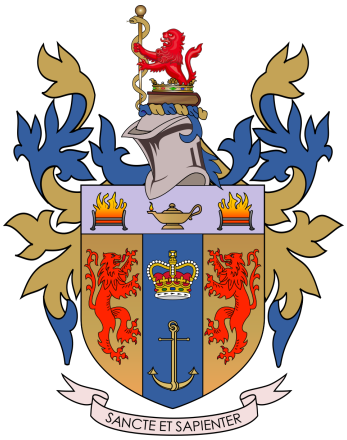
Synthesis of Model Transformations from Metamodels and Examples

Fang, Shichao

Awarding institution:
King's College London



Synthesis of Model Transformations from Metamodels and Examples

Shichao Fang

Department of Informatics

King's College London

Supervisors

Dr. Kevin Lano

Dr. Steffen Zschaler

This Thesis is Submitted for the Degree of Doctor of Philosophy

December 2022

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Dr Kevin Lano, for his invaluable guidance, advice, unwavering support, and continuous assistance throughout the completion of this thesis. Without his mentorship, I would not have been able to accomplish this significant milestone. Dr Lano's expertise, encouragement, and constructive feedback have been instrumental in shaping the quality and direction of my work.

I would also like to extend my sincere appreciation to my second supervisor, Dr Steffen Zschaler. His insightful recommendations, stimulating discussions, and scholarly insights have greatly enriched the depth and rigor of my research.

In addition to my supervisors, I would like to acknowledge the entire faculty and staff of the Department of Informatics at King's College London. Their contributions, resources, and academic environment have played a crucial role in fostering my academic growth.

I am profoundly thankful to my beloved parents for their years of unwavering support, encouragement, and belief in my potential. I take immense pride in making them proud. I will continue to be a son who brings them happiness and joy.

Finally, I would like to dedicate this thesis in loving memory of my dear grandmother, who sadly passed away during the COVID-19 pandemic. Although she was not able to witness this moment, I am sure she would have been proud of me. I will always miss her.

ABSTRACT

Model transformations are central elements of model-driven engineering (MDE). However, model transformation development requires a high level of expertise in particular model transformation languages, and model transformation specifications are often difficult to manually construct, due to the lack of tool support, and the dependencies involved in transformation rules.

In this thesis, we describe techniques for automatically or semi-automatically synthesising transformations from metamodels and examples, in order to reduce model transformation development costs and time, and improve model transformation quality.

We proposed two approaches for synthesising transformations from metamodels. The first approach is the Data Structure Similarity Approach, an exhaustive metamodel matching approach, which extracts correspondences between metamodels by only focusing on the type of features. The other approach is the Search-based Optimisation Approach, which uses an optimisation algorithm to extract correspondences from metamodels by data structure similarity, name syntax similarity, and name semantic similarity. The correspondence patterns between the classes and features of two metamodels are extracted by either of these two methods. To enable the production of specifications in multiple model transformation languages from correspondences, we introduced an intermediate language which uses a simplified transformation notation to express transformation specifications in a language-

independent manner, and defined the mapping rules from this intermediate language to different transformation languages.

We also investigated Model Transformation by Examples Approach. We used machine learning techniques to learn model transformation rules from datasets of examples, so that the trained model could generate target model from source model directly.

We evaluated our approaches on a range of cases of different kinds of transformation, and compared the model transformation accuracy and quality of our versions to the previously-developed manual versions of these cases.

Key words: model transformation, model-driven engineering, transformation synthesis, metamodel matching, model transformation by examples

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Model-Driven Engineering (MDE) has emerged as a foundational approach in contemporary software engineering, highlighting the utilization of models as primary assets across the software development lifecycle [1]. Central to the essence of MDE is the core concept of Model Transformation (MT), a process that provides the capability to migrate, refine, abstract and analyse models [2]. One prominent example of MT is the Statechart to Petri Net transformation [3]. This transformation involves transforming a Statechart diagram, which represents the behavior of a system in terms of states and transitions, into a Petri Net, a mathematical model used to describe concurrent and distributed systems. This example highlights how model transformations can facilitate the translation of a high-level behavioral model into a

formal mathematical representation, allowing for rigorous analysis and verification of system properties. [3], [4].

A MT specification defines the intended effect of the transformation in precise terms, but does not give an implementation. A MT language is specialised for implementing MT by defining a set of rules that identify how to transform between models. In principle MT can be produced from these specifications in different transformation languages. MT are often difficult to construct manually, and the definition of MT requires a high level of expertise in particular MT languages [5]. Large transformations are expensive to develop manually and can become unmanageably complex [6], [7]. These issues are due to the complex syntax and semantics of MT languages, and the lack of tool support for MT development (in contrast to facilities such as IDEs and user support forums for programming languages). In addition, transformations involve dependencies between MT rules, because of dependencies in the metamodels (eg., a *Table* depends on *Attributes*, in the relational data model). These dependencies are time-consuming to manually manage. Only a few works implement MT language synthesis [8], and these can only synthesise a single MT language [9]–[12]. However, different MT languages have different properties, therefore developers may need to produce more than one MT language in different situations. Due to the significant difference between MT languages, the learning cost is high.

Therefore, the difficulty of manual transformation development can be a signif-

icant barrier to the wider use of MDE, especially in an agile development context, where development effort needs to be focused on rapid delivery of functionality to customers, rather than on the production of development support tooling.

1.2 Overall Aim of the Thesis

The overall aim of this research is to reduce the development time and effort needed to develop model transformations, and improve model transformation quality. In practice, diverse application scenarios arise for the synthesis of model transformations. These encompass situations wherein a developer's available resources consist solely of metamodels, exclusively models, or metamodels of varying sizes. In this thesis, we propose three approaches for automated or semi-automated model transformation synthesis. Each of these three approaches pursues the objective from different perspectives, playing to the strengths of each situation. To achieve the aim, all these approaches should:

- accelerate transformation production by automatically deriving the main structure of a transformation from metamodels, reducing the amount of transformation code that needs to be manually written.

- produce transformations in multiple languages without additional effort, which can address different properties of MT languages.

- ensure that transformations use correct language structures and appropriate

design patterns.

- ensure that transformations satisfy quality criteria, such as low cyclomatic complexity and low redundancy [13], [14], and avoid deprecated features, such as ATL iterative target patterns, which are deprecated since ATL 2.0 as they break internal traceability links. It is better to use unique lazy rules instead.

1.3 Research Questions

The overall research questions investigated in this thesis are:

RQ1: Can the F-measure of the DSS approach exceed 0.8, which is higher than existing approaches [15]–[17], in small and medium-sized metamodel matching?

RQ2: Can the search-based approach in this thesis outperform the performance of the state-of-the-art approach [18] in terms of effectiveness, while maintaining acceptable execution times on large metamodels?

RQ3: Can this method substantially reduce developer effort (with less than 10% of matchings requiring modifications and significantly faster execution times compared to manual development), while simultaneously enhancing the quality of generated model transformation (with a Flaw/LOC ratio lower than that of the manual development version)?

RQ4: Does the proposed machine learning framework require less data and less

time for transformation compared to the state-of-the-art approach [19]?

We answer the research questions by evaluating our approach with regard to its effectiveness in recognising appropriate metamodel matchings (with respect to the matchings implied by the original manually-developed versions of cases) and in producing transformations with appropriate correspondence. We also evaluated the generated different model transformations with quality measures.

For the evaluation, we use 16 benchmark cases [18], [20] of metamodel matching, 10 ETL cases from Epsilon [21], and 8 ATL cases selected from ATL Zoo [22]. The evaluation cases cover a range of transformation categories. We define the metamodels containing 10 or fewer classes as small-sized metamodels, metamodels containing between 10 and 25 classes as medium-sized metamodels, and metamodels containing 25 or more classes as large-sized metamodels.

The effectiveness of solutions is measured in terms of the closeness of the matchings to the original manual version of the transformations:

Precision = (number of correct class and feature mappings identified)/(total number of class and feature mappings identified)

Recall = (number of correct class and feature mappings identified)/(total number of class and feature mappings in original version)

F-measure = (2*Recall*Precision)/(Recall + Precision)

Correct means that our approach identifies the same class or feature matching as in the original version (mannual version). The higher the recall, the less effort is

needed to add missing matchings to correct our version. The higher the precision, the less work is needed to delete additional incorrect matchings from our version.

