematic verification of model transformations has been documented by the research community by several publications outlining the challenges to be tackled [7, 8, 47, 132]. As a response, a plethora of approaches ranging from lightweight certification to full verification have been proposed to reason about different kinds of properties of M2M transformations [1, 143].

With respect to the contribution of this chapter, three threads of related work are discussed: *(i)* general traceability approaches in software engineering as well as specific approaches for tracking "guilty" transformation rules, i.e., those whose behavior violates the transformation specifications, *(ii)* approaches for generating test cases for model transformations, and *(iii)* approaches that build on model footprints as does our approach.

5.4.1 Tracing Faults in Model Transformations

IEEE [71] defines traceability as the degree to which a relationship between two or more artifacts can be established. Most tracing approaches are dedicated to establishing traceability links between artifacts that are in a predecessor/successor relationship with respect to their creation time in the software development process, e.g., between requirements, features, design, architecture, and code. Our approach for automatically finding the alignments between constraints and transformation rules is in the spirit of traceability rules as presented in [112, 111]. A survey dedicated to traceability in the field of MDE is presented in [49], where the possibilities of using trace links established by model transformations are discussed. However, this survey does not report on tracing approaches between transformation specifications and implementations.

Tracking guilty transformation rules using a dynamic approach, i.e., by executing the model transformation under testing, has been subject to investigations. Hibberd et al. [67] present forensic debugging techniques for model transformations based on the trace information of model transformation executions for determining the relationship between source elements, target elements, and the transformation logic involved. With the help of such trace information, it is possible to answer debugging questions implemented as queries. In [150], we used OCL-based queries for the backwards debugging of model transformations using an explicit runtime model based on the trace model between the source and target models. Aranega et al. [3] present an approach for locating transformations errors by also exploiting the traces between the source and target models. The dynamic approach is also used in [142] to build slices of model transformations and in [60] following a white-box testing approach. A complementary approach to model transformation testing has been proposed by Kessentini et al. [78], using a generic oracle function. The idea of this approach is that the traces between the source and target models of a transformation should be similar to existing example traces. Specifically, the oracle function checks how large a derivation there is of the generated traces of a model transformation from existing traces in the example base. While all these approaches track transformation rules using specific test input models, our aim is to statically build more general traceability models between transformations' specifications and their implementations for enabling static analysis (the pros and cons of dynamic vs. static approaches have already been discussed in Section 5.1.1).

In addition to Tracts, other approaches have been proposed that build on the notion of transformation contracts to specify transformation specifications [143]. While other OCL-based specification approaches, e.g., [32], are obviously supported by the approach presented in this paper, for non OCL-based approaches, e.g., [65], additional transformations for computing the metamodel footprints may be developed or these specifications may be internally translated to OCL to reuse the existing footprint computation. Analogously, if other transformation implementation languages such as RubyTL [122], ETL [82], or QVT [OMG] need to be supported, additional higher-order transformations like those for ATL need to be developed.

There are some other transformation testing approaches that directly annotate assertions inside transformation implementations [53, 34]. Thus, these approaches have no need to compute the alignments between the specification and the implementation, as they are already provided by the transformation engineer. However, the specification and implementation of the transformation is intermingled, and thus, specifications are specific to a certain transformation implementation.

There are several approaches that define contracts for model transformations by defining a set of input/output model pairs and employing model comparison techniques to look for differences between the expected output models (provided by the engineer) and the actual outputs of the transformation [94, 50]. In this context, basic support for a failure trace is provided, since the different elements (added, updated, and deleted elements) between an actual target model and an expected target model may be calculated, but the tracing to the corresponding source model elements as well as to the transformation rules is left open.

5.4.2 Test Generation for Model Transformations

For tracking guilty rules, the availability of appropriate test input models is assumed in our approach. In [57] we proposed a technique for test case generation. Nevertheless, we give an overview of the research efforts that have been investigated in this area so far. They include black-box, gray-box and white-box approaches.

Küster et al. [88], Gonzalez and Cabot [60], and Sánchez Cuadrado et al. [123] focus on white-box methods. In the former, the existence of a high-level design of model transformations, consisting of conceptual transformation rules, is assumed. In [60], a white-box based testing approach for ATL transformations is provided by extracting OCL constraints and using a model finder to compute test input models fulfilling certain path conditions. Finally, Sánchez Cuadrado et al. discuss the generation of test input models for confirming and explaining errors reported by a static checker for ATL transformations.

Many approaches have been proposed for black-box testing, whereby test source models are generated either on the basis of the source metamodel (e.g. [18, 44, 128]) or on the basis of specified requirements [53, 62]. For the actual test source model generation, most of these approaches rely on constraint satisfaction, e.g., by means of SAT solvers. Furthermore, an approach has been proposed, which allows automatically completing test input models, i.e., the transformation engineer has to specify an intention by defining a model fragment only, and an algorithm complements this fragment for a valid test input model [129].

5.4.3 Model Transformation Footprinting

Recently, some approaches for computing and utilizing model footprints have been presented. In [73], the footprints of model operations are statically computed by introducing the idea of metamodel footprints. We pursue this idea of computing metamodel footprints from transformation specifications and implementations for establishing traceability links instead of reasoning solely on model footprints. Mottu et al. [103] compute the input metamodel footprints for ATL transformations in order to slice the input metamodels as a prerequisite step for computing test input models for the transformations being studied with Alloy. Compared to our work, the work of Mottu et al. is orthogonal in the sense that their approach could complement ours. While we focus on fault localization, Mottu et al. are concerned with test model generation.

5.5 Summary

In this chapter we have presented a static approach to trace errors in model transformations. Taking as input elements an ATL model transformation and a set of constraints that specify its expected behavior, our approach automatically extracts the footprints of both artifacts and compares transformation rules and constraints one by one, obtaining the overlap of common footprints. Subsequently, it returns three matching tables where the alignments between rules and constraints are recorded. By using these tables, the transformation engineer is able to trace the rules that can be the cause of broken constraints due to faulty behavior.

Our evaluation shows that the presented approach is expected to be accurate for a large set of model transformations. By using the similarity matrixes, an automated and instant fitness test is available to check a-priori whether the approach will be helpful for a given transformation. Several executables of our approach are available on our website [23].

Chapter 6
Extending Tracts for Model-to-Text and Text-to-Model Transformations

Much effort has been put into the establishment of model-to-model (M2M) transformation testing techniques in the past years [1, 143]. As we have mentioned in the previous chapter, several approaches have been developed for defining contracts for M2M transformations that act as specifications for model transformation implementations [32, 56], as oracle functions to validate the output of transformations [56, 63], and as drivers for generating test cases [63]. In particular, constraints for input models, output models and for the relationship between both may be specified.

Besides M2M transformations, model-to-text (M2T) and text-to-model (T2M) transformations are of major importance in Model-Driven Engineering [39]. M2T transformations are typically used to bridge the gap between modeling languages and programming languages by defining code generators but may be employed in a generic manner to produce text from models such as documentation or textual representations of a model's content. T2M transformations are typically used for reverse engineering [20], e.g., transforming legacy applications to models in the case of model-driven software modernization. However, these kinds of transformations have not gained much attention when it comes to testing.

In this chapter we adopt current techniques for testing M2M transformations to the problem of testing T2M and M2T transformations. The prerequisite of using existing M2M transformation techniques is to have metamodels for the input and output of the transformations. However, for the side that is dealing with "just" text, no metamodels are usually available. Even more problematic, when considering T2M and M2T transformations, a set of metamodels and T2M parsers may be required as a prerequisite. For instance, consider Web applications where in addition to a general purpose programming language several other languages may be employed where some of the languages are even embeddable in other languages. Thus, developing metamodels and T2M parser support for such complex settings may introduce a huge overhead.

To alleviate the burden from T2M and M2T transformation developers, we introduce a generic approach that may be used for any transformation task where text is involved as input or output of the transformations. The main mechanism we employ is to represent text within a generic metamodel in order to transform M2T and T2M transformation specification problems into equivalent M2M transformation specification problems. The proposal is combinable with any contract-based M2M transformation approach, but in this chapter we demonstrate its application with Tracts [56].

The structure of this chapter is as follows. Section 6.1 shows how to represent text-based artifacts as models to allow for reusing the M2M transformation testing approaches, demonstrates how Tracts are defined for M2T and T2M transformations and gives details about the implementation of the approach. Section 6.2 presents an evaluation of the approach, in particular to explore its capabilities to find shortcomings in code generations delivered by current UML tools. In Section 6.3 we present related work and Section 6.4 summarizes this chapter.

6.1 Generic Metamodel for Representing Text Repositories

In order to reuse M2M transformation specification and testing approaches, we have to transform the M2T or T2M transformation specification problem into a M2M transformation specification problem. For this, the text artifacts residing in the input or output domain of the transformations under study have to be injected to the model engineering technical space [86].

For realizing this goal, there are several options. We may either decide to opt for a specific format conforming to a specific grammar or to use a generic format that is able to represent any text-based artifact. In case there is already a metamodel available for the specific grammar, then this metamodel may be a good choice anyway. However, for most transformation scenarios involving text at one side there are no metamodels available, because metamodels are often not required at all. Just consider the case of generating documentation from models. Although there is no generalized and fixed structure, it may be necessary to check certain requirements of the transformation. This is why we have decided to use the second option, which allows us to save upfront the effort required when developing M2T or T2M transformations in general. Furthermore, using a generic metamodel to represent the text artifacts also reflects best practices in the development of M2T transformations, where no metamodel is used for the text artifacts. For example, consider template-based M2T transformation languages [106]. Usually, template-based approaches are used to generate that text. Finally, even if there is a T2M parser, this is again a transformation that may have to be specified and tested. Thus, our generic approach may be used to test the specific approach.

Apart from this, there is a second aspect that needs to be considered when dealing with text-based artifacts. The artifacts are normally organized in a hierarchical folder structure, which should be taken into account. For instance, the output of a M2T transformation may not be just a single file but several, which should be also arranged in a certain folder structure. Thus,

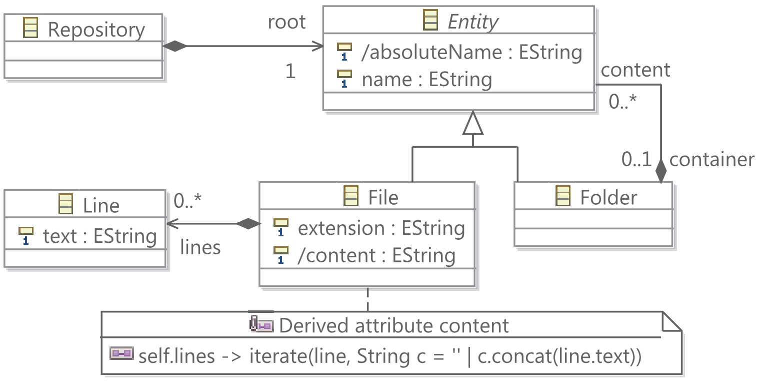


Fig. 6.1 Metamodel for representing text artifacts and repositories.

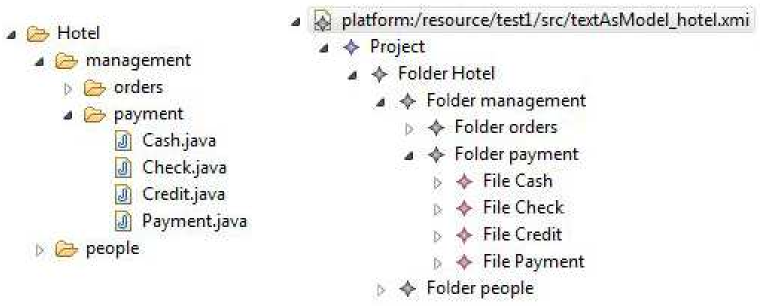
our approach has to cover concepts for describing the structure of a repository that contains the input or output artifacts of a transformation.

Fig. 6.1 shows the metamodel for representing text artifacts stored in repositories using certain folder structures. Meta-class `Repository` represents the entry point to the root folder containing folders and files or to a file if only one single artifact is used. While folders just contain a name, files have in addition an extension as well as a content. The content of files is represented by lines that are sequentially ordered. A derived attribute `content` is used to allow easy access to the complete content of a file.

Figures 6.2 and 6.3 present an instance of the text metamodel coming from a Java code repository. On the left hand side of the figures, the repository's folder structure as well as the content of a Java file are shown, while on the right hand side an excerpt of the corresponding text model (shown in the EMF tree browser) is illustrated.

6.1.1 M2T Example: UML to Java

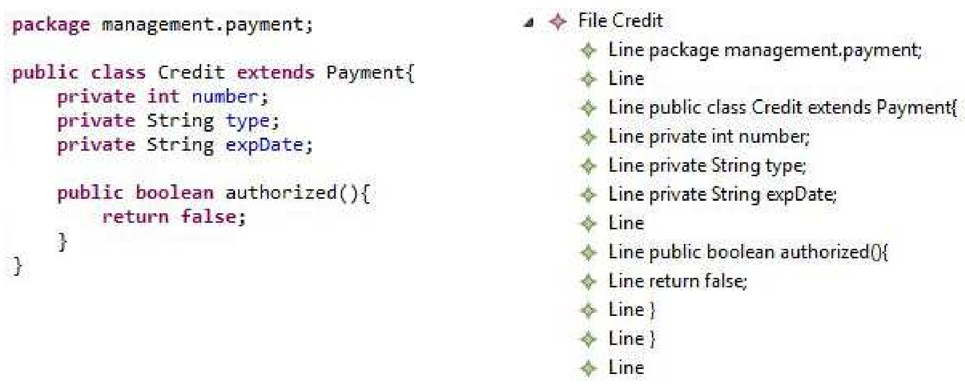
Once we presented the approach and the metamodel to represent text artifacts in the previous section, we introduce some particular examples for demonstrating how to use Tracts for T2M and M2T transformation testing.



```
Hotel
  management
    orders
    payment
      Cash.java
      Check.java
      Credit.java
      Payment.java
  people
```

```
platform:/resource/test1/src/textAsModel_hotel.xmi
  Project
    Folder Hotel
      Folder management
        Folder orders
        Folder payment
          File Cash
          File Check
          File Credit
          File Payment
      Folder people
```

Fig. 6.2 Exemplary folder structure and corresponding text model.



```
package management.payment;

public class Credit extends Payment{
    private int number;
    private String type;
    private String expDate;

    public boolean authorized(){
        return false;
    }
}
```

```
File Credit
  Line package management.payment;
  Line
  Line public class Credit extends Payment{
  Line private int number;
  Line private String type;
  Line private String expDate;
  Line
  Line public boolean authorized(){
  Line return false;
  Line }
  Line }
  Line
```

Fig. 6.3 Exemplary file content and corresponding text model.

For illustration purposes, let us apply our approach to a given case: the transformation that converts UML class models into the corresponding Java classes—which are text files that should be stored in folders inside a code repository. Figure 6.4 shows the subset of the UML metamodel that we will consider in this scenario. It is assumed that all meta-classes directly or indirectly inherit from `NamedElement`. The target metamodel is the one that we described above for specifying text artifacts, and that was shown in Fig. 6.1.

The specification of such a transformation is composed of a set of tracts, each one focusing on a particular property that we want to ensure. As illustrative examples we have chosen 10 tracts, which are described below.

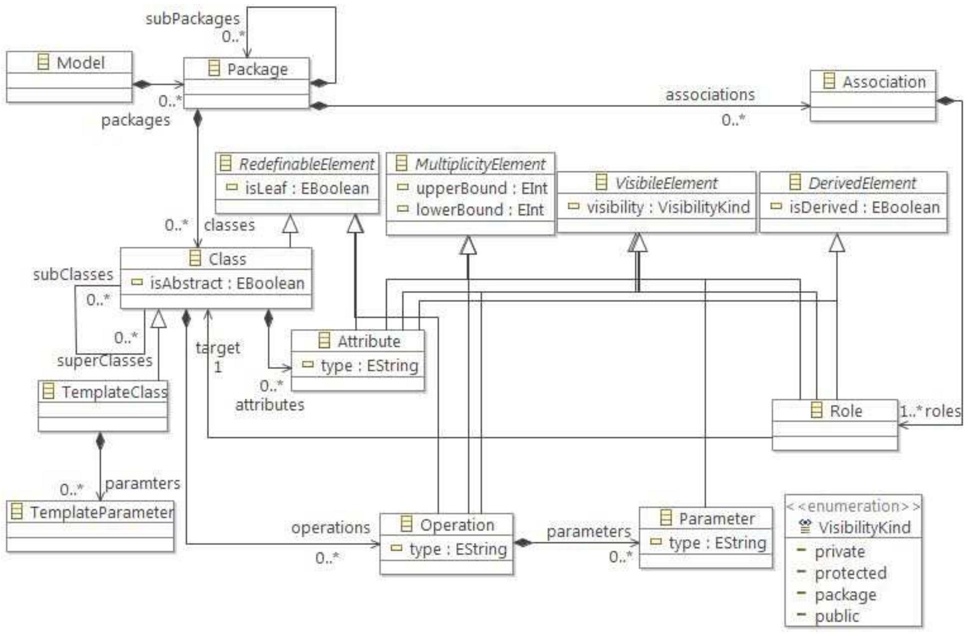


Fig. 6.4 A simplified metamodel for UML class diagrams.

Listing 6.1 Constraint C1

```
-- C1: Nested packages are transformed into nested folders
  Package.allInstances() -> forAll(p| Folder.allInstances()->
    exists(f| f.name = p.name and p.subPackages->
     forAll(subp | f.folders()->exists(subf | subf.name = subp.name))))
```

Notice that in some of them we have used auxiliary operations such as `toFirstUpper` and `toString` to clarify the code. We have also introduced the operation `matchesRE` to deal with regular expressions in OCL. How these auxiliary operations are defined as a user-defined library in OCL is explained in Subsection 6.1.3.

The first tract states in its constraint, which is specified in Listing 6.1, that nested UML packages should be transformed into nested folders. This is specified by the following constraint:

Listing 6.2 Constraint C2

```
-- C2: Package imports when associations are crossing package borders
 Association.allInstances -> select(a |
 a.roles->at(1).target.package <> a.roles->at(2).target.package )
 -> forAll(a| File.allInstances->exists(f |
   f.name = a.roles->at(1).target.name and f.extension = 'java' and
   f.content().matchesRE('import.*'+ a.roles->at(2).target.name))))
```

Listing 6.3 Constraint C3

```
-- C3: No leaf class as superclass
 Class.allInstances() -> forAll(c| c.isLeaf implies
   c.subClasses->isEmpty())
```

The second tract states in Listing 6.2 that Java packages should be imported when associations occur between elements contained in different UML packages.

We should also ask in Listing 6.3 for a precondition in order not to allow that any class inherits from a leaf class.

Another precondition should check that there is no multiple inheritance in use in the UML model (multiple inheritance is not allowed in Java). It is specified in Listing 6.4.

We also include in Listing 6.5 some tracts to specify how particular elements in the UML model should be transformed. For example, derived attributes can not be modified in Java, and therefore only getter methods are generated for them.

Similarly to the tract above, the tract in Listing 6.6 specifies how the visibility of attributes should be handled by the transformation.

And the same for association ends in Listing 6.3.

Finally, three further constraints specify in Listing 6.8 that there are no Java keywords in the UML models, that the names of the elements and folders

Listing 6.4 Constraint C4

```
-- C4: Only one superclass allowed in Java
 Class.allInstances()->forAll(c | c.superClasses->size()<=1)
```

Listing 6.5 Constraint C5

```
-- C5: Derived attributes only have a getter method
Class.allInstances->forAll(c| File.allInstances
  ->exists(f | f.name = c.name and f.extension = 'java' and
    c.attributes->select(a | a.isDerived)->forAll(a |
    not f.content().matchesRE(a.type+'.*?'+a.name+'.*?;') and
    f.content().matchesRE(a.type+'\\s+get'+ toFirstUpper(a.name)))))
```

Listing 6.6 Constraint C6

```
-- C6: Visibility of attributes is considered
Package.allInstances->forAll( p|
  p.classes->forAll(c | File.allInstances->exists(f |
    f.name = c.name and f.extension = 'java' and
    f.container.name = p.name and
    c.attributes->select(a | not a.isDerived)->forAll(a |
        f.content().matchesRE(toString(a.visibility)
            +'.*?'+a.type+'.*?'+a.name+'.*?;')))))
```

Listing 6.7 Constraint C7

```
-- C7: Visibility of roles is considered
 Association.allInstances->forAll(a | File.allInstances->exists(f |
  f.name = a.roles->at(1).target.name and f.extension = 'java' and
   f.content().matchesRE(toString(a.roles->at(2).visibility)+'.*?'+
    a.roles->at(2).target.name+'.*?'+a.roles->at(2).name+'.*?'))))
```

Listing 6.8 Constraints C8, C9 and C10

```
-- C8: No keywords as name in UML model
NamedElement.allInstances()->forAll(ne | not Set{'abstract',
  'extends','implements','class','public','private','protected',...}
  .includes(ne.name))

-- C9: Well-formed names
NamedElement.allInstances()->forAll(ne |
    ne.name.matchesRE('[a-zA-Z_][a-zA-Z0-9_]*'))

-- C10: Generic classes are supported
TemplateClass.allInstances->forAll(c | File.allInstances->exists(f |
    f.name=c.name and f.extension='java' and
    f.content().matchesRE('class\\s+'+c.name+'\\s+<.*?>.*?{'))))
```

are well formed (e.g., no control characters) and that generic UML classes are supported.

Of course, further constraints can be defined to deal with other requirements on the transformation. We have included here the tracts above in order to show examples of the expressiveness of our approach in the case of an M2T transformation. We do not try to claim completeness of full coverage of our specifications for the UML to Java case.

6.1.2 T2M Example: USE to UML

To illustrate the applicability and usage of our proposal in the case of T2M transformations, we have focused on a transformation between textual USE [55] specifications of structural models, and its corresponding UML specifications. USE features for representing models are similar to the ones defined in UML: classes, attributes, associations and operations. For example, the USE code in Listing 6.9 corresponds to a simple model of persons owning cars.

Listing 6.9 Example of USE code

```
class Person
   attributes
       name : String
       birthDate: Integer
   operations
       age() : Integer
end
```

Listing 6.10 Constraint D1

```
-- D1: Only one file per transformation run allowed
  File.allInstances()->size() = 1
```

Listing 6.11 Constraint D2

```
-- D2: Every USE class should result in UML class
  Line.allInstances()->select(l | l.text.matchesRE('^\\s*class'))->
    forAll(l|Class.allInstances->exists(c|l.text.matchesRE(c.name)))
```

```
abstract class Vehicle
   attributes
       brand : String
end
class Car < Vehicle
   attributes
       licenceNumber : String
end
association Person_Car between
   Person [0..1] role owns
   Car [*] role owner
end
```

The set of constraints in Listings 6.10–6.13 are examples to show how different requirements on the transformation from USE to UML can be stated.

The constraint in Listing 6.10 specifies that the USE model should reside in only one file.

The second constraint, in Listing 6.11 states that every USE class will correspond to one UML class with the same name.

The third one, in Listing 6.12 specifies how USE inheritance relationships (cf. '<' symbol in the USE example) are transformed into UML inheritance relationships.

Listing 6.12 Constraint D3

```
-- D3: less-than sign has to open an inheritance relationship
Line.allInstances()->select(l | l.text.matchesRE('\\s*class.*<'))->
 forAll(l|Class.allInstances->exists(c | l.text.matchesRE(c.name) and
 c.superClasses->exists(superClass|l.text.matchesRE(superClass.name))))
```

Listing 6.13 Constraints D4, D5 and D6

```
-- D4: USE abstract classes to UML abstract classes
Line.allInstances()->select(l|l.text.matchesRE('abstract\\s+class'))->
  forAll(l|Class.allInstances->
  exists(c|l.text.matchesRE(c.name) and c.isAbstract))

-- D5: USE attributes to UML attributes
Class.allInstances()->forAll(c|c.attributes->
  forAll(a|File.allInstances->any(f | f.content().
    matchesRE('class\\s*'+c.name+
      '\\s*(<\\s*[A-Za-z0-9]+)?\\s*attributes.*?'+
      a.name+'\\s*:\\s*'+a.type+'.*?(end|operations)'))))

-- D6: USE associations to UML associations
Association.allInstances->forAll(a |
  File.allInstances->any(f | f.content().matchesRE(
    a.roles->iterate(r; s : String =
        '(association|composition)\\s+[A-Za-z0-9_]+\\s+between.*?' |
        s.concat(r.target.name+'.*?role␣'+r.name+'.*?')))))
```

Similarly, the last three constraints are shown in Listing 6.13 and allow us to specify that USE abstract classes are transformed into UML abstract classes, USE attributes into UML attributes, and USE associations into UML associations.

6.1.3 Tool Support

In order to provide tool support for our proposal, we have developed a *injector* (parser) that converts the content of a repository, i.e., files, folders, and their structure, into a model that conforms the Text metamodel shown in Fig. 6.1, and an *extractor* that takes models conforming to the Text metamodel and produces text organized in folders.

In order to check that a given M2T transformation fulfils a set of constraints (such as the ones shown in Section 6.1.1), we run the transformation with the set of models defined by the tract test suite (these input models have not been shown before for the sake of simplicity) and then use the injector with the output text (organized in folders) resulting from the transformation to generate the corresponding output models conforming to the Text metamodel. Then we are in a position to check the validity of the constraints as in the case of tracts defined for M2M transformations with our TractsTool [143]. The TractsTool evaluates the defined constraints on the source and target models by a transparent translation to the USE tool [55].

The case of testing T2M transformations is similar. Here the test suite is defined by the tract as a set of repositories, which need to be transformed first into a model-based representation by our injector component to check the source constraints. When the source constraints are fulfilled, the content of the repository is transformed by the T2M transformation under test to produce the output models. The models produced from the repository and their corresponding output models can then be validated by the TractsTool against the tracts.

For easing the formulation of the OCL constraints, we have also enriched USE with a set of libraries and operations to deal with Strings. For instance, to deal with regular expressions in OCL we have introduced the `matchesRE` operation shown above that checks whether a given sequence matches a regular expression or not. Furthermore, we have also introduced some auxiliary functions that are currently provided by M2T transformation languages such as `toFirstUpper` to end up with more concise OCL constraints than just using the standard OCL String operation library.

The TractsTool for testing M2T/T2M transformations is available at our project website [25] with several examples.

6.2 Evaluation

Most UML tools provide code generation facilities to produce source code from UML models. In order to evaluate the usefulness of using contract-based specifications for code generators, we tested a selected set of currently available UML tools by checking a set of tracts.

6.2.1 Selected Tracts and Test Models

For the evaluation, we used the constraints defined by the tracts presented in Section 6.1.1, which represent some of the most essential requirements that any UML to Java code generator has to fulfil. These constraints are described below, together with their type ("Scr" for source constaints and "ScrTrg" for source-target constaints), as well as one example of the test suite models that was used to check the tracts.

C_1 SrcTrg: Nested packages are transformed into nested folders. Minimal test model: two nested packages in a UML model.

C_2 SrcTrg: Import of packages supported. Minimal test model: two packages, each one having one class and both connected by an association.

C_3 Src: Inheritance of a leaf class is not allowed. Minimal test model: a class inheriting from a leaf class.

C_4 Src: Only single inheritance is used in UML. Minimal test model: one class having two superclasses.

C_5 SrcTrg: Derived attributes only result in getter method. Minimal test model: one class having one derived attribute.

C_6 SrcTrg: Visibility of attributes mapped to Java. Minimal test model: one class having one public, one private, one package, and one protected attribute.

C_7 SrcTrg: Visibility of roles mapped to Java. Minimal test model: two classes related by three associations, whose association ends have different visibilities (public, private, package, and protected).

C_8 Src: No Java keywords are allowed as names in UML models. Minimal test model: one class with name "class", one attribute with name "public", and one operation with name "implements".

C_9 Src: Names in UML model have to be valid Java identifiers. Minimal test model: one class with name "-", attribute with name "+", and operation with name "?".

C_{10} SrcTrg: Generic classes mapped to Java. Minimal test model: one generic class with two parameters.

6.2.2 Selected Tools

We selected six UML tools from industry that claimed to support code generation from UML class diagrams into Java code. The selected sample covers both commercial tools and open-source projects.

- **Altova UModel**[1] is a UML 2.0 tool for software modeling. We evaluated Altova UModel Enterprise Edition 2013.
- **ArgoUML**[2] is a modeling tool supporting UML 1.4 diagrams. It is an open source project and distributed under the Eclipse Public License (EPL). Currently there is only one edition of ArgoUML available. We evaluated version 0.34.
- **BOUML**[3] is a UML 2.0 diagram designer which also allows for code generation. We evaluated version 4.22.2.
- **EnterpriseArchitect**[4] is a commercial modeling tool supporting UML 2.4.1 and is distributed by SparxSystems. We evaluated the professional edition, version 10.
- **MagicDraw**[5] is a commercial modeling tool supporting UML 2.0 and is distributed by NoMagic. We evaluated the enterprise edition, version 16.8.
- **Poseidon for UML**[6] is a modeling tool supporting UML 2.0, distributed by Gentleware. We evaluated the community edition of Poseidon for UML, version 6.0.2.

[1]http://www.altova.com/umodel.html
[2]http://argouml.tigris.org
[3]http://www.bouml.fr
[4]http://www.sparxsystems.com
[5]http://www.nomagic.com
[6]http://www.gentleware.com

6.2.3 Evaluation Procedure

We defined reference test models based on the UML metamodel shown in Fig. 6.4. Subsequently, we re-modelled the reference test models in all of the selected tools. Having the models within the specific tools allowed us to run the validation support and code generators of the specific tools. The validation support is related to the evaluation of support for the *Src* constraints that should act as filter for the code generator, i.e., only valid models should be transformed to code. Thus, we validated all test models in case validation support is available in a specific tool and checked if validation errors or at least warnings are reported for the negative test models associated to the *Src* constraints. For checking the *SrcTrg* constraints, we translated the output of the code generators to Text models and evaluated the resulting output in combination with the input models, i.e., the reference models, using the testing approach described in this paper. The reference models as well as examples of generated Java code and its corresponding text models are available at our project website.

It has to be mentioned that the UML tools are delivered with standard configurations for the code generators. Some tools also allow the users to tweak the code generation capabilities by configuring certain options using specific wizards before running the code generation. Others also allow the edition of code generation scripts, enabling further possibilities to customize the code generation facilities beyond the possibilities offered by the wizards. In this sense, we evaluated first the standard code generation features the tools offer, and after that we tried to tweak the tools by using the wizards to fulfil additional constraints that were not fulfilled in the standard configuration. However, the customization possibilities based on the wizards could not enhance further the evaluation results for the given constraints.

6.2.4 Results

Table 6.1 shows the results of the evaluation. In the table, a tick symbol (✓) means that the test passed for that tract and a cross symbol (×) means that the tract test failed. Some of the tests were not available for a given tool,

Tool/Constraint	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
Altova UModel	×	✓	✓	✓	×	✓	✓	✓	✓	✓
ArgoUML	✓	✓	×	×	-	✓	✓	×	×	✓
BOUML	×	✓	-	✓	×	✓	✓	×	✓	✓
EnterpriseArchitect	✓	✓	✓	×	×	✓	✓	×	×	✓
MagicDraw	✓	✓	✓	✓	×	✓	✓	×	✓	✓
Poseidon	✓	×	×	✓	×	✓	✓	×	✓	-

Table 6.1 Evaluation results

e.g., a particular modeling feature is missing, and were not performed. This is indicated by a dash (-).

In the first place, constraint C_1 did not hold for some tools. In the case of BOUML and Altova UModel, the code generation requires that UML elements are manually associated to certain artifacts for which a path must be specified. Thus, the user has to specify the folders and Java files that should be generated. All other tools work well with packages in an automated way.

Concerning associations that cross package borders (C_2), Poseidon is the only tool that does not take this feature into account in the code generation process.

Precondition C_3 checks that no class inherits by another class marked as leaf. BOUML does not include the option to set a class as leaf. Poseidon fails because it allows that a class inherits from a leaf class. ArgoUML passes the test and gives a warning during the model validation only when the superclass is marked as leaf before the creation of the generalization relationship.

C_4 checks that the UML model does not use multiple inheritance, because it cannot be used in Java. ArgoUML and MagicDraw fail because they do not check this constraint, and they both create a Java class which does not even compile.

Concerning C_5, ArgoUML does not allow the definition of derived features. The rest of the tools do, but derived features are ignored in the code generation process. An expected solution would create derived attributes into their corresponding getter methods.

All tools work well with the transformation of the visibility of attributes and roles (constraints C_6 and C_7).

Most tools fail with constraints C_8 and C_9 (use of Java keywords and invalid names in Java). Tools do not seem to conduct any validation check before the code generation starts. Although many tools allow several kinds of validation checks on the UML models, most of these tests only deal with UML constraints. A few tools also allow the development of user-defined validation checks, but they do not seem to have been defined for the code generation facilities they support. The best results in this respect are achieved by Altova UModel, which raises a warning if non-valid Java identifiers or Java keywords are used as names for UML elements.

Finally, generic classes are supported and correct Java code is generated by all UML tools (constraint C_{10}) except Poseidon, which does not allow the definition of generic classes.

In summary, the results show that code generators have to fulfil several properties that should be specified at a higher level for allowing their validation. In particular, we found that no tool performs well even with respect to the basic UML to Java code generation requirements. Furthermore, we discovered that several tools produced incorrect Java code, even not compilable in some situations. In this sense, the tracts representing the basic requirements could be used as the initial components of a benchmark for future improvements and developments of UML-to-Java code generators.

6.3 Related Work

Several kinds of works apply contracts for M2M transformation testing using different notations for defining the contracts. In the following, we divide them into two main categories. First, contracts may be defined on the *model level* by either giving *(i)* complete examples of source and target model pairs, or *(ii)* giving only model fragments which should be included in the produced target models for given source models. Second, contracts may be defined on the *metamodel level* either by using *(iii)* graph constraint languages or *(iv)* textual constraint languages such as OCL.