To address RQ1 and RQ2, as we expect the DSS and search-based optimisation approach can be used to extract correspondences from metamodels, evaluation needs to include the different size metamodels from established benchmark, and the effectiveness should be higher than existing works [18], [23].

To answer RQ3, we will evaluate our approach on a range of cases of different kinds of transformation (refinements, abstractions, evolutions, migrations and semantic mappings), which have previously-developed manual versions. As these manual versions are from published papers or official websites (such as ATL Zoo [22]), their quality and accuracy can be considered the higher standard among all manual versions.

To answer RQ4, we will use our MTBE approach to generate transformations and evaluate efficiency on different cases. We will also compare the performance of our machine learning model with the state-of-the-art approach [19].

1.4 Contributions

We first propose an approach that uses metamodel matching to produce transformations. We extract correspondences from metamodels by metamodel matching, and synthesise transformations from correspondences. Most work on metamodel matching has focused on model migration for metamodel evolution. In such cases

('homogeneous metamodels') there is usually a common vocabulary (names of classes and features) in the two metamodels, and relatively small structural differences between them, because one metamodel has been produced by incremental changes to the other metamodel. Our contribution is to define an exhaustive-search matching approach based on data-structure similarity (DSS) of classes, using a flattening process of classes to combine all of their owned, inherited and composed features. DSS takes into account the essential structure of the data associated with a class, regardless of class or feature names. DSS should be appropriate for cases where there are substantial differences between the structure of the metamodels, and between the terminology of classes and features in these metamodels ('heterogeneous metamodels').

However, exhaustive search for possible matches becomes infeasible even for cases of moderate size (around 25 to 30 metaclasses in the combined metamodels). To address this issue, we propose another metamodel matching approach based on search-based optimisation to obtain the optimal metamodel matchings by maximising three measures, Data structure similarity (DSS), Name syntactic similarity (NSS), and Name semantic similarity (NMS).

After extracting correspondences from metamodel matching, we introduce an intermediate language $\mathcal{TL}$ that provides a formal MT definition language independent of specific transformation languages. $\mathcal{TL}$ is able to generate multiple MT languages, including Query View Transformation Relations (QVTr), Query View Transforma-

tion Operations (QVTo), Atlas Transformation Language (ATL), Epsilon Transformation Language (ETL) and UML Reactive System Development Support (UML-RSDS).

Model transformation synthesis from metamodels can only detect direct feature value mapping, rather than detailed functional relationships between source and target data, such as applications of specific numeric or string functions. To solve this issue, we propose an approach based on Model Transformation by Examples (MTBE), which uses machine learning technique to learn model transformation examples and implement automated model transformation.

Our approaches are appropriate for refinements (mapping from a higher abstraction level model to a lower-level model), abstractions (the inverse of refinement) and semantic mappings (maps a model m in one language to a formal representation in a language with a formal semantics, to support semantic analysis of m), in addition to migrations (mapping models from one metamodel to another at the same level of abstraction) between evolved or non-evolved metamodels. We restrict our scope to out-place transformations, instead of in-place transformations which operate to restructure a single model (such as refactorings). We also do not consider model-to-text transformations such as code generators [2], [24].

To conclude, the main contributions of this thesis are as follows:

- Data structure similarity approach for metamodel matching (Chapter 4).
- Search-based optimisation approach for metamodel matching (Chapter 5).

- Defining an intermediate language $\mathcal{TL}$ to enable the production of specifications in multiple MT languages from correspondences (Chapter 6).

- A framework based on machine learning for model transformation by examples (Chapter 7).

In response to $RQ1$, our DSS approach has demonstrated the capability to surpass an F-measure threshold of 0.8, an achievement that surpasses the performance benchmarks set by existing approaches [15]–[17]. This accomplishment has been realized within the domain of small and medium-sized metamodel matching.

Addressing $RQ2$, our novel search-based approach has exhibited superior effectiveness when compared to the state-of-the-art approach presented by Kessentini et al. [18]. Importantly, this enhanced effectiveness has been maintained while ensuring that execution times remain at acceptable levels even in the context of large metamodels.

In the context of $RQ3$, our approach has offered a substantial reduction in developer effort, as fewer than 10% of matchings have required modifications. This efficiency gain has been accompanied by significantly faster execution times in comparison to manual development. Furthermore, the quality of generated model transformations, as measured by the Flaw/LOC ratio, has shown improvement, outperforming the manual development version.

Lastly, addressing $RQ4$, our proposed machine learning framework has demonstrated efficiency by requiring less data and less time for transformation in compar-

ison to the state-of-the-art approach proposed by Burgueño et al. [19].

In summary, our contributions encompass advancements not only in metamodel matching but also in the quality and efficiency of model transformation generation. These achievements demonstrate the efficacy and innovation of our approach in addressing pivotal research questions and surpassing the state-of-the-art within the field.

1.5 Structure

In Chapter 2 we present background on MDE and model transformations.

Chapter 3 reviews existing work in metamodel matching and model transformation synthesis.

Chapter 4 presents our DSS approach for metamodel matching. This addresses RQ1.

Chapter 5 shows how to extract correspondences from metamodels using search-based optimisation approach. This addresses RQ2.

Chapter 6 introduces $\mathcal{TL}$ specifications and how to generate different model transformation languages from correspondences. This addresses RQ3.

Chapter 7 describes the MTBE framework for model transformation. This answers RQ4.

Chapter 8 concludes the thesis and gives future work plans.

1.6 Publications

The following publications have resulted from the work presented in this thesis:

- S. Fang and K. Lano, "Extracting Correspondences from Metamodels Using Metamodel Matching," in *STAF (Co-Located Events)*, pp. 3-8, 2019.

  This paper corresponds to Chapter 4 in this thesis.

  Contributor roles: Conceptualization, Methodology, Validation, Writing – original draft, Writing – review and editing.

- S. Fang and K. Lano, "Search-based metamodel matching: an approach combining multi-objective optimisation and machine learning," submitted to *Information Systems Frontiers*, 2022.

  This paper corresponds to Chapter 5 in this thesis.

  Contributor roles: Conceptualization, Methodology, Validation, Writing – original draft, Writing – review and editing.

- S. Fang and K. Lano, "Model transformation by examples using a machine learning framework," submitted to *Applied Intelligence*, 2022.

  This paper corresponds to Chapter 7 in this thesis.

  Contributor roles: Conceptualization, Methodology, Validation, Writing – original draft, Writing – review and editing.

- K. Lano, S. Fang and S. Kolahdouz-Rahimi, "TL an abstract specification language for bidirectional transformations," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pp. 1-10, 2020.

  This paper corresponds to Chapter 6 in this thesis.

  Contributor roles: Conceptualization, Methodology, Validation, Writing – review and editing.

- K. Lano and S. Fang, "Automated Synthesis of ATL Transformations from Metamodel Correspondences," in *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*, pp. 263-270, 2020.

  This paper corresponds to Chapter 6 in this thesis.

  Contributor roles: Conceptualization, Methodology, Validation, Writing – review and editing.

- K. Lano, S. Kolahdouz-Rahimi and S. Fang, "Model transformation development using automated requirements analysis, metamodel matching, and transformation by example," in *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol 31, no. 2, pp. 1-77, 2022.

  This paper consists some content of Chapter 4 in this thesis.

Contributor roles: Conceptualization, Methodology, Validation, Writing – review and editing.

Chapter 2

Background

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [1] is a discipline in software engineering [25] that aims to raise the level of abstraction of program specification and increasing automation of program development [26], [27]. This can improve the maintainability, interoperability, portability, and productivity of software systems [28]. Compared to pure software system development activities, it focuses on the engineering processes [29]. To make the benefits of MDE in the development of software systems clearer, the nature and fundamental elements of MDE should be introduced first.

There are different fundamental elements in MDE. MDE relies on models as first class entities and that aims to develop, maintain and evolve software by performing model transformations. Model-Driven Development (MDD) [26], [30]–[32] is the

most common instance of MDE. It is a software engineering framework that aims to simplify and formalise the different activities and tasks that comprise the software life cycle by using models and model technologies to raise the level of abstraction at which developers create and evolve software [33]. Model-Driven Architecture (MDA) [34], [35] is a subset of MDD proposed by the Object Management Group (OMG) [36]. The MDA paradigm uses OMG standards for the purpose of development [29].