A straight-forward approach is to define the expected target model for a given source model which acts as a reference model for analyzing the actual produced target model of a transformation as proposed in [50, 81, 93, 94]. Model comparison frameworks are employed for computing a difference model between the expected and the actual target models. If there are differences then it is assumed that there exists an error either in the transformation or in the source/target model pair. Analogously, one could employ text comparison frameworks to reason about an expected text artifact and an computed text artifact. However, reasoning about the cause for the mismatch between the expected and actual text artifact solely based on the difference model is challenging. Several elements in the difference model may be effected by the same error, however, the transformation engineer has the burden to cluster the differences by herself.

A special form of verification by contract was presented in [102]. The authors propose to use model fragments (introduced in [113]) which are expected to be included in a target model which is produced from a specific source model. Using fragments as contracts is different from using examples as contracts. Examples require an equivalence relationship between the expected model and actual target model, while fragments require an inclusion relationship between the expected fragments and the actual target model. Using our text metamodel, one is able to define such fragments even for M2T/T2M transformations, but they still only define the oracle for one particular input model.

Guerra et al. [65] proposed a declarative language for the specification of visual contracts for defining pre- and post-conditions as well as invariants for model transformations. For evaluating the contracts on test models, the specifications are translated to QVT Relations which are executed in check-only mode. In particular, QVT Relations are executed before the transformation under test is executed to check the preconditions on the source models and afterwards to check relationships between the source and target models as well as postconditions on the target models. This approach may be used as an alternative syntax for our presented approach. Further alternative text-based approaches for defining oracles are presented in [32, 33, 50, 53, 80], however, they do not discuss how to apply their approaches for text artifacts.

The most closely related work is presented in Tiso et al. [139] where the problem of testing model-to-code transformations is explicitly mentioned. The authors enumerate two possibilities for such tests. First, they briefly mention a static approach which evaluates if certain properties are fulfilled by the transformation target code. However, they do not describe the details of this possibility. Second, they discuss a dynamic approach based on checking the execution of the transformation target, which is subsequently elaborated in their paper. In particular, they model, in addition to the domain classes, test classes that execute certain operations and check for given post-conditions after the operations have been executed. While we propose a generic and static approach to test M2T/T2M transformations in general, Tiso et al. propose an approach for testing a specific model-to-code transformation, namely from UML class diagrams to specific Java code and using JUnit tests that are also derived from a model representation. Furthermore, in our approach we have the possibility to directly test M2T/T2M transformations. However, in Tiso et al. [139] the execution output of the generated application has to be analyzed to trace eventual errors back to the M2T transformation.

Finally, an approach for testing code generators for executable languages is presented in [133]. The authors present a two-folded approach. On the one hand, first-order test cases that represents the models which are transformed into code are distinguished. On the other hand, second-order test cases are introduced representing tests that are executed on models as well as on the derived implementation, i.e., on the generated code. The output of the code execution is compared with the output of the model execution. If these outputs are equivalent, it is assumed that the code generator works as expected. Compared to our proposal, we provide an orthogonal approach for testing the syntactic equivalence by checking certain constraints, i.e., how to define oracles for the first-order test cases. Combining a syntactical with a semantical approach seems to be an interesting subject for future work.

6.4 Summary

This chapter presented a language-agnostic approach for testing M2T/T2M transformations. Agnostic means independent from the languages used for the source and target artifacts of the transformations, as well as to the transformation language used for implementing the transformations. By extending OCL with additional String operations, we have been able to specify contracts for practical examples and evaluated the correctness of current UML-to-Java code generators offered by well-known UML tools. This evaluation showed a great potential for further improving code generators and documents the real need for an engineering discipline to develop M2T/T2M transformations.

Chapter 7
Conclusions and Future Work

This chapter summarizes the proposal that has been explained throughout this dissertation, highlighting the conclusions of our work in Section 7.1 and the main publications obtained from these contributions and some others in Section 7.2. Finally, future work is described in Section 7.3.

7.1 Summary and Conclusions

In the first stages of this thesis, we studied the state of the art and observed that model transformations lacked appropriate concurrency mechanisms and tool support to be executed in parallel—like most software artifacts developed nowadays. This is why the first contribution of this thesis is the approach presented in Chapters 3 and 4 for the parallel execution of model transformations using the concepts and mechanisms of Linda.

We implemented a layer on top of existing Java-based data grids that specify model transformations in Java, transparent to the underlying technologies and parallelization strategies. In particular, we have shown how to represent metamodels and models for running transformations in parallel as well as how to provide intrinsic traceability for out-place transformations. Based on these results, the jLinTra platform seems quite promising as a base infrastructure for running efficient model transformations when combined with a data parallelism approach to achieve parallelization. The comparison against other well-known sequential transformation engines has also shown a significant speedup and the comparison against a parallel engine using task parallelism has demonstrated that data parallelism, as used by jLinTra, seems to work better than process parallelism for the model transformations used in the case study. Furthermore, we have shown with experiments that the performance of its in-place mode improves w.r.t. other in-place MT engines and that in those cases where in-place transformations can be achieved also by means of out-place transformations, the in-place transformations provide better performance and usability.

As in-memory data grids have a significant influence on our approach, the evaluation also shows that current data grids perform well, i.e., they do not necessarily introduce significant penalties. However, there is a relevant difference in the performance when accessing data.

All in all, the jLinTra solution presented here seems to be able to handle and scale larger models, and significantly outperforms current model transformation engines, both sequential and parallel.

The second main contribution of this thesis addresses the need that arises after the execution of a model transformation, namely to prove that execution was correct and the expected output was obtained. In Chapter 5 we presented a static approach to trace errors in model transformations. Taking a model transformation and a set of constraints that specify its expected behavior as input elements, our approach automatically extracts the footprints of both artifacts and compares transformation rules and constraints one by one, obtaining the overlap of common footprints. Subsequently, it returns three matching tables where the alignments between rules and constraints are recorded. By using these tables, the transformation engineer is able to trace the rules that can be the cause of broken constraints due to faulty behavior. Our evaluation shows that the presented approach is expected to be accurate for a large set of model transformations. By using the similarity matrixes, an automated and instant fitness test is available to check a-priori whether the approach is likely to be helpful for a given transformation.

The last contribution of this thesis is presented in Chapter 6. We broadened the scope of our testing approach for M2T and T2M transformations. We created a generic metamodel so that text repositories can be represented as models conforming to this metamodel. Once text has been represented as a model, the problem of testing M2T/T2M transformations is reduced to a M2M transformation testing problem. As in the text part of the transformation there are String attributes whose values need to be analyzed in many cases, so we extended OCL with additional String operations.

The evaluation that we carried out by means of evaluating the correctness of current UML-to-Java code generators showed great potential for further improving code generators. It further documents the very real need for an engineering discipline to develop M2T and T2M transformations and the importance of testing these kinds of transformations.

7.2 Publications

This section shows how the work that supports this dissertation has been published in journals, conferences and workshops with peer-review. It also enumerates the rest of the research contributions that the author of this dissertation has made over the course of her Ph.D. although they are not related to the lines of research presented here.

7.2.1 Publications Supporting this Dissertation

International Journals

- Loli Burgueño, Javier Troya, Manuel Wimmer, Antonio Vallecillo. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering* 41(5):490-506, 2015. This paper supports the content presented in Chapter 5.

International Journals under Review

- Loli Burgueño, Manuel Wimmer, Antonio Vallecillo. A Linda-based Platform for the Parallel Execution of Out-place Model Transformations. *Information and Software Technology.* Submitted. This paper presents the approach shown in Chapter 3.

International Conferences

- Loli Burgueño, Manuel Wimmer. Testing M2T/T2M Transformations. In *Proc. of the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems 2013 (MoDELS 2013).* Miami, FL, USA, September-October 2013. This paper presents the content of Chapter 6.

International Workshops

- Loli Burgueño, Javier Troya, Manuel Wimmer, Antonio Vallecillo. Parallel In-Place Model Transformations with LinTra. In *Proc. of the 3rd International Workshop on Big MDE (BigMDE 2015)* at STAF 2015. L'Aquila, Italy, July 2014. This paper presents the content of Chapter 4.

- Loli Burgueño, Javier Troya, Manuel Wimmer, Antonio Vallecillo. On the Concurrent Execution of Model Transformations with Linda. In *Proc. of the 1st International Workshop on Big MDE (BigMDE 2013)* at STAF 2013. Budapest, Hungary, June 2013. Part of the content of this paper is presented in Chapter 3.

- Loli Burgueño, Manuel Wimmer, Antonio Vallecillo. Towards Tracking Guilty Transformation Rules. In *Proc. of the 1st Workshop on the Analysis of Model Transformation 2012 (AMT 2012)* at MODELS 2012. Innsbruck, Austria, October 2012. This paper comprises a preliminary version of the work presented in Chapter 5.

National (Spanish) Conferences

- Loli Burgueño, Javier Troya, Antonio Vallecillo. Concurrent Model Transformations with Linda. In *Proc. of the XVIII Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2013)* at CEDI 2013. Madrid, Spain, September 2013. Part of the work presented in this paper is contained in Chapter 3.

- Manuel Wimmer, Loli Burgueño, Antonio Vallecillo. Prueba de Transformaciones de Modelos con TractsTool. In *Proc. of the XVII Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2012)*. Almería, Spain, September 2012. This paper presents the first version of the tool that supports the work presented in Chapters 5 and 6.

Others

- **ACM Student Research Competition:** Loli Burgueño. Testing M2M/M2T/T2M Transformations. In *Proc. of the ACM Student Research Competition (ACM SRC 2015)* at MODELS 2015. Ottawa, Canada, September 2015. (**Winner of the Bronze medal**). This contribution summarizes the content of Chapters 5 and 6.

- **Book chapter:** Antonio Vallecillo, Martin Gogolla, Loli Burgueño, Manuel Wimmer, Lars Hamann. Formal Specification and Testing of

Model Transformation. *Formal Methods for Model-Driven Engineering*, Springer LNCS 7320, pp. 399-437, June 2012. Part of the content of Section 2.3 is introduced in this book chapter.

7.2.2 Further Publications

International Conferences

- Martin Gogolla, Antonio Vallecillo, Loli Burgueño, Frank Hilken. Employing Classifying Terms for Testing Model Transformations. In *Proc. of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems 2015 (MoDELS 2015).* Ottawa, Canada, September-October 2015. (**Best Paper Award**).

- David Ameller, Xavier Frank, Cristina Gómez, Antonio Vallecillo, Loli Burgueño, et al. Handling Non-functional Requirements in Model-Driven Development: An Ongoing Industrial Survey. In *Proc. of the 23rd IEEE International Requirements Engineering Conference 2015 (RE 2015).* Ottawa, Canada, August 2015.

International Workshops

- Frank Hilken, Loli Burgueño, Martin Gogolla, Antonio Vallecillo. Iterative Development of Transformation Models by Using Classifying Terms. In *Proc. of the 4th Workshop on the Analysis of Model Transformation 2015 (AMT 2015)* at MODELS 2015. Ottawa, Canada, September 28, 2015.

- Javier Troya, Alexander Bergmayr, Loli Burgueño, Manuel Wimmer. Towards Systematic Mutations for and with ATL Model Transformations. In *Proc. of the 10th International Workshop on Mutation Analysis (Mutation 2015)* at ICST 2015. Graz, Austria, April 13, 2015.

- Javier Troya, Manuel Wimmer, Loli Burgueño, Antonio Vallecillo. Towards Approximate Model Transformations. In *Proc. of the 3rd Work-*

shop on the Analysis of Model Transformation 2014 (AMT 2014) at MODELS 2014. Valencia, Spain, September 29, 2014.

- Loli Burgueño, Eugene Syriani, Manuel Wimmer, Jeff Gray, Antonio Vallecillo. LinTraP: Primitive Operators for the Execution of Model Transformations with LinTra. In *Proc. of the 2nd International Workshop on Big MDE (BigMDE 2014)* at STAF 2014. York, United Kingdom, July 2014.

National (Spanish) Conferences

- Loli Burgueño, Antonio Moreno-Delgado, Antonio Vallecillo. Analysis of the Scientific Production of the Spanish Software Engineering Community. In *Proc. of the XX Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2015)* at SISTEDES 2015. Santander, Spain, September 2015.

- Loli Burgueño, Eugene Syriani, Manuel Wimmer, Jeff Gray, Antonio Vallecillo. Primitive Operators for the Concurrent Execution of Model Transformations Based on LinTra. In *Proc. of the XIX Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2014)* at SISTEDES 2014. Cádiz, Spain, September 2014.

7.3 Future Work

We consider that the work presented in this dissertation is mature and solves a specific problem. Nevertheless, all research can be improved in several directions. The different lines of work that we would like to explore next are explained in the following paragraphs.

We have already mentioned that one of the weaknesses of LinTra appears when a transformation needs to access the data layer many times in order to resolve a long navigation path and compute its final value. In order to solve this problem, when an element has a reference to another element, we plan to keep not only the identifier of the referenced element but the element itself.

We already know this will have a negative impact on the amount of memory needed to store the model. We plan to quantitatively analyze the penalty and the benefits of this solution and try to find the balance between time and space. We will also identify those situations where we can benefit from one of them and apply the most adequate alternative in each case. We also plan to investigate the integration of advanced query engines such as EMF Inc-Query.

Although model transformations can be written in jLinTra, a more ambitious and long-term goal is to explore how high-level transformation languages (either existing ones or newly developed, supporting parallel constructs) can be compiled to jLinTra so that model transformations written in those languages can be provided for our engine.

The experiments conducted so far with respect to the distribution of models and their transformation over several machines are positive and we can naturally deal with models spread over networked computers using jLinTra. However, the impact of the network latency on the performance is not negligible, and may require the introduction of optimization mechanisms when very large models need to be transformed using networked machines.

We also plan to apply the LinTra mechanisms presented to execute linear model transformation chains to execute generic model transformation networks.

Regarding the execution of in-place model transformations, we plan to provide a new in-place execution mode that supports recursive matchings such as those present in graph transformations.

There are always trade-offs between using embedded transformation frameworks like LinTra and dedicated model transformation languages such as ATL. For instance, model transformation languages own specific forms of providing modularity and reusability. On the other hand, frameworks are supposed to perform better as they do not spend time compiling or interpreting code. We leave for future work a complete comparative study between LinTra and different model transformation languages considering characteristics such as performance, analysability, coupling with particular modelling frameworks, modularity, reusability and testability.

As the last line of future work on the LinTra side, we plan to explore the benefits of hybrid parallelism combining the advantages of task and data parallelism, and to conduct further experiments and benchmarking exercises.

As stated in Chapter 5, our static approach to test the correctness of model transformations requires in its first step the extraction of the metamodel footprints. So far, the footprint extraction for transformation contracts in OCL is currently supported, but other contract languages such as PaMoMo [65] may be employed as well. Similarly, the application of the ideas presented here to other transformation languages which do not use OCL, like graph-based languages (e.g., AGG) or other kinds of languages (e.g., Tefkat), opens up further lines of research.

We have also introduced similarity matrixes as an a-priori applicability test. We aim to explore the use of similarity matrixes for other purposes, such as reasoning about the maintainability of transformations in the case of evolving metamodels or about the completeness of transformations.

As we offer support to test M2T and T2M transformations, we plan to investigate how current Architecture Driven Modernization (ADM)[1] modeling standards such as the Knowledge Discovery Metamodel (KDM) [110] may be used to define contracts that are programming language independent and reusable for a family of code generators. For example, the presented contracts may be expressed in a platform-independent way and reused for testing UML-to-C# code generators.

As our testing approach is static, it would be interesting to explore how dynamic approaches could complement our static approach and trace guilty transformation rules.

There is also room for improvement in our tool TractsTool. It is a prototype whose limits need to be explored and improved. The models defined in the Tracts' test suites are normally of reasonable size (less than one or two thousand elements) because this is usually enough to check the Tract constraints. However, we have discovered that large models (with several thousands of model elements) are hard to manage with the tools that we are

[1]http://adm.omg.org

currently using. For this reason we plan to look for internal optimizations of the tool.

References

[1] Amrani, M., Lúcio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y. L., and Cordy, J. R. (2012). A tridimensional approach for studying the formal verification of model transformations. In *Proc. of the VOLT'12 Workshop*.

[2] Anastasakis, K., Bordbar, B., and Küster, J. M. (2007). Analysis of model transformations via Alloy. In *Proc. of MODEVVA'07*.

[3] Aranega, V., Mottu, J.-M., Etien, A., and Dekeyser, J.-L. (2009). Traceability mechanism for error localization in model transformation. In *Proc. of ICSOFT'09*, pages 66–73. INSTICC Press.

[4] Associates, S. C. (2000). Linda. user's guide and reference manual. https://www.lcrc.anl.gov/jazz/Documentation/ApplicationDocumentation/g98/lindamanual.pdf.