2.1.1 Model in MDE

In different research areas, models have different definitions [28], [35], [37]–[40]. To conclude, a model uses a language to define a system in an abstract level without the details of the system, enabling a more focused analysis of its structural and behavioral aspects. In the MDE paradigm, software systems are specified and maintained as models, which represent the main elements of the systems.

Models can be expressed by different modelling languages, such as Unified Modelling Language (UML) [41], B [42], Z [43], etc. Modelling languages can be divided into two categories, Domain-Specific Languages (DSLs) [44], [45] and General-Purpose Modelling Languages (GPMLs) [46]. DSLs are tailored to particular application domains, while GPMLs can be applied to any domain [45].

UML [47], [48] is a standardised modelling language consisting of an integrated set of diagrams to build software models effectively. It has been adopted as the standard by OMG since 1997. From then on, UML has been used to help software

developers clarify, demonstrate, build and document the output of software systems, which makes it become a very important part of software system modelling [49].

Object Constraint Language (OCL) [47], [48] is a declarative language describing constraints applying to UML models [50]. OCL includes two main parts: invariants that have to be respected by software elements; specification of operations on the software elements through pre and post- conditions [51]. A pre-condition defines the conditions that must be met before an operation is executed, ensuring it's invoked in the right system state. Post-conditions must be true after an operation's execution, verifying if the behavior aligns with expectations. Together, they provide clear specifications and verification for the execution and behavior of operations.

2.1.2 Metamodel in MDE

A metamodel defines the structure of the models and is specified with sets of syntactic and semantic constraints in a modelling language [52]. A metamodel is the higher abstraction of a model, therefore the hierarchical process is applicable to them. Having a consistent and precise metamodel is a prerequisite for representation of a model [2]. In addition, a valid model must conform to its metamodel and satisfy the constraints of the metamodel [53].

The Meta Object Facility (MOF) [54] is a metamodelling framework and metadata repository standard proposed by OMG. In order to generate metamodels, the MOF framework was described by UML. There are four layers in the framework to

represent the abstraction of the model, and each layer is an instance of the layer above it [55]. XMI [56] is a standard proposed by OMG, which is used to save UML models in XML format [57]. Therefore, XMI is used to exchange data between different layers by the MOF framework [58]. The advent of MOF not only provides consistency between modelling, but also makes it possible to interact between different tools of MDE.

2.1.3 Model Transformation in MDE

Since the model is the most important element in MDE, the model transformation plays a key role in MDE [24]. In MDE paradigm, to reach different aims, models are often transformed from one representation to another one. The source models should conform to the source metamodels, and the target model should conform to the target metamodels [59].

The model transformation specifications are a set of rules defining how to transform the source model to the target model. Different transformation specifications are used to deal with different model transformations. Some transformation approaches have been proposed and some tools have been developed on the basis of these approaches. These approaches and tools are aimed at using transformation specifications to generate a target model from a source model.

2.1.4 Model-Driven Development

As an instance of MDE, MDD focuses on software development. Specifically, the defining characteristic of MDD is concentrating on models, not on software programming. Compared to programming languages, models bring fewer technical constraints and a more intuitive representation of the problem to be solved. This also makes it possible to allow software developers to complete software development with only a modelling language, without the need to be familiar with specific programming languages [26], [32].

However, two important prerequisites for the widespread use of MDD are that mature automation transformation technologies and industry-wide standards have emerged [26].

2.1.5 Model-Driven Architecture

Model-Driven Architecture (MDA) [28], [60], [61] is a framework proposed by OMG. The framework meets several OMG standards and is used to support software development in MDE. Specifically, the aim of MDA is to decrease the amount of implementation detail required in system development by utilizing patterns of implementation and raise the level of abstraction and integration at which system developers work [62].

MDA includes three levels of abstraction for models, which are Computational Independent Model (CIM), Platform Independent Model (PIM) and Platform Spe-

cific Model PSM).

CIM is an informational view point model which represents system and software knowledge from the business perspective, rather than the details of the software system. It may contain business knowledge about system organisation, roles, functions, processes and activities, documentation, constraints, etc. [63]. Therefore, a CIM does not require a specific modelling language [64].

A PIM is a model that focuses on the structure and function of a system, rather than the details of technologies. To be more specific, the PIM takes into account the parts of the software system that need to be developed, but does not determine the technological platform that will be supported by the implementation [65].

A PSM is a model that is related to a specific technical platform, such as a specific programming language or operating system. It is essential for the implementation of system development [66].

Model transformation plays an important role in MDA. After building a CIM from the problem domain, model transformation is applied to generate a PIM from the CIM. Similarly, a PSM is generated from a PIM by model transformation as well. Finally, the specific technical details (such as code) are generated by PSM.

2.2 Model Transformation Taxonomy

Model transformation can be distinguished between *Model-to-Model (M2M)* and *Model-to-Text (M2T)* transformations [29]. In the Model-to-Model transformation,

both input and output parameters are models. While in the Model-to-Text transformation, the input parameter is a model and the output parameter is a text string. Similarly, Text-to-Model (T2M) transformations have a text string as input and a model as output, and such transformations are typically applied in reverse engineering [67].

Based on the language in which the source and target models of a transformation are expressed, a distinction can be made between *endogenous* and *exogenous* transformations. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages [39].

A further taxonomy is *in-place* versus *out-place (separate-model)* transformations, which is based on the number of models involved. If the number is only one, the source and target model are the same, and all changes are made in-place. If the transformations create model elements in one model based on properties of another model, such transformations are called out-place. Note that exogenous transformations are always out-place [2].

In addition, a *horizontal transformation* is a transformation where the source and target models reside at the same abstraction level. A *vertical transformation* is a transformation where the source and target models reside at different abstraction levels [68], [69].

2.3 Model Transformation Languages

Model transformations are supported by different dedicated transformation languages [70], [71]. Model transformation languages are domain-specific languages, have ever since been associated with advantages in areas like productivity, expressiveness and comprehensibility [71]–[73]. In our research, we implement the transformation derivation for several transformation languages. This section introduces several common model transformation languages.

2.3.1 Query View Transformation Language

Query View Transformation Relations (QVTr) [74] is a standard transformation language developed by OMG for supporting complex object pattern matching, and implicitly creating trace classes and their instances to record what occurred during a transformation execution. Query View Transformation Operations (QVTo) [74] is a standard transformation language that can be regarded as a way of providing imperative implementations, which is populated with the same trace models as the QVTr. OCL extensions are used in QVTr, making it very user-friendly. In the situation that there is challenge on providing a purely declarative specification of how a Relation is to be populated, more than one Relation from a Relation specification can be implemented by Mappings Operations. To create a trace between model elements, the intercall between Mappings Operations will always involve Relation. A transformation entirely written using Mapping Operations is called an operational

transformation [74].

2.3.2 UML Reactive System Development Support

UML Reactive System Development Support (UML-RSDS) [75] is based on UML. For different levels of abstraction, necessary specification notations are provided for MT definition. Precise axiomatic semantics are also provided which allows UML-RSDS to generate executable code.

UML-RSDS can be used to define model transformations in two alternative ways: (i) declaratively and abstractly using constraints, which express implicitly how two (or more) models are related, and what changes to one model need to be made (to preserve the truth of the constraints) when a change in another model takes place, or (ii) by using operations of metamodel classes to explicitly define how a target model is produced from a source model [75].

2.3.3 Atlas Transformation Language

Atlas Transformation Language (ATL) [76], [77] is a hybrid transformation language. Both declarative and imperative constructs can be found in ATL. Declarative style is encouraged. ATL transformations are unidirectional. It transforms read-only source models to write-only target models. A bidirectional transformation can be implemented by dividing it into several transformations. The source model can be navigated when a transformation is executed, however, due to the source model

being read-only, it cannot be changed. In contrast, navigating is not allowed for a target model.

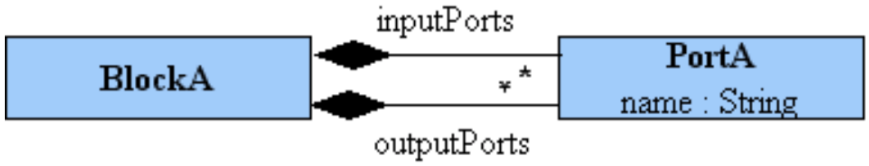
ATL has two particular features: *rule* and *helper*.