[5] Atkinson, C. and Kühne, T. (2001). The essence of multilevel metamodeling. In *Proc. of UML'01*, pages 19–33, London, UK, UK. Springer-Verlag.

[6] Baresi, L., Ehrig, K., and Heckel, R. (2007). Verification of model transformations: A case study with BPEL. In *Proc. of TGC'06*, volume 4661 of *LNCS*, pages 183–199. Springer.

[7] Baudry, B., Dinh-Trong, T., Mottu, J.-M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., and Traon, Y. L. (2006). Model transformation testing challenges. In *Proc. of IMDD-MDT'06*.

[8] Baudry, B., Ghosh, S., Fleurey, F., France, R., Traon, Y. L., and Mottu, J.-M. (2010). Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143.

[9] Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., and Launay, D. (2014). Neo4EMF, a scalable persistence layer for EMF models. In *Proc. of ECMFA'14*, volume 8569 of *LNCS*, pages 230–241. Springer.

[10] Benelallam, A., Gómez, A., Tisi, M., and Cabot, J. (2015). Distributed model-to-model transformation with ATL on MapReduce. In *Proc. of SLE 2015*, pages 37–48. ACM.

[11] Bergmann, G., Ráth, I., and Varró, D. (2009). Parallelization of graph transformation based on incremental pattern matching. *ECEASST*, 18:1–15.

[12] Bergmayr, A., Brunelière, H., Canovas Izquierdo, J., Gorronogoitia, J., Kousiouris, G., Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria, L., Pezuela, C., and Wimmer, M. (2013). Migrating Legacy Software to the Cloud with ARTIST. In *Proc. of CSMR*, pages 465–468.

[13] Bergmayr, A., Troya, J., and Wimmer, M. (2014). From out-place transformation evolution to in-place model patching. In *Proc. of ASE'14*, pages 647–652. ACM.

[14] Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, 4(2):171–188.

[15] Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., and Lindow, A. (2006). Model transformations? transformation models! In *Proc. of MODELS'06*, pages 440–453.

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

[17] Brooks, F. P. (1986). No Silver Bullet – Essence and Accident in Software Engineering. In *Proc. of the IFIP Tenth World Computing Conference*, pages 1069–1076.

[18] Brottier, E., Fleurey, F., Steel, J., Baudry, B., and Traon, Y. L. (2006). Metamodel-based test generation for model transformations: an algorithm and a tool. In *Proc. of ISSRE'06*, pages 85–94. IEEE.

[19] Bruneliére, H., Cabot, J., Dup´e, G., and Madiot, F. (2014). MoDisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032.

[20] Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proc. of ASE'10*, pages 173–174. ACM.

[21] Burgueño, L. (2013). Concurrent and distributed model transformations based on linda. In *Proc. of the MODELS 2013 Doctoral Symposium*, volume 1071 of *CEUR Workshop Proceedings*, pages 9–16. CEUR-WS.org.

[22] Burgueño, L., Troya, J., Wimmer, M., and Vallecillo, A. (2015). Parallel in-place model transformations with lintra. In *Proc. of BigMDE'15 @ STAF'15*, volume 1406 of *CEUR Workshop Proceedings*, pages 52–62. CEUR-WS.org.

[23] Burgueño, L., Wimmer, M., Troya, J., and Vallecillo, A. (2014a). Fault localization in model transformations. http://atenea.lcc.uma.es/index.php/Main_Page/Resources/FaultLocMT.

[24] Burgueño, L., Wimmer, M., Troya, J., and Vallecillo, A. (2014b). Similarity matrixes. http://atenea.lcc.uma.es/index.php/Main_Page/Resources/MTB/SimilarityMatrix.

[25] Burgueño, L., Wimmer, M., Troya, J., and Vallecillo, A. (2014c). TractsTool. http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Tracts.

[26] Burgueño, L., Wimmer, M., Troya, J., and Vallecillo, A. (2015). Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506.

[27] Burgueño, L., Wimmer, M., and Vallecillo, A. (2014d). Lintra. http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra.

[28] Burgy, L., Consel, C., Latry, F., Lawall, J., Palix, N., and Reveillere, L. (2006). Language technology for internet-telephony service creation. In *Proc. of ICC'06*, pages 1795–1800. IEEE.

[29] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, Chichester, UK.

[30] Cabot, J., Clarisó, R., Guerra, E., and de Lara, J. (2010). Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302.

[31] Cabot, J. and Teniente, E. (2006). Transforming OCL constraints: a context change approach. In *Proc. of SAC'06*, pages 1196–1201. ACM.

[32] Cariou, E., Belloir, N., Barbier, F., and Djemam, N. (2009). OCL contracts for the verification of model transformations. *ECEASST*, 24.

[33] Cariou, E., Marvie, R., Seinturier, L., and Duchien, L. (2004). OCL for the specification of model transformation contracts. In *Proc. of the OCL and Model Driven Engineering Workshop*.

[34] Ciancone, A., Filieri, A., and Mirandola, R. (2010). MANTra: Towards model transformation testing. In *Proc. of QUATIC'10*, pages 97–105. IEEE.

[35] Clasen, C., Didonet Del Fabro, M., and Tisi, M. (2012). Transforming Very Large Models in the Cloud: a Research Roadmap. In *Proc. of CloudMDE 2012*, pages 1–10.

[36] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2007). *All About Maude – A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer.

[37] Cuadrado, J. S. and de Lara, J. (2013). Streaming model transformations: Scenarios, challenges and initial solutions. In *Proc. of ICMT 2013*, volume 7909 of *LNCS*, pages 1–16. Springer.

[38] Czarnecki, K. and Helsen, S. (2006a). Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645.

[39] Czarnecki, K. and Helsen, S. (2006b). Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645.

[40] Daniel, G., Sunyé, G., Benelallam, A., and Tisi, M. (2014). Improving memory efficiency for processing large-scale models. In *Proc. of BigMDE'14 @ STAF'14*, volume 1206 of *CEUR Workshop Proceedings*, pages 31–39. CEUR-WS.org.

[41] Deak, L., Mezei, G., Vajk, T., and Fekete, K. (2013). Graph partitioning algorithm for model transformation frameworks. In *Proc. of EUROCON 2013*, pages 475–481. IEEE.

[42] Dijkstra, E. W. (1974). On the role of scientific thought. published as EWD:EWD447pub.

[43] Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., and Varró-Gyapay, S. (2005). Termination criteria for model transformation. In *Proc. of FASE'05*, volume 3442 of *LNCS*, pages 49–63. Springer.

[44] Ehrig, K., Küster, J. M., and Taentzer, G. (2009). Generating instance models from meta models. *Software and Systems Modeling*, 8(4):479–500.

[45] Espinazo-Pagán, J., Cuadrado, J. S., and Molina, J. G. (2015). A repository for scalable model management. *Software and Systems Modeling*, 14(1):219–239.

[46] Espinazo Pagán, J. and García Molina, J. (2014). Querying large models efficiently. *Information and Software Technology*, 56(6):586–622.

[47] France, R. B. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *Proc. of ISCE'07*, pages 37–54.

[48] Fritzsche, M., Gilani, W., Lämmel, R., and Jouault, F. (2010). Model transformation chains in model-driven performance engineering: Experiences and future research needs. In *Proc. of Modellierung 2010*, pages 213–220.

[49] Galvão, I. and Goknil, A. (2007). Survey of traceability approaches in model-driven engineering. In *Proc. of EDOC'07*, pages 313–326. IEEE.

[50] García-Domínguez, A., Kolovos, D. S., Rose, L. M., Paige, R. F., and Medina-Bulo, I. (2011). EUnit: A unit testing framework for model management tasks. In *Proc. of MODELS'11*, volume 6981 of *LNCS*, pages 395–409. Springer.

[51] Gelernter, D. (1985). Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112.

[52] Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Commun. ACM*, 35(2):96–107.

[53] Giner, P. and Pelechano, V. (2009). Test-driven development of model transformations. In *Proc. of MODELS'09*, volume 5795 of *LNCS*, pages 748–752. Springer.

[54] Gogolla, M., Bohling, J., and Richters, M. (2005). Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Software and Systems Modeling*, 4(4):386–398.

[55] Gogolla, M., Büttner, F., and Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69:27–34.

[56] Gogolla, M. and Vallecillo, A. (2011). *Tract*able model transformation testing. In *Proc. of ECMFA'11*, volume 6698 of *LNCS*, pages 221–236. Springer.

[57] Gogolla, M., Vallecillo, A., Burgueño, L., and Hilken, F. (2015). Employing classifying terms for testing model transformations. In *Proc. of MODELS'15*, pages 312–321.

[58] Goldschmidt, T. and Wachsmuth, G. (2008). Refinement Transformation Support for QVT Relational Transformations. In *Proc. of the MDSE'08 Workshop*.

[59] Gómez, A., Tisi, M., Sunyé, G., and Cabot, J. (2015). Map-based transparent persistence for very large models. In *Proc. of FASE 2015*, volume 9033 of *LNCS*, pages 19–34. Springer.

[60] González, C. A. and Cabot, J. (2012). ATLTest: A white-box test generation approach for atl transformations. In *Proc. of MODELS'12*, volume 7590 of *LNCS*, pages 449–464. Springer.

[61] Group, A. (2016). Atl zoo. http://www.eclipse.org/atl/atlTransformations/.

[62] Guerra, E. (2012a). Specification-driven test generation for model transformations. In *Proc. of ICMT'12*, volume 7307 of *LNCS*, pages 40–55. Springer.

[63] Guerra, E. (2012b). Specification-driven test generation for model transformations. In *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT 2012)*, volume 7307 of *LNCS*, pages 40–55. Springer.

[64] Guerra, E., de Lara, J., Kolovos, D. S., Paige, R. F., and Santos, O. (2012). Engineering model transformations with transML. *Software and Systems Modeling*, 12(3):555–577.

[65] Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., and Schwinger, W. (2013). Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46.

[66] Heckel, R., Küster, J. M., and Taentzer, G. (2002). Confluence of typed attributed graph transformation systems. In *Proc. of ICGT 2002*, volume 2505 of *LNCS*, pages 161–176. Springer.

[67] Hibberd, M., Lawley, M., and Raymond, K. (2007). Forensic debugging of model transformations. In *Proc. of MODELS'07*, volume 4735 of *LNCS*, pages 589–604. Springer.

[68] Hidaka, S., Tisi, M., Cabot, J., and Hu, Z. (2015). Feature-based classification of bidirectional transformation approaches. *Software and Systems Modeling*, pages 1–22.

[69] Hilken, F., Burgueño, L., Gogolla, M., and Vallecillo, A. (2015). Iterative development of transformation models by using classifying terms. In *Proc. of the AMT'15 Workshop @ MODELS'15*, pages 1–6.

[70] Horn, T., Krause, C., and Tichy, M. (2014). The TTC 2014 Movie Database Case. In *Proc. of the 7th Transformation Tool Contest (TTC 2014)*, volume 1035, pages 93–97. CEUR Workshop Proceedings.

[71] IEEE (1990). *Standard glossary of software engineering terminology.* IEEE Std. 610.12.

[72] Imre, G. and Mezei, G. (2012). Parallel graph transformations on multicore systems. In *Proc. of MSEPT 2012*, volume 7303 of *LNCS*, pages 86–89. Springer.

[73] Jeanneret, C., Glinz, M., and Baudry, B. (2011). Estimating footprints of model operations. In *Proc. of ICSE'11*, pages 601–610. ACM.

[74] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

[75] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39.

[76] Jouault, F., Bézivin, J., Consel, C., Kurtev, I., and Latry, F. (2006). Building DSLs with AMMA/ATL, a case study on SPL and CPL telephony languages. In *Proc. of ECOOP Workshop on Domain-Specific Program Development.*

[77] Jouault, F. and Tisi, M. (2010). Towards incremental execution of ATL transformations. In *Proc. of ICMT 2010*, volume 6142 of *LNCS*, pages 123–137. Springer.

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

[79] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA explained - the Model Driven Architecture: practice and promise.* Addison Wesley object technology series. Addison-Wesley.

[80] Kolovos, D., Paige, R., Rose, L., and Polack, F. (2008a). Unit testing model management operations. In *Proc. of ICSTW'08*, pages 97–104. IEEE.

[81] Kolovos, D. S., Paige, R. F., and Polack, F. A. (2006). Model comparison: a foundation for model composition and model transformation testing. In *GaMMa'06*, pages 13–20. ACM.

[82] Kolovos, D. S., Paige, R. F., and Polack, F. A. (2008b). The Epsilon Transformation Language. In *Proc. of ICMT'08*, volume 5063 of *LNCS*, pages 46–60. Springer.

[83] Kolovos, D. S., Rose, L. M., Matragkas, N., Paige, R. F., Guerra, E., Sánchez Cuadrado, J., De Lara, J., Ráth, I., Varró, D., Tisi, M., and Cabot, J. (2013). A research roadmap towards achieving scalability in model driven engineering. In *Proc. of BigMDE'13 @ STAF'13*, pages 2:1–2:10. ACM.

[84] Krause, C., Tichy, M., and Giese, H. (2014). Implementing graph transformations in the bulk synchronous parallel model. In *Proc. of FASE 2014*, volume 8411 of *LNCS*, pages 325–339. Springer.

[85] Kurtev, I. (2008). Application of Reflection in Model Transformation Languages. In *Proc. of ICMT'08*, volume 5063 of *LNCS*, pages 199–213. Springer.

[86] Kurtev, I., Bézivin, J., and Akşit, M. (2002). Technological spaces: An initial appraisal. In *Proceedings of the Confederated International Conferences (CoopIS, DOA, and ODBASE), Industrial track*.

[87] Küster, J. M. (2006). Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259.

[88] Küster, J. M. and Abd-El-Razik, M. (2006). Validation of model transformations: First experiences using a white box approach. In *Proc. of MODELS'06 Workshops*, volume 4364 of *LNCS*, pages 193–204. Springer.

[89] Lara, J. D., Guerra, E., and Cuadrado, J. S. (2014). When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2):12:1–12:46.

[90] Lee, A. S. (1989). A scientific methodology for MIS case studies. *MIS Quarterly*, 13(1):33–50.

[91] Lehman, T. J., McLaughry, S. W., and Wycko, P. (1999). T Spaces: The next wave. In *Proc. of HICSS'99*, pages 8037–8045. IEEE Computer Society.

[92] Lennox, J., Wu, X., and Schulzrinne, H. (2004). Call processing language (CPL): A language for user control of internet telephony services. http://www.ietf.org/rfc/rfc3880.txt.

[93] Lin, Y., Zhang, J., and Gray, J. (2004). Model comparison: A key challenge for transformation testing and version control in model driven software development. In *Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development*, pages 219–236. Springer.

[94] Lin, Y., Zhang, J., and Gray, J. (2005). A testing framework for model transformations. In *Model-Driven Software Development – Research and Practice in Software Engineering*, pages 219–236. Springer.

[95] Ludewig, J. (2004). Models in software engineering - An introduction. *Inform., Forsch. Entwickl.*, 18(3-4):105–112.

[96] Mamoud, Q. H. (2005). Getting started with javaspaces technology: Beyond conventional distributed programming paradigms. http://www.oracle.com/technetwork/articles/javase/javaspaces-140665.html.

[97] Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.

[98] Mens, T. and Gorp, P. V. (2006). A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142.

[99] Mens, T., Taentzer, G., and Runge, O. (2005). Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electr. Notes Theor. Comput. Sci.*, 127(3):113–128.

[100] Meyer, B. (1992). Applying design by contract. *IEEE Computer*, 25(10):40–51.

[101] Mottu, J.-M., Baudry, B., and Traon, Y. L. (2006). Mutation analysis testing for model transformations. In *Proc. of ECMDA-FA*, volume 4066 of *LNCS*, pages 376–390. Springer.

[102] Mottu, J.-M., Baudry, B., and Traon, Y. L. (2008). Model transformation testing: oracle issue. In *ICSTW'08*, pages 105–112. IEEE.

[103] Mottu, J.-M., Sen, S., Tisi, M., and Cabot, J. (2012). Static analysis of model transformations for effective test generation. In *Proc. of ISSRE'12*, pages 291–300. IEEE.

[104] Object Management Group (OMG) (2001). *OMG. Model Driven Architecture - A Technical Perspective.* http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.

[105] Object Management Group (OMG) (2003). *OMG. MDA Guide Version 1.0.1.* http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.

[106] Object Management Group (OMG) (2008). *OMG. MOF Model To Text Transformation Language. Version 1.0.* http://www.omg.org/spec/MOFM2T/.

[107] Object Management Group (OMG) (2010). *OMG. UML 2.3.1 Superstructure specification.* http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[OMG] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation. Version 1.1.* Object Management Group. http://www.omg.org/spec/QVT/1.1/.

[109] Papadopoulos, G. A. and Arbab, F. (1998). Coordination models and languages. *Advances in Computers*, 46:329–400.

[110] Pérez-Castillo, R., de Guzmán, I. G. R., and Piattini, M. (2011). Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532.

[111] Pinheiro, F. A. C. and Goguen, J. A. (1996). An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64.

[112] Ramesh, B. and Dhar, V. (1992). Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510.