The ATL *matched rule* mechanism provides a convenient way to specify how to generate source model elements to target model elements for model transformation developers. For this purpose, a matched rule enables to specify 1) which source model element must be matched, 2) the number and the type of the generated target model elements, and 3) the way these target model elements must be initialised from the matched source elements. Matched rules are declarative rules that are not explicitly called anywhere in the transformation, but are matched by the ATL engine. However, the elements created in some matched rules may reference some elements created by other matched rules. *Lazy rules* and *Unique lazy rules* are used for creating new elements in the target model. They are called from matched rules or from another unique lazy rule or lazy rule. The difference between lazy rules and unique lazy rules is that the former always create new elements when they are called, independently from the parameters with which they are called. The behaviour of unique lazy rules is different. They do create new elements when they are first called with any parameter. However, if a unique lazy rule is ever called again with a specific parameter used before to call the same unique lazy rule, it does not create again the same elements, but it returns a pointer to the elements previously created. ATL also defines an additional kind of rules, *called rule*, which is used in situation

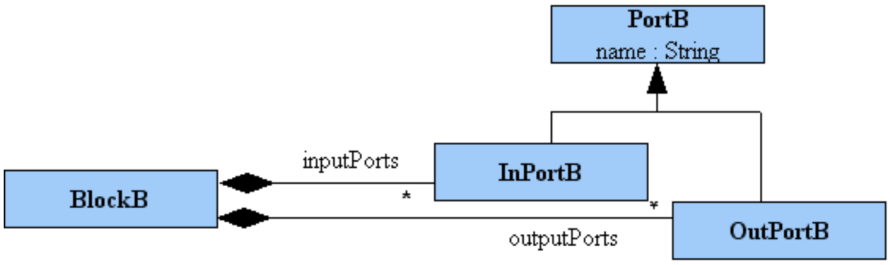
that requiring to generate target model elements explicitly by using imperative code [78].

ATL enables developers to define methods to respond to different requirements. These methods are called *helpers*. The *helpers* make it possible to define factorised ATL code which can then be called from different points of an ATL program. There are two kinds of helpers: the *functional helper* and the *attribute helper*. A given data type is required for defining both functional helper and attribute helper. However, a functional helper can accept parameters, but an attribute helper cannot accept parameters [78].

Here is an example of ATL, which is from the Port case of ATL Zoo [22]:



(a) Type A



(b) Type B

Figure 2.1: Port metamodels

```
1 module TypeA2TypeB;
2 create inB : TypeB from inA : TypeA;
```

```
3
4  rule BlkA2BlkB {
5          from
6                  blkA : TypeA!BlockA
7          to
8                  blkB : TypeB!BlockB (
9                          inputPorts <- blkA.inputPorts->
                              collect(e | thisModule.
                              PortA2InPortB(e)),
10                         outputPorts <- blkA.outputPorts->
                              collect(e | thisModule.
                              PortA2OutPortB(e))
11                 )
12 }
13
14 lazy rule PortA2InPortB {
15         from
16                 s : TypeA!PortA
17         to
18                 t : TypeB!InPortB (
19                         name <- s.name
20                 )
```

```
21 }
22
23 lazy rule PortA2OutPortB {
24         from
25                 s : TypeA!PortA
26         to
27                 t : TypeB!OutPortB (
28                         name <- s.name
29                 )
30 }
```

2.3.4 Epsilon Transformation Language

The Epsilon Transformation Language (ETL), a hybrid model transformation language, was proposed by Epsilon [79]. It has been built on top of the infrastructure provided by the Epsilon Eclipse GMT component. ETL provides a task-specific rule execution scheme, but also allows for the imperative features within EOL [80] to be used for transformation rules.

An ETL file consists of a module which contains transformation rules, each rule must have a unique name (with respect to the module) and operates on one source element that can be transformed into many target elements. A transformation rule is defined as one of three keyword rules: abstract, lazy and primary. A rule can

also extend another, using the extends attribute. Transformation rules may also contain guards, which are defined in EOL, that restrict the rules applicability to the source model elements. The structure of each transformation rule requires a keyword to aid in its definition, as well as a transform keyword which must list the source keyword and the to keyword that declares one or more targets. The extends keyword can then be used if necessary to define a comma-separated list of rules to which the current rule extends, enabling further detail to be given to pre-defined rules, or abstract rules. (This is similar to the extends functionality in Java). If required, a guard can then be used which defines an EOL block, and finally the body of the rule is specified as a sequence of EOL statements. ETL also offers the ability to define operations, which are imperative blocks of EOL that can be written to aid in the model transformation process. These operations can take multiple inputs as well as provide return values, furthermore they are able to create new instances of model elements. This provides the freedom to choose between imperative operations and the more declarative lazy rules when defining a transformation module. As previously mentioned EOL is the basis for the Epsilon suite, and as such different aspects of the Epsilon suite can build upon this. A key addition to EOL within ETL, which enhances its transformational powers, is the *equivalent()* operator. The *equivalent()* or *equivalents()* operators automatically resolve the source elements to their transformed counterparts in the target model; *equivalent()* returning a single element and *equivalents()* returning a bag [80].

Here is an example of ETL, which is from the Epsilon [79]:

```
rule Tree2Node
    transform t : Tree!Tree
    to n : Graph!Node {
    n.label := t.label;

    if (t.parent.isDefined()) {
    var edge := new Graph!Edge; edge.source := n;
    edge.target := t.parent.equivalent();}
}
```

2.4 Model Transformation Categories

To precisely define the scope of the approaches elucidated in this thesis, specifically, the range of different model transformation categories they should encompass, we classify the category of a transformation based on the transformation intent definitions of Lucio et al. [81]:

1. **Migration** – mapping models from one metamodel to another at the same level of abstraction. This can be subdivided into:

   (a) *Copy* transformations where the source and target metamodels are the same or isomorphic

(b) *Evolution* cases where one metamodel is an evolved version of the other

(c) *Heterogeneous migration*, where the metamodels are unrelated by evolution.

2. **Analysis** – extracting information from a model as a view or other analysis result.

3. **Refinement** – mapping from a higher abstraction level model to a lower-level model.

4. **Abstraction** – the inverse of refinement.

5. **Code generation** – mapping from a model to text or executable code.

6. **Refactoring** – Update-in-place transformations that restructure a model, retaining its conformance to the same metamodel or a closely related metamodel.

7. **Semantic mapping** – maps a model m in one language to a formal representation in a language with a formal semantics, to support semantic analysis of m.

8. **Bidirectional (Bx)** – transformations which can be applied in either source-to-target or target-to-source directions, supporting model synchronisation and change-propagation, which ensure the source and target models are kept consistent with each other by the bx.

In general, we would expect copy transformations to be simpler to synthesise than evolution cases, evolution to be simpler than heterogeneous migration cases, etc.

For refinements and semantic mappings, where the target metamodel is more detailed than the source, we could look for feature matchings of direct source features to composite target features: $f \longmapsto g.h$. Abstraction would work in the other way: $g.h \longmapsto f$.

Evolution cases are more likely to have direct-to-direct matchings $f \longmapsto g$.

But general migrations could involve both refinement and abstraction aspects.

2.5 Scope of the Research

In terms of scope, we not only consider the metamodel matching between one source class and one target class, but also consider many-to-one, one-to-many, and many-to-many matching.

We consider both model-to-model (M2M) transformations between two homogeneous metamodels (such as migration transformations to support metamodel evolution, where one metamodel is an evolved version of the other), and general M2M transformations relating two heterogeneous metamodels of different languages. In this thesis, we focus on the separate-models case, including refinements, abstractions, semantic mappings and migrations. We do not cover model-to-text (M2T), text-to-model (T2M), or text-to-text (T2T) cases.

Chapter 3

Related Work

In this chapter, we survey related work in metamodel matching and transformation synthesis. We summarise the gaps and limitations in existing approaches. Additionally, we illustrate how our approaches address these shortcomings through a comparative analysis, thereby highlighting the novelty of our approach.

3.1 Metamodel Matching Approaches

In this section, we classify some metamodel matching approaches in the literature by the matching algorithm on which it is based. Some of these approaches reuse algorithms that have already been applied for database schema matching, such as similarity flooding, or search-based algorithms. While others are based on customised rules that are specifically developed for metamodels.

3.1.1 Matching Approaches based on Similarity Flooding

An approach based on similarity flooding (SF) [82] is proposed by Falleri et al. [16], which automatically detects mappings between two metamodels and uses them to generate an alignment between those metamodels. However, this approach requires manual specification of the initial 'seed', and the quality of the 'seed' directly impacts the final matching results. In addition, this alignment needs to be manually checked and can then be used to generate a model transformation manually, which our approaches check and generate automatically.

Atlas Model Weaver (AMW) [10] is an approach that uses matching transformations and weaving models to semi-automate the development of transformations. Weaving models are models that contain different kinds of relationships between model elements. These relationships capture different transformation patterns. Matching transformations are a special kind of transformations that implement methods that create weaving models. The first phase of matching transformation is to create weaving models. The second phase is calculating element similarity, which computes a similarity value between the elements referred by the source and target references, for every link of a weaving model. This similarity value is used to evaluate the semantic proximity between the linked elements. A link with a high similarity value indicates that there is a good probability that the source element must be translated into the target element. The third phase is selecting best links, which selects only the links that satisfy a set of conditions. The selected links are included

in the final weaving model. In the fourth phase, a graphical approach is used to easily chain and customise different matching transformations. Finally, the weaving model produced by the matching transformations is translated into a transformation model. However, AMW [10] can only handle particular source and target metamodels, however we define general-purpose patterns and consistency and completeness checks to identify and refine correspondences for arbitrary metamodel pairs. For example, the mutual consistency of feature mappings of the two directions of a bidirectional association is a logical property which must hold for any semantically-valid transformation, and hence a proposed matching *cm* (class matching), *fm* (feature matching) must satisfy this property or be modified to satisfy it

3.1.2 Matching Approaches based on Customised Rules

COPE [12] is an integrated approach to specify the coupled evolution of metamodels and models to reduce migration effort. COPE is based on a language that provides means to combine metamodel adaptation and model migration into so-called coupled transactions. Two central requirements for adequate tool support, reuse and expressiveness, are fulfilled by two kinds of coupled transactions: reusable and custom coupled transactions. A reusable coupled transaction allows the reuse of recurring coupled transformations across metamodels. A custom coupled transaction can be manually defined by the metamodel developer for complex migrations that are specific to a metamodel. Reuse is provided by reusable coupled transac-

tions that encapsulate recurring migration knowledge. Expressiveness is provided by a complete set of primitives embedded into a Turing-complete language, which can be used to specify custom coupled transactions. Our concept of DSS metamodel matching is related to the concept of metamodel-independent coupled changes in the COPE system [12], however metamodel matchings are defined as logical relations between metamodel structures, whilst COPE coupled changes are operationally defined. Adopting a more abstract and general formalism for expressing metamodel relationships gives us the capability to automatically produce transformations in multiple languages, whilst the COPE transformations are manually coded in a specific language. We consider arbitrary pairs of metamodels, whilst COPE is focused on the case of metamodel evolution.

AtlanMod Matching Language (AML) [9] provides a notation for the construction of metamodel matching using heuristics. The process of matching is divided into three steps. The first step calculates the equivalence and changes between a metamodel and its previous version metamodel. The second step transforms the equivalence and changes into an adaptation transformation. In the third step, this adaptation transformation can be adapted to the new version of any model that conforms to the previous version. However, the AML [9] requires more substantial intervention from the developer than our approach, which is primarily automated and requires relatively low manual intervention.

A metamodel matching approach based on the planar graph edit distance (PGED)

is proposed [83], which requires a planarization algorithm for metamodel graphs. This approach proposes to apply an efficient approximate graph edit distance algorithm and presents the necessary adjustments and extensions of the general algorithm as well as an optimisation with ranked partial seed mappings. However, they rely on an initial manually-constructed 'seed' matching of classes, and on planarization of metamodel graphs, which our approaches do not need.

A class matching algorithm based on ontology is introduced [11], which is not limited to a certain ontology language or to a certain domain of interest, rather any kind of ontologies can be used as long as they classify their concepts in a taxonomy. In particular, the algorithm can access common knowledge ontologies like the linguistic resource WordNet [84], the DB-pedia ontology [85], and further Web Ontology Language (OWL) ontologies. From the obtained class model mappings, a QVT script is generated automatically, whose particular relations reflect identified correspondences between classes of any cardinality. Another approach based on semantic matching with EMFCompare [86] is proposed [23], which presents a custom matching engine extending the default one of EMFCompare. In addition to the syntactical and structural correlations, the proposed extension compares model elements with respect to their semantic meaning using the WordNet lexical database [84].

The ontology approach [11] and the EMFCompare approach [23] use ontology-based matching for class and attribute names. This may however require access to

substantial background knowledge and extension of the metamodels. In addition, the EMFCompare approach [23] is probably only effective when considering different versions of the same models, and may tend to create inefficiencies whenever applied to models conforming to different metamodels, while our approaches faces both situations. Ontology approach [11] may require manual intervention and improvement, and there are limitations on the forms of association that can be handled. Quality measures of the generated QVTr are not assessed.

3.1.3 Matching Approaches based on Search-based Algorithms

GAMMA [18] is an approach that considers metamodel matching as an optimisation problem. The approach uses a global search, namely genetic algorithm, to generate an initial solution and, subsequently, a local search, namely simulated annealing, to refine the initial solution generated by the genetic algorithm. The approach starts by generating randomly a set of possible matching solutions between source and target metamodels. Then, these solutions are evaluated using a fitness function based on structural and syntactic measures. However, they do not consider DSS or composed features in their matchings, but only non-composed (directly owned) and inherited features. In addition, a general limitation of genetic algorithm approaches is that they are probabilistic, which means that they could produce different results in different executions. Also, it usually requires some initial population to be defined.

Transformation synthesis is not addressed in GAMMA [18].

3.1.4 Schema Matching

Schema matching is a fundamental challenge in various domains of database applications, including data integration, schema evolution, schema migration, and so on [87]. Despite its primary application in the database field, this problem also involves establishing associations between different abstraction levels. Additionally, schema matching encounters the need for automation due to the time and effort involved in manual matching, paralleling the requirements found in metamodel matching [87]. Several approaches to automate schema matching have already been developed. SKAT [88], [89] and TranScm [90] employ implicit rules for schema matching, while DIKE [91], [92] uses algorithms to calculate synonyms, homonyms to get similarity for matching. ARTEMIS [93] relies on a computer software tool [94] for schema matching. CUPID [95] focuses on tree-like data schemas (XML) and performs matching based on similarity, utilizing techniques like matching subtrees and weighted leaves. These approaches primarily rely on name similarity criteria such as name equality, synonyms, homonyms, and hypernyms. SemInt [96], [97] employs pattern schema through neural network training.

However, a common limitation among these approaches is the need for manual intervention. SKAT [88], [89] and TranScm [90] rquire developer to add matching and mismatching rules. DIKE [91], [92] necessitates manual resolution of structural

conflicts, and ARTEMIS [93] and CUPID [95] allows users to adjust weights or threshold values, requiring user intervention. SemInt [96], [97] requires developers to select matching attributes from attribute clusters. In contrast, our approaches in this thesis minimizes the need for manual intervention and supports n-m matching, distinguishing it from these existing approaches.

3.2 Transformation Synthesis Approaches

There are several approaches for transformations synthesis from correspondences [9], [10].

AMW [10] defines case-specific patterns to create transformations for particular source and target metamodels, and its ATL generation approach does not appear to address the issue of composed target features, which considerably complicates ATL production. Compared to our approaches in this thesis, AMW does not consider the quality of generated transformations, and its ATL generation approach does not appear to address the issue of composed target features, which considerably complicates ATL production.

The AML [9] provides a means to define customised matching criteria and techniques. This requires more substantial intervention from the developer. It is unclear if AML is able to generate complex ATL for cases of composed target features, while our approaches can hanldel this.

Model transformation by example (MTBE) is an approach initially introduced

by Varro et al. [98]. An automated approach [99] is proposed by using inductive logic programming (ILP) [100]. An initial tool support for MTBE using ILP [101] is presented to infer explicit transformation rules from example models. The approach was only demonstrated on small examples. An approach for programming by examples (PBE) [102], [103] is technically related to MTBE, but focusing on programming languages.

Other approaches to MT synthesis are MTBE using machine learning [19], [104] or search-based model matching using optimisation algorithms [105]. While these approaches achieve complete automation, large numbers of examples are needed, and the transformation is derived as a 'black box'. Existing ML approaches [19], [104] cannot recognise numeric relationships.