[113] Ramos, R., Barais, O., and Jézéquel, J.-M. (2007). Matching model-snippets. In *Proc. of MODELS'07*, volume 4735 of *LNCS*, pages 121–135. Springer.

[114] Razavi, A. and Kontogiannis, K. (2012). Partial evaluation of model transformations. In *Proc. of ICSE'12*, pages 562–572. IEEE.

[115] Richters, M. and Gogolla, M. (2002). OCL: Syntax, semantics, and tools. In *Object Modeling with the OCL*, volume 2263 of *LNCS*, pages 42–68. Springer.

[116] Rivera, J. E., Durán, F., and Vallecillo, A. (2009a). A Graphical Approach for Modeling Time-Dependent Behavior of DSLs. In *Proc. of VL/HCC'09*, pages 51–55. IEEE.

[117] Rivera, J. E., Durán, F., and Vallecillo, A. (2009b). Formal specification and analysis of domain specific models using maude. *Simulation*, 85(11-12):778–792.

[118] Rivera, J. E., Ruiz-Gonzalez, D., Lopez, F., Bautista, J., and Vallecillo, A. (2009c). Orchestrating ATL Model Transformations. In *Proc. of the MtATL 2009 Workshop*, pages 34–46.

[119] Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164.

[120] Sadalage, P. J. and Fowler, M. (2012). *NoSQL Distilled*. Addison-Wesley Professional.

[121] Sánchez Cuadrado, J. (2012). Towards a family of model transformation languages. In *Proc. of ICMT 2012*, volume 7307 of *LNCS*, pages 176–191. Springer.

[122] Sánchez Cuadrado, J., García Molina, J., and Menárguez Tortosa, M. (2006). RubyTL: A practical, extensible transformation language. In *Proc. of ECMDA-FA'06*, volume 4066 of *LNCS*, pages 158–172. Springer.

[123] Sánchez Cuadrado, J., Guerra, E., and de Lara, J. (2014). Uncovering errors in atl model transformations using static analysis and constraint solving. In *Proc. of ISSRE'14*. IEEE.

[124] Santiago, I., Jiménez, A., Bollati, V. A., Vara, J. M., de Castro, V., and Marcos, E. (2012). Model-driven engineering as a new landscape for traceability management: A systematic literature review. *Information and Software Technology*, 54(12):1340–1356.

[125] Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31.

[126] Sedlmeyer, R. L., Thompson, W. B., and Johnson, P. E. (1983). Knowledge-based fault localization in debugging. *Journal of Systems and Software*, 3(4):301–307.

[127] Seki, M. (2009). dRuby and Rinda: Implementation and Application of Distributed Ruby and its Parallel Coordination Mechanism. *International Journal of Parallel Programming*, 37(1):37–57.

[128] Sen, S., Baudry, B., and Mottu, J.-M. (2009). Automatic model generation strategies for model transformation testing. In *Proc. of ICMT'09*, volume 5563 of *LNCS*, pages 148–164. Springer.

[129] Sen, S., Mottu, J.-M., Tisi, M., and Cabot, J. (2012). Using models of partial knowledge to test model transformations. In *Proc. of ICMT'12*, volume 7307 of *LNCS*, pages 24–39. Springer.

[130] Stachowiak, H. (1973). *Allgemeine Modelltheorie.* Springer-Verlag.

[131] Stevens, P. (2007). A Landscape of Bidirectional Model Transformations. In *Proc. of GTTSE 2007*, volume 5235 of *LNCS*, pages 408–424. Springer.

[132] Straeten, R. V. D., Mens, T., and Baelen, S. V. (2008). Challenges in model-driven software engineering. In *Models in Software Engineering*, volume 5421 of *LNCS*, pages 35–47. Springer.

[133] Stürmer, I., Conrad, M., Dörr, H., and Pepper, P. (2007). Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622–634.

[134] Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., and Varró, D. (2014). IncQuery-D: A distributed incremental model query framework in the cloud. In *Proc. of MODELS'14*, volume 8767 of *LNCS*, pages 653–669. Springer.

[135] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J. (2009). On the Use of Higher-Order Model Transformations. In *Proc. of ECMDA-FA'09*, volume 5562 of *LNCS*, pages 18–33. Springer.

[136] Tisi, M., Martínez, S., Jouault, F., and Cabot, J. (2011a). Refining models with rule-based model transformations. Research Report RR-7582. https://hal.inria.fr/inria-00580033v1/document.

[137] Tisi, M., Perez, S. M., and Choura, H. (2013). Parallel execution of ATL transformation rules. In *Proc. of MODELS'13*, volume 8107 of *LNCS*, pages 656–672. Springer.

[138] Tisi, M., Perez, S. M., Jouault, F., and Cabot, J. (2011b). Lazy execution of model-to-model transformations. In *Proc. of MoDELS 2011*, volume 6981 of *LNCS*, pages 32–46. Springer.

[139] Tiso, A., Reggio, G., and Leotta, M. (2012). Early Experiences on Model Transformation Testing. In *Proc. of the AMT'12 Workshop @ MODELS'12*, pages 15–20. ACM.

[140] Troya, J., Burgueño, L., Wimmer, M., and Vallecillo, A. (2014). Mutations in ATL transformations and their identification with matching tables. Technical report. http://atenea.lcc.uma.es/Descargas/MTB/Mutations/TechReport.pdf.

[141] Troya, J. and Vallecillo, A. (2011). A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10:5:1–29.

[142] Ujhelyi, Z., Horváth, Á., and Varró, D. (2012). Dynamic backward slicing of model transformations. In *Proc. of ICST'12*, pages 1–10. IEEE.

[143] Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., and Hamann, L. (2012). Formal specification and testing of model transformations. In *Formal Methods for Model-Driven Engineering (SFM)*, volume 7320 of *LNCS*, pages 399—-437. Springer.

[144] van Amstel, M., Bosems, S., Kurtev, I., and Pires, L. F. (2011). Performance in model transformations: Experiments with ATL and QVT. In *Proc. of ICMT 2011*, volume 6707 of *LNCS*, pages 198–212. Springer.

[145] Varró, D. and Pataricza, A. (2004). Generic and meta-transformations for model transformation engineering. In *Proc. of UML'04*, volume 3273 of *LNCS*, pages 290–304. Springer.

[146] von Pilgrim, J., Vanhooff, B., Schulz-Gerlach, I., and Berbers, Y. (2008). Constructing and visualizing transformation chains. In *Proc. of ECMDA-FA 2008*, volume 5095 of *LNCS*, pages 17–32. Springer.

[147] Wagelaar, D., Tisi, M., Cabot, J., and Jouault, F. (2011). Towards a general composition semantics for rule-based model transformation. In *Proc. of MODELS 2011*, volume 6981 of *LNCS*, pages 623–637. Springer.

[148] Wells, G. (2006). New and improved: Linda in java. *Science of Computer Programming*, 59(1-2):82–96.

[149] Wells, G., Chalmers, A., and Clayton, P. G. (2004). Linda implementations in java for concurrent systems. *Concurrency – Practice and Experience*, 16(10):1005–1022.

[150] Wimmer, M., Kappel, G., Schönböck, J., Kusel, A., Retschitzegger, W., and Schwinger, W. (2009). A Petri Net based debugging environment for QVT Relations. In *Proc. of ASE'09*, pages 3–14. IEEE.

[151] Wimmer, M., Martínez, S., Jouault, F., and Cabot, J. (2012). A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, 11(2):1–40.

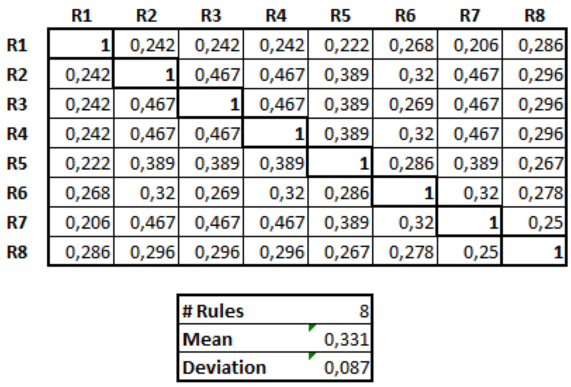
[152] Wischenbart, M., Mitsch, S., Kapsammer, E., Kusel, A., Pröll, B., Retschitzegger, W., Schwinger, W., Schönböck, J., Wimmer, M., and Lechner, S. (2012). User profile integration made easy: model-driven extraction and transformation of social network schemas. In *Companion Proceedings of the 21st World Wide Web Conference (WWW 2012)*, pages 939–948. ACM.

[153] Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P., and Vouk, M. A. (2006). On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253.

Appendix A Similarity Matrixes

This appendix shows some examples of similarity matrixes for transformations in the ATL Zoo [61] and some more. Due to the number of rules and thus, the size of the table, the similarity matrix of some transformations cannot be displayed here. Nevertheless, all of them are available in our website [24].

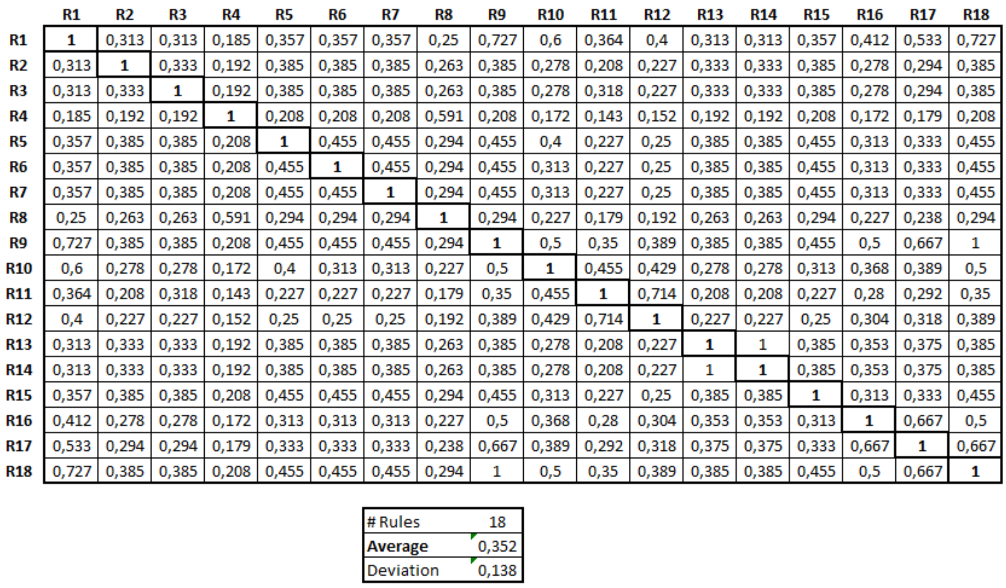
The similarity matrixes—which have been obtained with our tool—are shown in Figures A.2–A.32. The documentation, which also includes the transformation and the metamodels involved, can be found at the ATL Zoo website.



	R1	R2	R3	R4	R5	R6	R7	R8
R1	1	0,242	0,242	0,242	0,222	0,268	0,206	0,286
R2	0,242	1	0,467	0,467	0,389	0,32	0,467	0,296
R3	0,242	0,467	1	0,467	0,389	0,269	0,467	0,296
R4	0,242	0,467	0,467	1	0,389	0,32	0,467	0,296
R5	0,222	0,389	0,389	0,389	1	0,286	0,389	0,267
R6	0,268	0,32	0,269	0,32	0,286	1	0,32	0,278
R7	0,206	0,467	0,467	0,467	0,389	0,32	1	0,25
R8	0,286	0,296	0,296	0,296	0,267	0,278	0,25	1

# Rules	8
Mean	0,331
Deviation	0,087

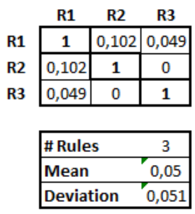
Fig. A.1 Similarity Matrix for the ATOM2XML example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18
R1	**1**	0,313	0,313	0,185	0,357	0,357	0,357	0,25	0,727	0,6	0,364	0,4	0,313	0,313	0,357	0,412	0,533	0,727
R2	0,313	**1**	0,333	0,192	0,385	0,385	0,385	0,263	0,385	0,278	0,208	0,227	0,333	0,333	0,385	0,278	0,294	0,385
R3	0,313	0,333	**1**	0,192	0,385	0,385	0,385	0,263	0,385	0,278	0,318	0,227	0,333	0,333	0,385	0,278	0,294	0,385
R4	0,185	0,192	0,192	**1**	0,208	0,208	0,208	0,591	0,208	0,172	0,143	0,152	0,192	0,192	0,208	0,172	0,179	0,208
R5	0,357	0,385	0,385	0,208	**1**	0,455	0,455	0,294	0,455	0,4	0,227	0,25	0,385	0,385	0,455	0,313	0,333	0,455
R6	0,357	0,385	0,385	0,208	0,455	**1**	0,455	0,294	0,455	0,313	0,227	0,25	0,385	0,385	0,455	0,313	0,333	0,455
R7	0,357	0,385	0,385	0,208	0,455	0,455	**1**	0,294	0,455	0,313	0,227	0,25	0,385	0,385	0,455	0,313	0,333	0,455
R8	0,25	0,263	0,263	0,591	0,294	0,294	0,294	**1**	0,294	0,227	0,179	0,192	0,263	0,263	0,294	0,227	0,238	0,294
R9	0,727	0,385	0,385	0,208	0,455	0,455	0,455	0,294	**1**	0,5	0,35	0,389	0,385	0,385	0,455	0,5	0,667	1
R10	0,6	0,278	0,278	0,172	0,4	0,313	0,313	0,227	0,5	**1**	0,455	0,429	0,278	0,278	0,313	0,368	0,389	0,5
R11	0,364	0,208	0,318	0,143	0,227	0,227	0,227	0,179	0,35	0,455	**1**	0,714	0,208	0,208	0,227	0,28	0,292	0,35
R12	0,4	0,227	0,227	0,152	0,25	0,25	0,25	0,192	0,389	0,429	0,714	**1**	0,227	0,227	0,25	0,304	0,318	0,389
R13	0,313	0,333	0,333	0,192	0,385	0,385	0,385	0,263	0,385	0,278	0,208	0,227	**1**	1	0,385	0,353	0,375	0,385
R14	0,313	0,333	0,333	0,192	0,385	0,385	0,385	0,263	0,385	0,278	0,208	0,227	1	**1**	0,385	0,353	0,375	0,385
R15	0,357	0,385	0,385	0,208	0,455	0,455	0,455	0,294	0,455	0,313	0,227	0,25	0,385	0,385	**1**	0,313	0,333	0,455
R16	0,412	0,278	0,278	0,172	0,313	0,313	0,313	0,227	0,5	0,368	0,28	0,304	0,353	0,353	0,313	**1**	0,667	0,5
R17	0,533	0,294	0,294	0,179	0,333	0,333	0,333	0,238	0,667	0,389	0,292	0,318	0,375	0,375	0,333	0,667	**1**	0,667
R18	0,727	0,385	0,385	0,208	0,455	0,455	0,455	0,294	1	0,5	0,35	0,389	0,385	0,385	0,455	0,5	0,667	**1**

# Rules	18
Average	0,352
Deviation	0,138

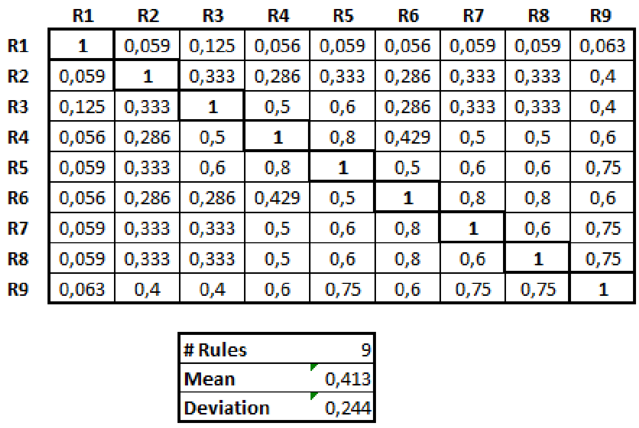
Fig. A.2 Similarity Matrix for the ATL2Problem example.