Compared with the existing MTBE approaches [19], [101], [104], [105], our MTBE approach can implement several common feature value transformations. In addition, due to the decision tree technique, our approach has some interpretability.

3.3 Conclusion

Table 3.1 summarises the different approaches that have been taken for metamodel matching and transformation synthesis. Table 3.2 summarises different MTBE approaches.

To conclude, existing approaches mainly consider name syntactic similarity or name semantic similarity, or graph structure similarity, rather than data-structure

Table 3.1: Comparison of metamodel matching and transformation synthesis approaches

Approach	Scope	Matching measures	Matching algorithms	Consistency/ completeness	Synthesised transformations	Transformation quality
SF [16]	general	graph structure	similarity flooding	✗	none	✗
AML [9]	evolution	name syntactic, name semantic, customised	customised	✗	ATL	✗
AMW [10]	evolution, migration	name syntactic, name semantic	similarity flooding	✗	ATL	✗
COPE [12]	evolution, migration	name syntactic	customised; manual	✗	COPE	✗
PGED [83]	general	graph structure	customised	✗	none	✗
SBSE [18]	general	name syntactic, name semantic, structure	search-based	✗	none	✗
Ontology [11]	general	name semantic, ontology	customised	✗	QVTr	✗
EMF Compare [23]	general	name semantic, customised	similarity flooding	✗	none	✗

Table 3.2: Comparison of MTBE approaches

Approach	Scope	MTBE algorithms	Complex feature value transformation	Interpretability
ILP [101]	general	ILP	No	Yes
Search-based [105]	general	Genetic	No	No
LSTM [19], [104]	general	Neural networks	No	No

similarity. Name complexity, potential semantic ambiguity, and differences in naming conventions can hinder similarity assessment. Graph structures are often intricate, involving nested elements, multi-level relationships, and possible inconsistencies. These makes it more difficult to derive transformations from the correspondences [8]. Genetic algorithms have the problem of probabilistic behaviour, which is due to several stochastic components inherent in their operation, such as the initial population is randomly generated, and the processes of crossover and muta-

tion introduce randomness in the generation of new individuals. These stochastic operations collectively contribute to the probabilistic nature of genetic algorithms, and may fail to achieve an optimal mapping [106]. Deterministic approaches have problems of scalability.

Some approaches do generate transformations, however, the generated transformation is only in one language (ATL [9]; QVTr [11]; COPE transformation [12]).

The existing MTBE approach cannot handle complex feature value transformations, such as numerical or String transformation.

There are gaps in current work for approaches which generate transformations in multiple languages, also for approaches which effectively combine data-structure and name similarity techniques. There is a lack of MTBE approaches focused on complex feature value transformation.

Chapter 4

Data-structure Similarity Approach for Metamodel Matching

4.1 Introduction

A correspondence of two metamodels defines a class matching between classes of the metamodels, and a feature matching between features in the source classes and target classes. These correspondences can be used to define transformations directly. Figure 4.1 shows the connection between these elements.

In cases where it is difficult to perform model transformation directly at the model level, it is a simpler choice to operate from the upper level, that is, between

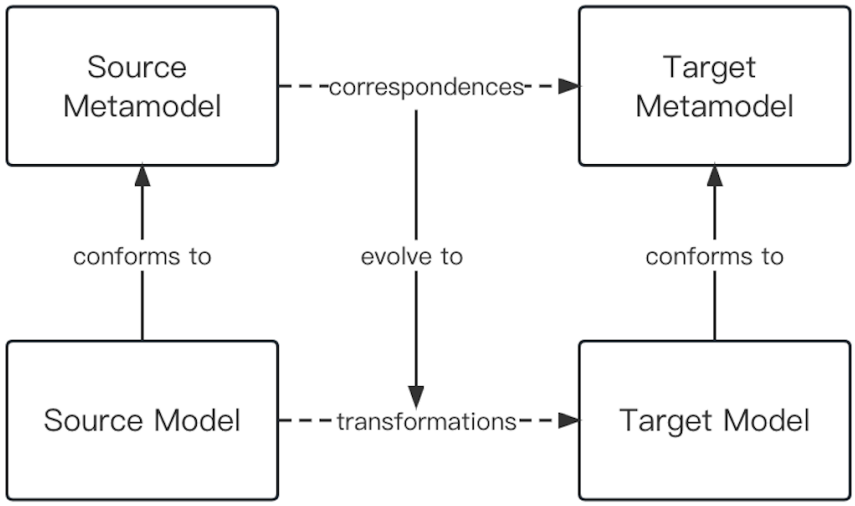


Figure 4.1: Connection between MDE elements

metamodels. In this chapter, we introduce the concept that correspondences can be extracted from metamodels by using metamodel matching.

We propose a Data-structure similarity (DSS) approach for metamodel matching. To better demonstrate our approach, we introduce a pair of metamodels, Tree metamodel (Figure 4.2) and Graph metamodel (Figure 4.3) as an illustrative example. We will use this illustrative example to show in detail how to use DSS approach for metamodel matching in the following sections. The modeling language used for presenting model diagrams in this thesis is UML.

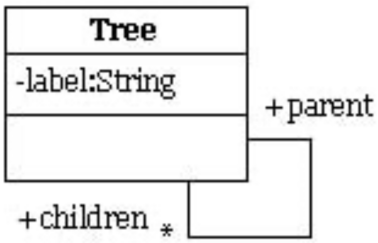


Figure 4.2: Tree metamodel

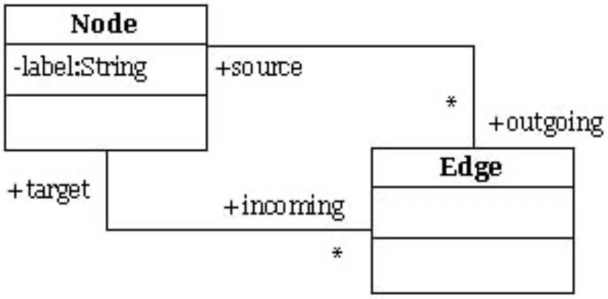


Figure 4.3: Graph metamodel

4.2 Flattening Metamodel

To make the class information completely be expressed, we need to flatten all classes in the source and target metamodels first. Flattening a class is the process of representing the class in a form which represents all of its recursively inherited and composed features [107]. The process of flattening inheritance is as follows: for each class C, if C has a superclass D, copy all features of D to C if they are not already in C. To flatten navigation, the process is: for each class C, if it has an association r to another class D, add features $r.a$ (for each feature $a : T$ of D) to

C. Specifically, due to the existence of multiplicity, there may be three cases of a type T: T, $Set(T)$, and $Sequence(T)$, Table 4.1 summarises the types of $r.a$ under different conditions.

Table 4.1: Feature types of $r.a$ for $a : T$

Feature type	Condition
T	if r has 1-multiplicity
$Set(T)$	if r has *-multiplicity and T is not a collection type
$Sequence(T)$	if r has * ordered-multiplicity and T is not a collection type
$Set(S)$	if r has *-multiplicity and T is a collection type $Set(S)$ or $Sequence(S)$
$Sequence(S)$	if r has * ordered-multiplicity and T is $Sequence(S)$

The associations are also represented as features in the flattened representation, and the type of these features is the target class to which the association refers. This ensures that the structure of the metamodel is fully expressed in the flat version. The flattening process terminates when the flattened association of a class redirects to the same class. For example, class 'Edge' has an association to 'Node', and there is also an association 'outgoing' from 'Node' to 'Edge', so 'Edge' has an association 'target.outgoing' to itself, as the multiplicity of 'outgoing' is many (*), therefore, the type of 'target.outgoing' is 'Set(Edge)'.

These steps are repeated until there is no change in the metamodel. The flattened metamodels of the illustrative case are shown in Figure 4.4 and Figure 4.5.

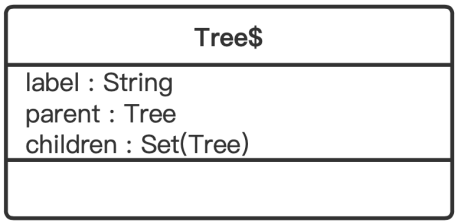


Figure 4.4: Flattened tree metamodel

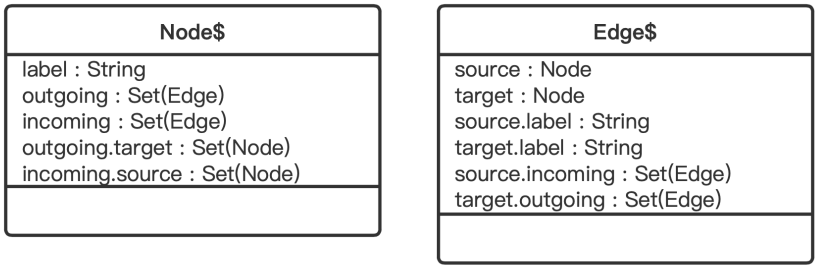


Figure 4.5: Flattened graph metamodel

4.3 Data Structure Similarity Measure

Data structure similarity (DSS) approach only measures similarity of the types of the class features. For a class in a metamodel, in addition to its directly owned features, its other features are derived from inheritance and navigation.

For two features with the same type, their DSS should be 1, otherwise it is 0. The DSS for 2 classes should be 1 if neither class has any features. For the types of associations, if we assume that class $E1$ matches class $E2$, then type $E1$ is considered fully similar (value 1 equivalent) to type $E2$.

An interesting question is whether features with different multiplicity should

be regarded as the same type. We give two alternatives: exact matching and fuzzy matching. In the exact matching, the similarity between these different types should be 0 (Table 4.2). However, for the fuzzy matching, we performed a regression analysis on a number of metamodel matching cases to obtain the most reasonable type similarity with different multiplicity (Table 4.3) [108]. *Sequence(T)* could be considered 0.8 similar to *Set(T)*, because the data structures are quite similar (they both allow for multiple elements, and differ only in terms of ordering). T is only partly similar (0.5) to *Set(T)* or *Sequence(T)* because the type of elements is the same, but the multiplicities are different. This allows more flexible matches than strict equality of the types.

Exact type matching is appropriate for homogeneous metamodel matching cases, e.g., evolution cases. If exact type matching is used for heterogeneous metamodel matching cases, then many matchings have 0 similarity, due to the different vocabulary (names of classes and features) in the two metamodels. Hence, fuzzy type matching is more appropriate for other cases, where more differences between the metamodels can be expected.

Table 4.2: Exact type matching similarity

Similarity	*a : T*	*a : Set(T)*	*a : Sequence(T)*
b : T	1	0	0
b : Set(T)	0	1	0
b : Sequence(T)	0	0	1

Table 4.3: Fuzzy type matching similarity

Similarity	*a : T*	*a : Set(T)*	*a : Sequence(T)*
b : T	1	0.5	0.5
b : Set(T)	0.5	1	0.8
b : Sequence(T)	0.5	0.8	1

4.4 Other Similarity Measures

In addition to DSS, we also investigated several different alternatives (Table 4.4), driven by our intention to assess and contrast diverse methods for measuring similarity in order to pinpoint the most appropriate similarity measure for our research.

Table 4.4: Similarity measures

Measure	Definition of similarity
Graph structural similarity (GSS)	Class neighbourhoods in MM_1, MM_2 have similar graph structure metrics [109]
Graph edit similarity (GES)	Class reachability graphs in MM_1, MM_2 have low graph edit distance [83]
Name syntactic similarity (NSS)	Names with low string edit distances [110]
Name semantic similarity (NMS)	Names are synonymous terms or in the same/linked term families according to a thesaurus [111]
Semantic context similarity (SCS)	Classes play similar semantic roles in the 2 metamodels [112].

4.4.1 Graph Structure Similarity

Graph similarity measures treat a metamodel as a graph of nodes (classes) and edges (associations, aggregations and inheritances). GSS involves three separate measures of graph structure: leadership (L), bonding (B) and diversity (D) [109]. Leadership measures how dominant one vertex v is in the graph, ie., by how much the number of v-incident edges d_v exceeds that of other verticies. From the viewpoint of one vertex e the L measure is defined as $\frac{n*d_e \ - \ \Sigma_{x \neq e} d_x}{(n-2)*(n-1)}$, where n is the number of verticies. Bonding measures the number of triples e_1, e_2, e_3 of distinct nodes connected in a triangle, as a proportion of the triples that only have two connecting edges. For a specific class e, we restrict the considered nodes to the immediate neighbours of e. Diversity measures the number of disjoint edges in the graph (edges with no common end points). The proportion of edges adjacent to e or a neighbour of e which are pairwise disjoint is taken as the D measure.

4.4.2 Graph Edit Similarity

GES is based on the concept of graph edit distance $d(g1, g2)$: the minimum number of basic graph rewrites needed to transform one graph $g1$ into another $g2$, modulo the matching cm of nodes (classes are the graph nodes in this analysis) [83]. We compute the subgraph R_e of nodes reachable from source class e in MM_1 and the subgraph $R_{e'}$ of nodes reachable from a potential matching class e' in MM_2 and evaluate the similarity of e and e' as $1/(1 + d(R_e, R_{e'}))$. The basic edit steps permitted are

splitting an edge with a new node, removing a node and joining incident edges, and introducing/removing edges.

4.4.3 Name Syntactic Similarity

Name syntactic similarity (NSS) measures the string edit distances [110] of the names of two features. String edit distance measures the minimum number of operations (insertions, deletions, substitutions) which change one string into another one. For example, the edit distance, ed, between the two lexical entries "TopHotel" and "Top_Hotel" equals 1, ed("TopHotel", "Top_Hotel")=1, because one insertion operation changes the string "TopHotel" into "Top_Hotel".

For two strings A and B, the NSS is:

$$NSS = \frac{A.size + B.size - ed(A, B)}{A.size + B.size}$$

4.4.4 Name Semantic Similarity

For NMS a suitable thesaurus (in XML format) is used. A thesaurus has data consisting of *concepts* each of which has a set of preferred and alternative *terms* [111]. A term may be in the term sets of several concepts. Terms have a semantic distance of 0 if they are identical or are both preferred terms of one concept, a distance of 1 if they are other alternatives for the same concept, and a distance of 2 if they are in the term sets of two concepts linked by a common term. Otherwise the distance is 3.

NMS similarity of terms $nmsterm(s1, s2)$ is then $\frac{3-distance}{3}$. We decompose class and feature names using camel-case splitting [113] and common prefix/suffix removal, and apply NMS to the resulting substrings if they have length ≥ 2. $nms(c, d)$ for classes c, d is $\frac{1}{K*L}*$ the sum of the NMS similarities $nmsterm(cc, dd)$ of the substrings cc of $c.name$ and dd of $d.name$, where K, L are the number of $c.name$, $d.name$ substrings.

4.4.5 Semantic Context Similarity

For SCS we use measures of ontology similarity [112]. The *semantic cotopy* $SC(e)$ of a class in a metamodel is the set of all classes related directly or indirectly to e by inheritance. The *upwards cotopy* $UC(e)$ of a class e is the set of ancestors of e, including e. Given a matching cm of classes of MM_1 to the classes of MM_2, a semantic similarity measure based on UC is $CM(e, e') = \frac{\#(UC(e) \cap cm^{-1}(\!| UC(e') |\!))}{\#(UC(e) \cup cm^{-1}(\!| UC(e') |\!))}$. The similarity of reference features of e and e' is also incorporated in the SCS measure, using the UC similarity of the class types or element types of the features.

4.5 DSS Approach

As our approach is to use similarity measures to extract correspondences from metamodel matching, we compared the effectiveness of different similarity, the result is shown in Section 4.6. We found that the DSS performed best, so our metamodel

matching was based primarily on the DSS.

The DSS measures mentioned in Section 4.3 is feature-based, for two flattened classes $A\$$ and $B\$$, the similarity for class mapping is DSS_{class} :

$$DSS_{class} = \frac{max(SumOfTypeSimilarities)}{max(A\$.features.size, B\$.features.size)}$$

We sum the individual DSS_{class} for two metamodels to obtain a mapping score for two metamodels, as DSS approach can handle one-to-one, one-to-many, many-to-one matchings, the mapping score should be divided by the number of matchings between the metamodels, and the matchings corresponding to the largest result are the optimal correspondences obtained by DSS approach. However, sometimes there are multiple matching situations with the same highest mapping score, we therefore use exhaustive DSS as the principal matching technique, with NSS/NMS used to distinguish matchings that have the same score.