	R1	R2	R3
R1	**1**	0,102	0,049
R2	0,102	**1**	0
R3	0,049	0	**1**

# Rules	3
Mean	0,05
Deviation	0,051

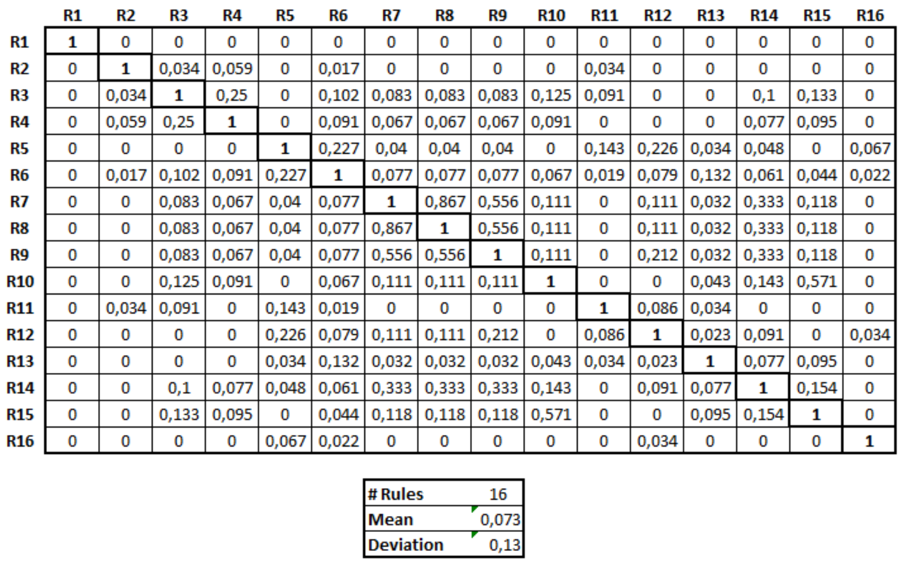
Fig. A.3 Similarity Matrix for the ATOM2RSS example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9
R1	**1**	0,059	0,125	0,056	0,059	0,056	0,059	0,059	0,063
R2	0,059	**1**	0,333	0,286	0,333	0,286	0,333	0,333	0,4
R3	0,125	0,333	**1**	0,5	0,6	0,286	0,333	0,333	0,4
R4	0,056	0,286	0,5	**1**	0,8	0,429	0,5	0,5	0,6
R5	0,059	0,333	0,6	0,8	**1**	0,5	0,6	0,6	0,75
R6	0,056	0,286	0,286	0,429	0,5	**1**	0,8	0,8	0,6
R7	0,059	0,333	0,333	0,5	0,6	0,8	**1**	0,6	0,75
R8	0,059	0,333	0,333	0,5	0,6	0,8	0,6	**1**	0,75
R9	0,063	0,4	0,4	0,6	0,75	0,6	0,75	0,75	**1**

# Rules	9
Mean	0,413
Deviation	0,244

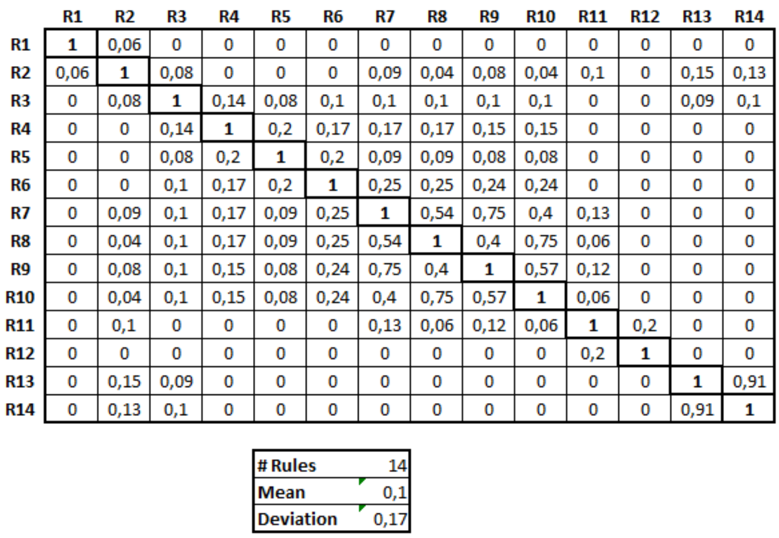
Fig. A.4 Similarity Matrix for the BibTex2DocBook example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
R1	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R2	0	**1**	0,034	0,059	0	0,017	0	0	0	0	0,034	0	0	0	0	0
R3	0	0,034	**1**	0,25	0	0,102	0,083	0,083	0,083	0,125	0,091	0	0	0,1	0,133	0
R4	0	0,059	0,25	**1**	0	0,091	0,067	0,067	0,067	0,091	0	0	0	0,077	0,095	0
R5	0	0	0	0	**1**	0,227	0,04	0,04	0,04	0	0,143	0,226	0,034	0,048	0	0,067
R6	0	0,017	0,102	0,091	0,227	**1**	0,077	0,077	0,077	0,067	0,019	0,079	0,132	0,061	0,044	0,022
R7	0	0	0,083	0,067	0,04	0,077	**1**	0,867	0,556	0,111	0	0,111	0,032	0,333	0,118	0
R8	0	0	0,083	0,067	0,04	0,077	0,867	**1**	0,556	0,111	0	0,111	0,032	0,333	0,118	0
R9	0	0	0,083	0,067	0,04	0,077	0,556	0,556	**1**	0,111	0	0,212	0,032	0,333	0,118	0
R10	0	0	0,125	0,091	0	0,067	0,111	0,111	0,111	**1**	0	0	0,043	0,143	0,571	0
R11	0	0,034	0,091	0	0,143	0,019	0	0	0	0	**1**	0,086	0,034	0	0	0
R12	0	0	0	0	0,226	0,079	0,111	0,111	0,212	0	0,086	**1**	0,023	0,091	0	0,034
R13	0	0	0	0	0,034	0,132	0,032	0,032	0,032	0,043	0,034	0,023	**1**	0,077	0,095	0
R14	0	0	0,1	0,077	0,048	0,061	0,333	0,333	0,333	0,143	0	0,091	0,077	**1**	0,154	0
R15	0	0	0,133	0,095	0	0,044	0,118	0,118	0,118	0,571	0	0	0,095	0,154	**1**	0
R16	0	0	0	0	0,067	0,022	0	0	0	0	0	0,034	0	0	0	**1**

# Rules	16
Mean	0,073
Deviation	0,13

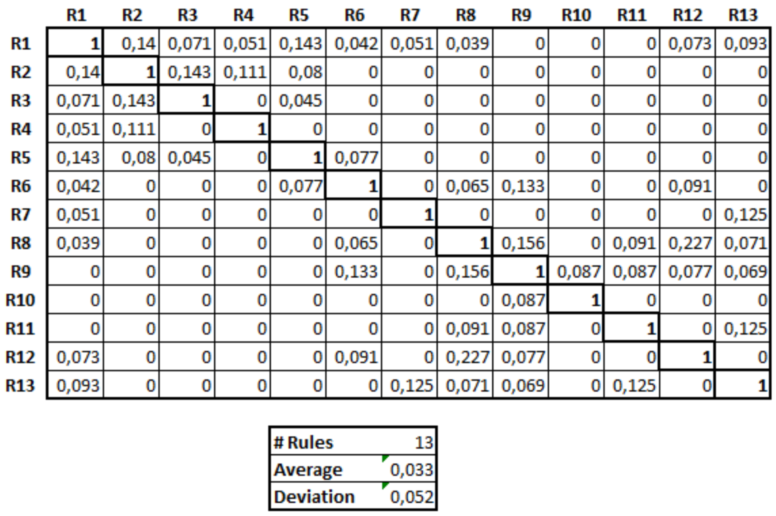
Fig. A.5 Similarity Matrix for the CPL2SPL example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14
R1	**1**	0,06	0	0	0	0	0	0	0	0	0	0	0	0
R2	0,06	**1**	0,08	0	0	0	0,09	0,04	0,08	0,04	0,1	0	0,15	0,13
R3	0	0,08	**1**	0,14	0,08	0,1	0,1	0,1	0,1	0,1	0	0	0,09	0,1
R4	0	0	0,14	**1**	0,2	0,17	0,17	0,17	0,15	0,15	0	0	0	0
R5	0	0	0,08	0,2	**1**	0,2	0,09	0,09	0,08	0,08	0	0	0	0
R6	0	0	0,1	0,17	0,2	**1**	0,25	0,25	0,24	0,24	0	0	0	0
R7	0	0,09	0,1	0,17	0,09	0,25	**1**	0,54	0,75	0,4	0,13	0	0	0
R8	0	0,04	0,1	0,17	0,09	0,25	0,54	**1**	0,4	0,75	0,06	0	0	0
R9	0	0,08	0,1	0,15	0,08	0,24	0,75	0,4	**1**	0,57	0,12	0	0	0
R10	0	0,04	0,1	0,15	0,08	0,24	0,4	0,75	0,57	**1**	0,06	0	0	0
R11	0	0,1	0	0	0	0	0,13	0,06	0,12	0,06	**1**	0,2	0	0
R12	0	0	0	0	0	0	0	0	0	0	0,2	**1**	0	0
R13	0	0,15	0,09	0	0	0	0	0	0	0	0	0	**1**	0,91
R14	0	0,13	0,1	0	0	0	0	0	0	0	0	0	0,91	**1**

# Rules	14
Mean	0,1
Deviation	0,17

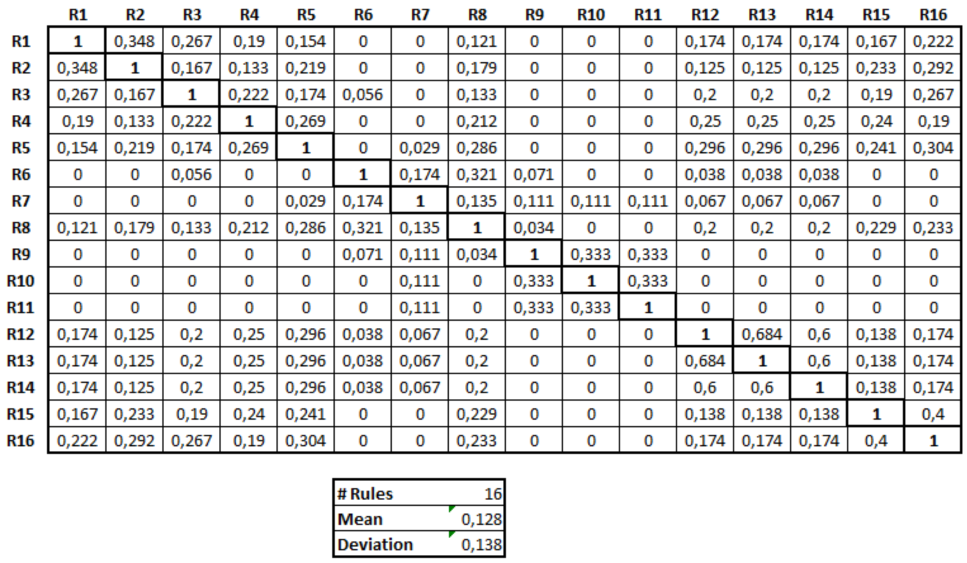
Fig. A.6 Similarity Matrix for the ECORE2USE example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
R1	**1**	0,14	0,071	0,051	0,143	0,042	0,051	0,039	0	0	0	0,073	0,093
R2	0,14	**1**	0,143	0,111	0,08	0	0	0	0	0	0	0	0
R3	0,071	0,143	**1**	0	0,045	0	0	0	0	0	0	0	0
R4	0,051	0,111	0	**1**	0	0	0	0	0	0	0	0	0
R5	0,143	0,08	0,045	0	**1**	0,077	0	0	0	0	0	0	0
R6	0,042	0	0	0	0,077	**1**	0	0,065	0,133	0	0	0,091	0
R7	0,051	0	0	0	0	0	**1**	0	0	0	0	0	0,125
R8	0,039	0	0	0	0	0,065	0	**1**	0,156	0	0,091	0,227	0,071
R9	0	0	0	0	0	0,133	0	0,156	**1**	0,087	0,087	0,077	0,069
R10	0	0	0	0	0	0	0	0	0,087	**1**	0	0	0
R11	0	0	0	0	0	0	0	0,091	0,087	0	**1**	0	0,125
R12	0,073	0	0	0	0	0,091	0	0,227	0,077	0	0	**1**	0
R13	0,093	0	0	0	0	0	0,125	0,071	0,069	0	0,125	0	**1**

# Rules	13
Average	0,033
Deviation	0,052

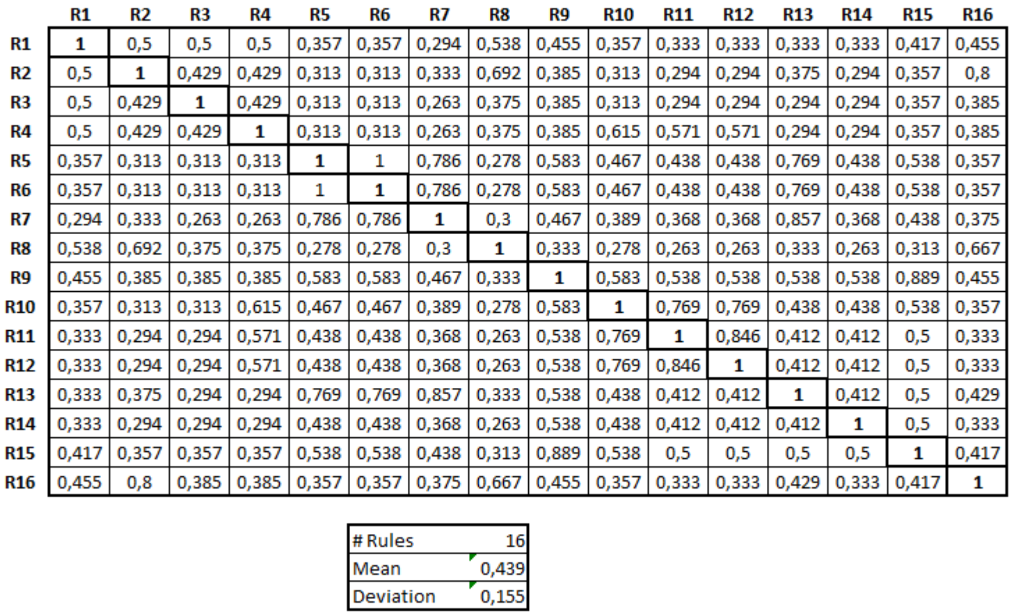
Fig. A.7 Similarity Matrix for the IEEE14712MoDAF example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
R1	**1**	0,348	0,267	0,19	0,154	0	0	0,121	0	0	0	0,174	0,174	0,174	0,167	0,222
R2	0,348	**1**	0,167	0,133	0,219	0	0	0,179	0	0	0	0,125	0,125	0,125	0,233	0,292
R3	0,267	0,167	**1**	0,222	0,174	0,056	0	0,133	0	0	0	0,2	0,2	0,2	0,19	0,267
R4	0,19	0,133	0,222	**1**	0,269	0	0	0,212	0	0	0	0,25	0,25	0,25	0,24	0,19
R5	0,154	0,219	0,174	0,269	**1**	0	0,029	0,286	0	0	0	0,296	0,296	0,296	0,241	0,304
R6	0	0	0,056	0	0	**1**	0,174	0,321	0,071	0	0	0,038	0,038	0,038	0	0
R7	0	0	0	0	0,029	0,174	**1**	0,135	0,111	0,111	0,111	0,067	0,067	0,067	0	0
R8	0,121	0,179	0,133	0,212	0,286	0,321	0,135	**1**	0,034	0	0	0,2	0,2	0,2	0,229	0,233
R9	0	0	0	0	0	0,071	0,111	0,034	**1**	0,333	0,333	0	0	0	0	0
R10	0	0	0	0	0	0	0,111	0	0,333	**1**	0,333	0	0	0	0	0
R11	0	0	0	0	0	0	0,111	0	0,333	0,333	**1**	0	0	0	0	0
R12	0,174	0,125	0,2	0,25	0,296	0,038	0,067	0,2	0	0	0	**1**	0,684	0,6	0,138	0,174
R13	0,174	0,125	0,2	0,25	0,296	0,038	0,067	0,2	0	0	0	0,684	**1**	0,6	0,138	0,174
R14	0,174	0,125	0,2	0,25	0,296	0,038	0,067	0,2	0	0	0	0,6	0,6	**1**	0,138	0,174
R15	0,167	0,233	0,19	0,24	0,241	0	0	0,229	0	0	0	0,138	0,138	0,138	**1**	0,4
R16	0,222	0,292	0,267	0,19	0,304	0	0	0,233	0	0	0	0,174	0,174	0,174	0,4	**1**

# Rules	16
Mean	0,128
Deviation	0,138

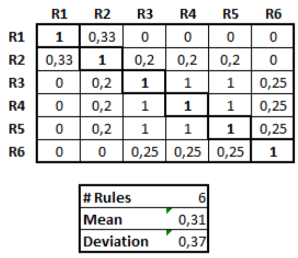
Fig. A.8 Similarity Matrix for the KM32OWLF example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
R1	**1**	0,5	0,5	0,5	0,357	0,357	0,294	0,538	0,455	0,357	0,333	0,333	0,333	0,333	0,417	0,455
R2	0,5	**1**	0,429	0,429	0,313	0,313	0,333	0,692	0,385	0,313	0,294	0,294	0,375	0,294	0,357	0,8
R3	0,5	0,429	**1**	0,429	0,313	0,313	0,263	0,375	0,385	0,313	0,294	0,294	0,294	0,294	0,357	0,385
R4	0,5	0,429	0,429	**1**	0,313	0,313	0,263	0,375	0,385	0,615	0,571	0,571	0,294	0,294	0,357	0,385
R5	0,357	0,313	0,313	0,313	**1**	1	0,786	0,278	0,583	0,467	0,438	0,438	0,769	0,438	0,538	0,357
R6	0,357	0,313	0,313	0,313	1	**1**	0,786	0,278	0,583	0,467	0,438	0,438	0,769	0,438	0,538	0,357
R7	0,294	0,333	0,263	0,263	0,786	0,786	**1**	0,3	0,467	0,389	0,368	0,368	0,857	0,368	0,438	0,375
R8	0,538	0,692	0,375	0,375	0,278	0,278	0,3	**1**	0,333	0,278	0,263	0,263	0,333	0,263	0,313	0,667
R9	0,455	0,385	0,385	0,385	0,583	0,583	0,467	0,333	**1**	0,583	0,538	0,538	0,538	0,538	0,889	0,455
R10	0,357	0,313	0,313	0,615	0,467	0,467	0,389	0,278	0,583	**1**	0,769	0,769	0,438	0,438	0,538	0,357
R11	0,333	0,294	0,294	0,571	0,438	0,438	0,368	0,263	0,538	0,769	**1**	0,846	0,412	0,412	0,5	0,333
R12	0,333	0,294	0,294	0,571	0,438	0,438	0,368	0,263	0,538	0,769	0,846	**1**	0,412	0,412	0,5	0,333
R13	0,333	0,375	0,294	0,294	0,769	0,769	0,857	0,333	0,538	0,438	0,412	0,412	**1**	0,412	0,5	0,429
R14	0,333	0,294	0,294	0,294	0,438	0,438	0,368	0,263	0,538	0,438	0,412	0,412	0,412	**1**	0,5	0,333
R15	0,417	0,357	0,357	0,357	0,538	0,538	0,438	0,313	0,889	0,538	0,5	0,5	0,5	0,5	**1**	0,417
R16	0,455	0,8	0,385	0,385	0,357	0,357	0,375	0,667	0,455	0,357	0,333	0,333	0,429	0,333	0,417	**1**

# Rules	16
Mean	0,439
Deviation	0,155

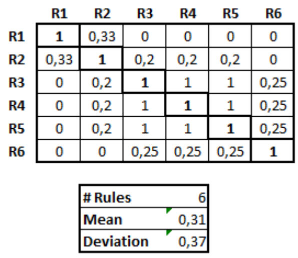
Fig. A.9 Similarity Matrix for the KM32Problem example.



	R1	R2	R3	R4	R5	R6
R1	**1**	0,33	0	0	0	0
R2	0,33	**1**	0,2	0,2	0,2	0
R3	0	0,2	**1**	1	1	0,25
R4	0	0,2	1	**1**	1	0,25
R5	0	0,2	1	1	**1**	0,25
R6	0	0	0,25	0,25	0,25	**1**

# Rules	6
Mean	0,31
Deviation	0,37

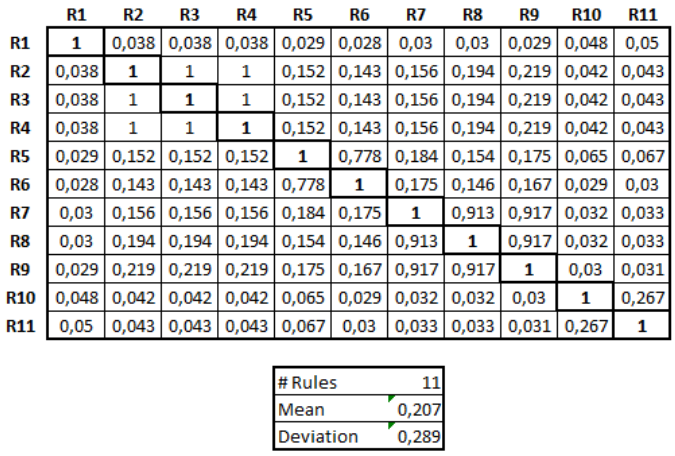
Fig. A.10 Similarity Matrix for the Measure2Table example.



	R1	R2	R3	R4	R5	R6
R1	**1**	0,33	0	0	0	0
R2	0,33	**1**	0,2	0,2	0,2	0
R3	0	0,2	**1**	1	1	0,25
R4	0	0,2	1	**1**	1	0,25
R5	0	0,2	1	1	**1**	0,25
R6	0	0	0,25	0,25	0,25	**1**

# Rules	6
Mean	0,31
Deviation	0,37

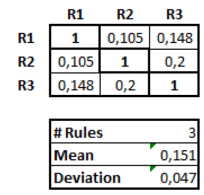
Fig. A.11 Similarity Matrix for the Measure2XHTML example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
R1	**1**	0,038	0,038	0,038	0,029	0,028	0,03	0,03	0,029	0,048	0,05
R2	0,038	**1**	1	1	0,152	0,143	0,156	0,194	0,219	0,042	0,043
R3	0,038	1	**1**	1	0,152	0,143	0,156	0,194	0,219	0,042	0,043
R4	0,038	1	1	**1**	0,152	0,143	0,156	0,194	0,219	0,042	0,043
R5	0,029	0,152	0,152	0,152	**1**	0,778	0,184	0,154	0,175	0,065	0,067
R6	0,028	0,143	0,143	0,143	0,778	**1**	0,175	0,146	0,167	0,029	0,03
R7	0,03	0,156	0,156	0,156	0,184	0,175	**1**	0,913	0,917	0,032	0,033
R8	0,03	0,194	0,194	0,194	0,154	0,146	0,913	**1**	0,917	0,032	0,033
R9	0,029	0,219	0,219	0,219	0,175	0,167	0,917	0,917	**1**	0,03	0,031
R10	0,048	0,042	0,042	0,042	0,065	0,029	0,032	0,032	0,03	**1**	0,267
R11	0,05	0,043	0,043	0,043	0,067	0,03	0,033	0,033	0,031	0,267	**1**

# Rules	11
Mean	0,207
Deviation	0,289

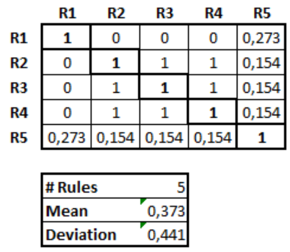
Fig. A.12 Similarity Matrix for the MySQL2KM3 example.



	R1	R2	R3
R1	**1**	0,105	0,148
R2	0,105	**1**	0,2
R3	0,148	0,2	**1**

# Rules	3
Mean	0,151
Deviation	0,047

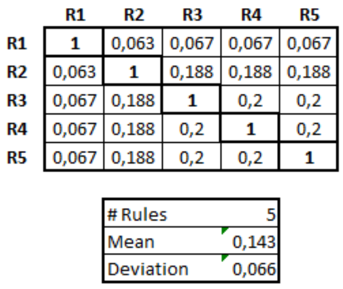
Fig. A.13 Similarity Matrix for the PathExp2PetriNet example.



	R1	R2	R3	R4	R5
R1	**1**	0	0	0	0,273
R2	0	**1**	1	1	0,154
R3	0	1	**1**	1	0,154
R4	0	1	1	**1**	0,154
R5	0,273	0,154	0,154	0,154	**1**

# Rules	5
Mean	0,373
Deviation	0,441

Fig. A.14 Similarity Matrix for the PathExp2TextualPath example.



	R1	R2	R3	R4	R5
R1	**1**	0,063	0,067	0,067	0,067
R2	0,063	**1**	0,188	0,188	0,188
R3	0,067	0,188	**1**	0,2	0,2
R4	0,067	0,188	0,2	**1**	0,2
R5	0,067	0,188	0,2	0,2	**1**

# Rules	5
Mean	0,143
Deviation	0,066

Fig. A.15 Similarity Matrix for the PetriNet2Grafcet example.

	R1	R2	R3
R1	1	0,222	0,211
R2	0,222	1	0,4
R3	0,211	0,4	1

# Rules	3
Mean	0,278
Deviation	0,106

Fig. A.16 Similarity Matrix for the PetriNet2PathExp example.

	R1	R2	R3	R4
R1	1	0,129	0,129	0,135
R2	0,129	1	0,222	0,208
R3	0,129	0,222	1	0,208
R4	0,135	0,208	0,208	1

# Rules	4
Mean	0,172
Deviation	0,045

Fig. A.17 Similarity Matrix for the PetriNet2PNML example.

	R1	R2	R3	R4	R5
R1	1	0,563	0,467	0,368	0,368
R2	0,563	1	0,583	0,533	0,533
R3	0,467	0,583	1	0,667	0,667
R4	0,368	0,533	0,667	1	0,6
R5	0,368	0,533	0,667	0,6	1

# Rules	5
Mean	0,535
Deviation	0,107

Fig. A.18 Similarity Matrix for the PetriNet2XML example.

	R1	R2	R3	R4	R5
R1	**1**	0,28	0,28	0,2	0,2
R2	0,28	**1**	0,4	0,231	0,231
R3	0,28	0,4	**1**	0,231	0,231
R4	0,2	0,231	0,231	**1**	0,565
R5	0,2	0,231	0,231	0,565	**1**

# Rules	5
Mean	0,285
Deviation	0,115

Fig. A.19 Similarity Matrix for the PNML2PetriNet example.

	R1	R2	R3	R4
R1	**1**	0,567	0,567	0,548
R2	0,567	**1**	0,905	0,864
R3	0,567	0,905	**1**	0,864
R4	0,548	0,864	0,864	**1**

# Rules	4
Mean	0,719
Deviation	0,174

Fig. A.20 Similarity Matrix for the PNML2XML example.

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14
R1	**1**	0,105	0	0	0	0	0	0	0,097	0	0	0	0	0
R2	0,105	**1**	0,034	0,034	0,034	0,05	0,048	0	0	0,053	0	0,059	0,059	0
R3	0	0,034	**1**	0,667	0,667	0,192	0,231	0	0	0,034	0	0,037	0,037	0
R4	0	0,034	0,667	**1**	0,667	0,148	0,231	0	0	0,034	0	0,077	0,077	0
R5	0	0,034	0,667	0,667	**1**	0,148	0,231	0	0,024	0,071	0,027	0,037	0,037	0
R6	0	0,05	0,192	0,148	0,148	**1**	0,353	0	0	0,05	0	0,056	0,056	0
R7	0	0,048	0,231	0,231	0,231	0,353	**1**	0,067	0	0,048	0	0,053	0,053	0
R8	0	0	0	0	0	0	0,067	**1**	0	0	0	0	0	0
R9	0,097	0	0	0	0,024	0	0	0	**1**	0,222	0,139	0,033	0,033	0,077
R10	0	0,053	0,034	0,034	0,071	0,05	0,048	0	0,222	**1**	0,12	0,125	0,125	0,154
R11	0	0	0	0	0,027	0	0	0	0,139	0,12	**1**	0,04	0,04	0,095
R12	0	0,059	0,037	0,077	0,037	0,056	0,053	0	0,033	0,125	0,04	**1**	0,455	0
R13	0	0,059	0,037	0,077	0,037	0,056	0,053	0	0,033	0,125	0,04	0,455	**1**	0
R14	0	0	0	0	0	0	0	0	0,077	0,154	0,095	0	0	**1**

# Rules	14
Mean	0,073
Deviation	0,136

Fig. A.21 Similarity Matrix for the R2ML2WSDL example.

	R1	R2	R3
R1	1	0,091	0,057
R2	0,091	1	0
R3	0,057	0	1

# Rules	3
Mean	0,049
Deviation	0,046

Fig. A.22 Similarity Matrix for the RSS2ATOM example.

	R1	R2	R3	R4
R1	1	0,257	0,267	0,241
R2	0,257	1	0,444	0,412
R3	0,267	0,444	1	0,636
R4	0,241	0,412	0,636	1

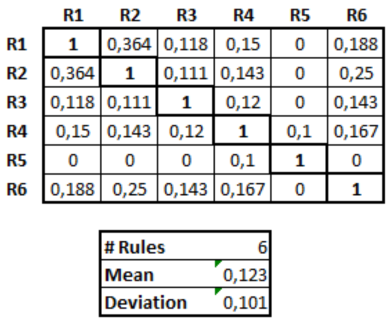
# Rules	4
Mean	0,376
Deviation	0,154

Fig. A.23 Similarity Matrix for the RSS2XML example.

	R1	R2	R3	R4	R5	R6	R7	R8
R1	1	0	0	0	0	0	0	0
R2	0	1	0,25	0	0	0	0	0
R3	0	0,25	1	0,2	0	0,25	0	0
R4	0	0	0,2	1	0,2	0,14	0,25	0,33
R5	0	0	0	0,2	1	0,11	0,17	0,2
R6	0	0	0,25	0,14	0,11	1	0,13	0,14
R7	0	0	0	0,25	0,17	0,13	1	0,25
R8	0	0	0	0,33	0,2	0,14	0,25	1

#Rules	8
Mean	0,094
Deviation	0,111

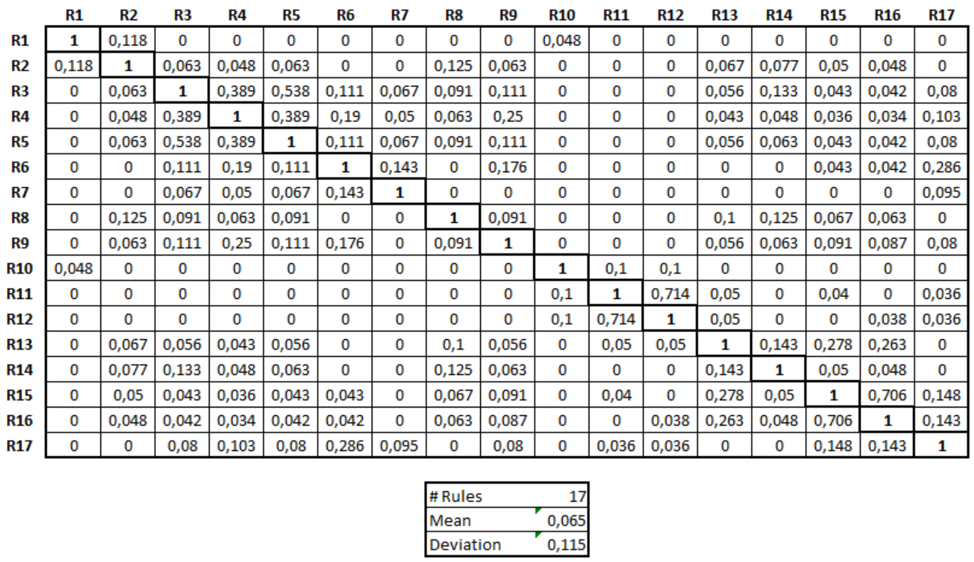
Fig. A.24 Similarity Matrix for the UML2ER example.



	R1	R2	R3	R4	R5	R6
R1	**1**	0,364	0,118	0,15	0	0,188
R2	0,364	**1**	0,111	0,143	0	0,25
R3	0,118	0,111	**1**	0,12	0	0,143
R4	0,15	0,143	0,12	**1**	0,1	0,167
R5	0	0	0	0,1	**1**	0
R6	0,188	0,25	0,143	0,167	0	**1**

# Rules	6
Mean	0,123
Deviation	0,101

Fig. A.25 Similarity Matrix for the XML2MySQL example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17
R1	**1**	0,118	0	0	0	0	0	0	0	0,048	0	0	0	0	0	0	0
R2	0,118	**1**	0,063	0,048	0,063	0	0	0,125	0,063	0	0	0	0,067	0,077	0,05	0,048	0
R3	0	0,063	**1**	0,389	0,538	0,111	0,067	0,091	0,111	0	0	0	0,056	0,133	0,043	0,042	0,08
R4	0	0,048	0,389	**1**	0,389	0,19	0,05	0,063	0,25	0	0	0	0,043	0,048	0,036	0,034	0,103
R5	0	0,063	0,538	0,389	**1**	0,111	0,067	0,091	0,111	0	0	0	0,056	0,063	0,043	0,042	0,08
R6	0	0	0,111	0,19	0,111	**1**	0,143	0	0,176	0	0	0	0	0	0,043	0,042	0,286
R7	0	0	0,067	0,05	0,067	0,143	**1**	0	0	0	0	0	0	0	0	0	0,095
R8	0	0,125	0,091	0,063	0,091	0	0	**1**	0,091	0	0	0	0,1	0,125	0,067	0,063	0
R9	0	0,063	0,111	0,25	0,111	0,176	0	0,091	**1**	0	0	0	0,056	0,063	0,091	0,087	0,08
R10	0,048	0	0	0	0	0	0	0	0	**1**	0,1	0,1	0	0	0	0	0
R11	0	0	0	0	0	0	0	0	0	0,1	**1**	0,714	0,05	0	0,04	0	0,036
R12	0	0	0	0	0	0	0	0	0	0,1	0,714	**1**	0,05	0	0	0,038	0,036
R13	0	0,067	0,056	0,043	0,056	0	0	0,1	0,056	0	0,05	0,05	**1**	0,143	0,278	0,263	0
R14	0	0,077	0,133	0,048	0,063	0	0	0,125	0,063	0	0	0	0,143	**1**	0,05	0,048	0
R15	0	0,05	0,043	0,036	0,043	0,043	0	0,067	0,091	0	0,04	0	0,278	0,05	**1**	0,706	0,148
R16	0	0,048	0,042	0,034	0,042	0,042	0	0,063	0,087	0	0	0,038	0,263	0,048	0,706	**1**	0,143
R17	0	0	0,08	0,103	0,08	0,286	0,095	0	0,08	0	0,036	0,036	0	0	0,148	0,143	**1**

# Rules	17
Mean	0,065
Deviation	0,115

Fig. A.26 Similarity Matrix for the WSDL2R2ML example.

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
R1	**1**	0,069	0,067	0,194	0,033	0,033	0,061	0,069	0,069	0,065
R2	0,069	**1**	0,182	0,091	0,2	0,2	0,143	0,2	0,2	0,167
R3	0,067	0,182	**1**	0,087	0,182	0,182	0,133	0,182	0,182	0,154
R4	0,194	0,091	0,087	**1**	0,091	0,091	0,12	0,143	0,143	0,13
R5	0,033	0,2	0,182	0,091	**1**	0,2	0,143	0,2	0,2	0,167
R6	0,033	0,2	0,182	0,091	0,2	**1**	0,143	0,2	0,2	0,167
R7	0,061	0,143	0,133	0,12	0,143	0,143	**1**	0,143	0,143	0,4
R8	0,069	0,2	0,182	0,143	0,2	0,2	0,143	**1**	0,2	0,167
R9	0,069	0,2	0,182	0,143	0,2	0,2	0,143	0,2	**1**	0,167
R10	0,065	0,167	0,154	0,13	0,167	0,167	0,4	0,167	0,167	**1**

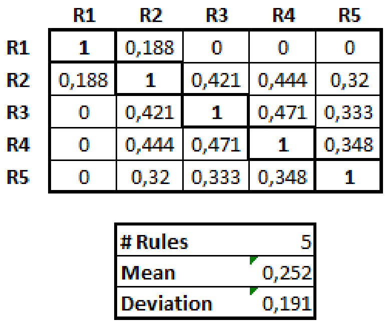
# Rules	10
Mean	0,149
Deviation	0,064

Fig. A.27 Similarity Matrix for the XML2ATOM example.

	R1	R2	R3	R4	R5
R1	**1**	0,2	0,2	0,286	0,286
R2	0,2	**1**	0,333	0,333	0,333
R3	0,2	0,333	**1**	0,333	0,333
R4	0,286	0,333	0,333	**1**	0,333
R5	0,286	0,333	0,333	0,333	**1**

# Rules	5
Mean	0,297
Deviation	0,055

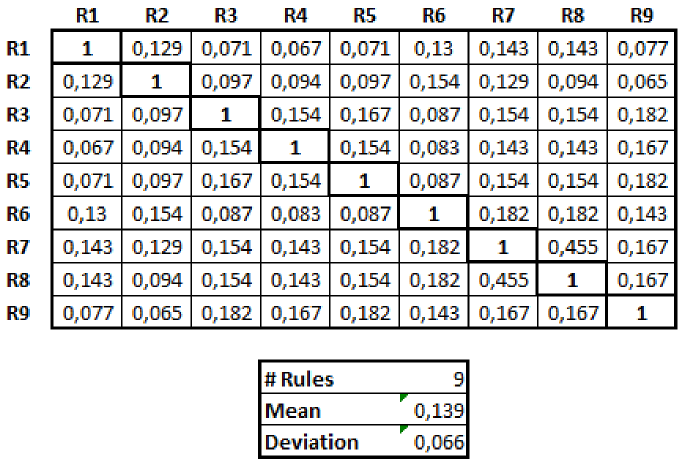
Fig. A.28 Similarity Matrix for the XML2PetriNet example.



	R1	R2	R3	R4	R5
R1	**1**	0,188	0	0	0
R2	0,188	**1**	0,421	0,444	0,32
R3	0	0,421	**1**	0,471	0,333
R4	0	0,444	0,471	**1**	0,348
R5	0	0,32	0,333	0,348	**1**

# Rules	5
Mean	0,252
Deviation	0,191

Fig. A.29 Similarity Matrix for the XML2PNML example.



	R1	R2	R3	R4	R5	R6	R7	R8	R9
R1	**1**	0,129	0,071	0,067	0,071	0,13	0,143	0,143	0,077
R2	0,129	**1**	0,097	0,094	0,097	0,154	0,129	0,094	0,065
R3	0,071	0,097	**1**	0,154	0,167	0,087	0,154	0,154	0,182
R4	0,067	0,094	0,154	**1**	0,154	0,083	0,143	0,143	0,167
R5	0,071	0,097	0,167	0,154	**1**	0,087	0,154	0,154	0,182
R6	0,13	0,154	0,087	0,083	0,087	**1**	0,182	0,182	0,143
R7	0,143	0,129	0,154	0,143	0,154	0,182	**1**	0,455	0,167
R8	0,143	0,094	0,154	0,143	0,154	0,182	0,455	**1**	0,167
R9	0,077	0,065	0,182	0,167	0,182	0,143	0,167	0,167	**1**

# Rules	9
Mean	0,139
Deviation	0,066

Fig. A.30 Similarity Matrix for the XML2RSS example.

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19
R1	**1**	0,231	0,111	0,091	0	0,111	0,091	0,188	0,118	0,083	0,125	0,125	0,125	0,125	0,182	0,111	0,111	0,25	0,188
R2	0,231	**1**	0,273	0,125	0	0,167	0,125	0,182	0,182	0,111	0,2	0,2	0,2	0,2	0,111	0,167	0,167	0,167	0,182
R3	0,111	0,273	**1**	0,1	0	0,286	0,222	0,133	0,214	0,091	0,231	0,231	0,143	0,143	0,091	0,125	0,125	0,125	0,133
R4	0,091	0,125	0,1	**1**	0	0,1	0,13	0,105	0,105	0,077	0,111	0,111	0,111	0,111	0,077	0,1	0,1	0,1	0,105
R5	0	0	0	0	**1**	0,182	0,133	0	0	0	0	0	0	0	0	0	0	0	0
R6	0,111	0,167	0,286	0,1	0,182	**1**	0,692	0,133	0,308	0,143	0,231	0,231	0,143	0,143	0,091	0,2	0,2	0,125	0,133
R7	0,091	0,125	0,222	0,13	0,133	0,692	**1**	0,105	0,235	0,12	0,176	0,176	0,111	0,111	0,077	0,158	0,158	0,1	0,105
R8	0,188	0,182	0,133	0,105	0	0,133	0,105	**1**	0,231	0,15	0,154	0,154	0,25	0,25	0,15	0,214	0,214	0,214	0,143
R9	0,118	0,182	0,214	0,105	0	0,308	0,235	0,231	**1**	0,15	0,25	0,25	0,25	0,25	0,095	0,214	0,308	0,133	0,143
R10	0,083	0,111	0,091	0,077	0	0,143	0,12	0,15	0,15	**1**	0,158	0,158	0,158	0,158	0,071	0,2	0,143	0,091	0,095
R11	0,125	0,2	0,231	0,111	0	0,231	0,176	0,154	0,25	0,158	**1**	0,273	0,167	0,167	0,1	0,143	0,143	0,143	0,154
R12	0,125	0,2	0,231	0,111	0	0,231	0,176	0,154	0,25	0,158	0,273	**1**	0,167	0,167	0,1	0,143	0,143	0,143	0,154
R13	0,125	0,2	0,143	0,111	0	0,143	0,111	0,25	0,25	0,158	0,167	0,167	**1**	0,273	0,1	0,143	0,231	0,143	0,154
R14	0,125	0,2	0,143	0,111	0	0,143	0,111	0,25	0,25	0,158	0,167	0,167	0,273	**1**	0,1	0,143	0,231	0,143	0,154
R15	0,182	0,111	0,091	0,077	0	0,091	0,077	0,15	0,095	0,071	0,1	0,1	0,1	0,1	**1**	0,143	0,143	0,143	0,15
R16	0,111	0,167	0,125	0,1	0	0,2	0,158	0,214	0,214	0,2	0,143	0,143	0,143	0,143	0,143	**1**	0,2	0,125	0,133
R17	0,111	0,167	0,125	0,1	0	0,2	0,158	0,214	0,308	0,143	0,143	0,143	0,231	0,231	0,143	0,2	**1**	0,125	0,133
R18	0,25	0,167	0,125	0,1	0	0,125	0,1	0,214	0,133	0,091	0,143	0,143	0,143	0,143	0,143	0,125	0,125	**1**	0,214
R19	0,188	0,182	0,133	0,105	0	0,133	0,105	0,143	0,143	0,095	0,154	0,154	0,154	0,154	0,15	0,133	0,133	0,214	**1**

# Rules	19
Mean	0,144
Deviation	0,079

Fig. A.31 Similarity Matrix for the XML2WSDL example.

	R1	R2	R3	R4	R5	R6	R7
R1	**1**	0,6	0,267	0,087	0	0,04	0
R2	0,6	**1**	0,235	0,074	0	0,034	0
R3	0,267	0,235	**1**	0,08	0	0,037	0
R4	0,087	0,074	0,08	**1**	0	0	0
R5	0	0	0	0	**1**	0	0
R6	0,04	0,034	0,037	0	0	**1**	0
R7	0	0	0	0	0	0	**1**

# Rules	7
Mean	0,069
Deviation	0,143

Fig. A.32 Similarity Matrix for the XSLT2XQuery example.

Appendix B
Resumen

La creciente complejidad del software originada por el progreso tecnológico ha fomentado el uso de modelos a la hora de desarrollar software. En sus orígenes, los modelos eran muy básicos y se utilizan principalmente como esquemas simplificados que ayudaban a los desarrolladores a entender sus programas. Con el paso del tiempo su uso se fue extendiendo hasta que una nueva disciplina llamada Ingeniería Dirigida por Modelos (cuyas siglas son MDE dado su nombre en inglés Model-Driven Engineering) se estableció. En el ámbito de la Ingeniería Dirigida por Modelos, junto con los modelos, las transformaciones de modelos cobran especial interés ya que permiten el análisis y manipulación de los mismos. Por lo tanto, es necesario estudiar aspectos como el rendimiento, la escalabilidad y la corrección de las transformaciones de modelos. Los motores de transformaciones de modelos actuales usan principalmente estrategias de ejecución secuencial para las cuales los modelos deben residir en memoria, por lo que su capacidad para transformar modelos grandes en entornos paralelos y distribuidos es limitada. Es más, las herramientas y lenguajes existentes en la actualidad ni siquiera permiten el tratamiento de modelos de tamaño mediano y obligan a los modelos que se encuentren ubicados en una sola máquina o peor aún, la mayoría de ellos, obligan a que vengan dados en un único archivo. Además, una vez que una transformación de modelos se ejecuta (ya sea de forma secuencial o en paralelo) hay una creciente necesidad de contar con métodos, mecanismos y herramientas para comprobar su corrección.

Nuestra contribución en esta tesis distingue dos aspectos. En primer lugar, presentamos una plataforma de ejecución que permite la ejecución en paralelo de transformaciones de modelos tanto out-place como in-place independientemente de que los modelos quepan en la memoria de una única máquina o no. Las transformaciones de modelos escritas en cualquier lenguaje de alto nivel pueden ser compiladas a dicha plataforma y ejecutadas en paralelo y de forma distribuida. Las principales ventajas son que se permite la reutilización de transformaciones de modelos existentes y que no obliga a los desarrolladores a aprender un nuevo lenguaje sino que pueden usar aquel o aquellos con los que se sientan cómodos.

Ya que los lenguajes de transformaciones de modelos más populares se basan en reglas, es decir, las transformaciones están formadas por un conjunto de reglas que especifican cómo los elementos deben ser transformados, la segunda contribución de esta tesis es un enfoque estático para localizar las reglas que puedan contener errores y que provocan así que la transformación no se comporte de la forma esperada. Los enfoques actuales que hacen una comprobación exhaustiva (tales como técnicas de model checking) requieren demasiado tiempo y memoria por lo que muchas veces son inviables. Nuestro enfoque no es capaz de probar con total certeza que una transformación sea correcta pero es útil para identificar errores en una fase muy temprana del desarrollo y de forma rápida.

Appendix C
Conclusiones y Contribuciones

En este apéndice se exponen en español las conclusiones y contribuciones a las que se ha llegado tras la realización de esta tesis doctoral.

En las primeras etapas de esta tesis doctoral, se estudió el estado del arte de la Ingeniería Dirigida por Modelos (MDE, por sus siglas en inglés) y se observó que las transformaciones de modelos no seguían la misma línea de evolución tecnológica que el resto de disciplinas y por lo tanto se estaban quedando desactualizadas. Las transformaciones de modelos carecen de los mecanismos de concurrencia adecuados y de las herramientas pertinentes para poder ser ejecutadas en paralelo, al igual que la mayoría del software que se desarrolla hoy en día. Por ello, la primera contribución de esta tesis ha sido el enfoque que se presenta en los capítulos 3 y 4 para la ejecución paralela de transformaciones de modelos para la cual se utilizan los conceptos y mecanismos del lenguaje de coordinación Linda.

Se ha implementado una capa software sobre arquitecturas existentes para el almacenamiento distribuido de datos (data-grids) que permiten la especificación de transformaciones de modelos en Java y que abstrae al desarrollador o ingeniero de las tecnologías y las estrategias de paralelización subyacentes. Además, hemos mostrado cómo representar metamodelos y modelos para ejecutar transformaciones de modelos en paralelo así como cómo se mantiene la trazabilidad en las transformaciones out-place. Basándonos en los resultados, la plataforma jLinTra parece bastante prometedora como la base de una infraestructura sobre la cual ejecutar transformaciones de modelos de forma eficiente que, combinada con un enfoque de paralelismo de datos, permite lograr la paralelización. La comparación de jLinTra con respecto otros motores de transformación secuencial conocidos reveló un decremento significativo del tiempo de ejecución, y la comparación con respecto a una solución paralela que usa paralelización de tareas mostró que la paralelización de datos, la cual jLinTra usa, parece funcionar mejor para las transformaciones de modelos presentadas en nuestros casos de estudio. Los experimentos realizados han mostrado que, además del buen rendimiento del modo out-place, el modo in-place también ha mejorado con respecto a otros motores de transformación de modelos in-place. Además, otro experimento que también hicimos fue implementar las transformaciones in-place como transformaciones out-place donde los elementos que no se modifican se copian tal cual y observamos que el modo in-place proporciona un mejor rendimiento y facilidad de uso.

Ya que nuestro enfoque depende de los data grids empleados, la evaluación también muestra que los data grids ofrecen un buen rendimiento y que no introducen penalizaciones significativas. Sin embargo, hay una gran diferencia en el tiempo que necesitan para acceder a los datos.

Para concluir podemos decir que jLinTra parece ser capaz de manejar y escalar grandes modelos al mismo tiempo que mejora el rendimiento de los motores de transformaciones de modelos actuales tanto secuenciales como paralelos.

La segunda contribución principal de esta tesis trata de resolver la necesidad que aparece tras la ejecución de una transformación de modelos de probar su corrección y de comprobar que el resultado obtenido es el esperado. En el capítulo 5 presentamos un enfoque estático para buscar el origen de los errores en las implementaciones de las transformaciones de modelos. Dada una transformación de modelos basada en reglas y un conjunto de restricciones OCL que especifican su comportamiento, nuestro enfoque extrae automáticamente los elementos de los metamodelos que ambos referencian y se obtiene el solapamiento existente entre cada regla y cada restricción. A continuación, se proporcionan tres tablas donde se refleja la relación entre reglas y restricciones. Dichas tablas son usadas para localizar qué reglas que pueden estar provocando que ciertas restricciones sean violadas. Nuestra evaluación muestra que el enfoque que presentamos es preciso para un gran conjunto de transformaciones de modelos. No obstante, como hay determinados casos en los que puede no ser útil, mediante el uso de matrices de similitud, un test automatizado e instantáneo de aptitud, es posible comprobar a-priori si nuestro enfoque proveerá información fiable para una determinada transformación de modelos o no.

La última contribución de esta tesis se presenta en el capítulo 6 donde ampliamos el alcance de nuestro enfoque a transformaciones de modelo-a-texto y de texto-a-modelo. Para ello hemos creado un metamodelo genérico que permite la representación de repositorios de texto como modelos conforme a dicho metamodelo. Una vez que el texto es representado como modelo, el problema de probar la corrección de transformaciones de modelo-a-texto y de texto-a-modelo queda reducido a un problema de corrección de transformaciones modelo-a-modelo. Como en el modelo que representa el texto existen atributos de tipo String cuyos valores deben ser analizados en muchos casos, hemos extendido OCL con operaciones adicionales para el manejo de cadenas.

La evaluación llevada a cabo consistió en evaluar el código Java generado por varias herramientas de uso extendido a partir de modelos UML. Dicha evaluación muestra que los generadores que actualmente se usan tienen un gran margen a mejora y documenta la necesidad de una disciplina para desarrollar transformaciones de modelo-a-texto y de texto-a-modelo y la importancia de la corrección de este tipo de transformaciones.