In the Tree to Graph example (Figure 4.4 and Figure 4.5), the source metamodel contains 1 class, and the target metamodel contains 2 classes, meaning there are 3 potential class mappings. After calculating these 3 mappings, the mapping score for *Tree*\$ to *Edge*\$ is 0.42, for *Tree*\$ to *Node*\$ is 0.5, and for *Tree*\$ to *Edge*\$&*Node*\$ is 0.46. Therefore, the appropriate matching is *Tree*\$ to *Node*\$, with the correspondences: *Tree\$ [label, parent]* to *Node\$ [name, incoming source]*.

4.6 Evaluation

First, we compare the effectiveness of different similarity measures for metamodel matching on benchmark examples. In Table 4.5 we give the F-measure results for metamodel matching using the different similarity measures, on the 6 benchmark cases [20]. Each of these examples is quite small (with no more than 11 classes in total between the source and target metamodels), however they illustrate a range of situations in which source and target metamodels can differ structurally, linguistically and semantically.

Table 4.5: DSS metamodel matching F-measure results for benchmark cases [20]

Cases number	DSS	NSS	NMS	GSS	GES	SCS
1 (evolution)	0.86	0.86	0.86	0.25	0.44	0.71
2 (abstraction)	0.5	0.75	0.5	0	0.39	0.5
3 (refinement)	1.0	1.0	1.0	0	0.33	1.0
4 (refinement)	1.0	1.0	1.0	0	0.28	1.0
5 (abstraction)	0.71	0.92	0.71	0.09	0.28	0.85
6 (refinement)	1.0	0.75	0.25	0	0.25	0.75
Averages	0.85	0.88	0.72	0.06	0.33	0.8

As can be seen from these results, GSS has generally poor accuracy, and GES has variable accuracy. It also has exponential time complexity for general metamodel graphs [83]. NMS and NSS depend upon linguistic similarities between MM_1 and MM_2. SCS is suitable when the metamodels have similar terminologies and taxonomies [114]. These techniques perform less well than DSS in case 6, where there is substantial change in class names and taxonomies. DSS is the most consistently

accurate similarity measure for metamodel matching, and relatively independent of metamodel terminologies. It is also insensitive to alternative inheritance arrangements, because inheritance hierarchies are flattened in computing all features of a class. Therefore, we base our combined measure on DSS as the primary factor, with NSS and NMS as secondary factors.

In Table 4.6 we show the results of our DSS metamodel matching approach on the large benchmark examples of GAMMA [18]. For each pair of metamodels we indicate the category of transformation and size, which is the numbers of class and feature mappings in original versions. We compare the F-measure achieved by our approach and that reported in GAMMA [18]. We also give the execution time of our metamodel matching approach.

Our F-measure results on these cases are similar to those of GAMMA [18], however the the search procedure is deterministic.

We also evaluated DSS metamodel matching using 10 published ATL cases and 10 published ETL cases, from papers and from the Epsilon example set [21], [22]. These include a range of cases from copy transformations to refinement cases with substantial metamodel differences (OO2DB; uml2Simulink). In Table4.7 and Table 4.8 we quantify the recall, precision and F-measure of our matching result with respect to the mappings of the original versions. For ATL cases, the average F-measure score of our solutions is 0.871, and the average F-measure score for ETL cases of our solutions is 0.882, indicating high accuracy in recapturing the original

Table 4.6: DSS metamodel matching F-measure results for benchmark cases [18]

Case (category; size)	F-measure (DSS)	F-measure (GAMMA [18])	Execution time (s)
EER-WebML (migration; 13)	0.88	0.70	0.9
Ecore-EER (refinement; 22)	0.75	0.52	0.188
Ecore-WebML (refinement; 21)	0.72	0.74	0.125
UML1.4-EER (refinement; 31)	0.83	0.71	0.172
UML2.0-EER (refinement; 35)	0.64	0.68	0.422
UML1.4-WebML (refinement; 30)	0.63	0.91	0.865
UML2.0-WebML (refinement; 34)	0.77	0.81	0.219
UML1.4-Ecore (migration; 39)	0.69	0.79	104.5
Ecore-UML2.0 (migration; 43)	0.77	0.75	510
UML1.4-UML2.0 (evolution; 52)	0.63	0.8	393
Averages	0.73	0.74	101

intent of the transformation cases.

Overall for $RQ1$, our findings indicate that metamodel matching through the DSS approach effectively identifies the majority of transformation mappings in homogeneous metamodel scenarios. In terms of F-measure, our DSS approacha consistently achieves values exceeding 0.8, surpassing the performance reported in existing literature [15]–[17].

Table 4.7: DSS metamodel matching results for ATL cases [22]

Case (category; size)	Recall	Precision	F-measure
Ports (evolution; 7)	1.0	1.0	1.0
PetriNet2PathExp (semantic mapping; 11)	0.78	1.0	0.875
Class2Relational (migration; 10)	1.0	0.88	0.94
PathExp2PetriNet (semantic mapping; 11)	0.94	1.0	0.97
SimpleClass2SimpleRDB (migration; 13)	1.0	0.87	0.93
Ant2Maven (evolution; 109)	0.84	0.99	0.91
Maven2Ant (evolution; 109)	0.92	0.91	0.91
MOF2UML (evolution; 71)	0.69	0.78	0.73
MySQL2KM3 (abstraction; 24)	0.79	0.81	0.8
UML2MOF (evolution; 71)	0.53	0.8	0.64
Averages	0.849	0.904	0.871

However, there are practical barriers to its application in large cases due to the large numbers of possible matchings to be searched. In addition, accuracy can be low in cases where there is high heterogenity between metamodels.

Table 4.8: DSS metamodel matching results for ETL cases [21]

Case (category; size)	Recall	Precision	F-measure
Tree2Graph (refinement; 3)	1.0	1.0	1.0
Competition2TVApp (refinement; 14)	0.56	1.0	0.72
Flowchart2Html (code generation; 12)	1.0	1.0	1.0
CopyTVApp (evolution; 11)	1.0	1.0	1.0
CopyFlowchart (evolution; 14)	1.0	1.0	1.0
RSS2Atom (migration; 20)	0.79	0.86	0.82
ArgoUML2Ecore (migration; 56)	0.89	0.64	0.74
CopyOO (evolution; 14)	1.0	1.0	1.0
OO2DB (refinement; 21)	0.49	0.8	0.61
uml2Simulink (refinement; 52)	0.93	0.94	0.93
Averages	0.866	0.924	0.882

4.7 Conclusion

The sections above of this chapter present our DSS approach. The approach handles 1-1, 1-n and n-1 matchings, however, it cannot handle n-m matchings yet, because this causes a lot of calculations and thus increases the matching time.

In addition, there is a scalability problem for large metamodels: explicit enumeration of all possible class-class matchings becomes too time-consuming for metamodels with 20 or more classes.

Regarding n-m and scalability limitations, some search-based technique such as genetic algorithms (GA) could be used, instead of explicit enumeration of matchings. We will introduce this approach in the next chapter.

Chapter 5

Search-based Optimisation Approach for Metamodel Matching

5.1 Introduction

For metamodel matching, an exhaustive search algorithm has been applied, which lists and calculates all possible matchings and chooses the best matchings among them. However, matching between metamodels is not only in the form of one-to-one, but also one-to-many, many-to-one, and many-to-many. In this case, it is almost impossible to use the exhaustive search algorithm, especially when the metamodels are large-scale. In addition, most work on metamodel matching has

focused on homogeneous metamodels, which is relatively simple because in such cases there is usually a common vocabulary (names of classes and features) in the two metamodels, and relatively small structural differences between them. However, metamodel matching cases in practice are often with heterogeneous metamodels, where there are substantial differences between the structure of the metamodels, and between the terminology of classes and features, which undoubtedly increases the difficulty of matching.

Search-based software engineering (SBSE) [115] uses heuristic search algorithms in the solution space to search for the near optimal solutions by using the fitness functions of the specific problem as a guide. In other words, SBSE expresses activities in software engineering as optimisation problems. The most powerful search-based optimisation algorithms are the genetic algorithm (GA) and is variants [116]. Using SBSE approach with appropriate measures seems to be able to address the challenges (the large-scale problem and the heterogeneity) mentioned above.

In this chapter, we propose a search-based metamodel matching approach. We transform metamodel matching into an optimisation problem, obtainning the optimal metamodel matchings by maximising three measures, Data structure similarity (DSS), Name syntactic similarity (NSS), and Name semantic similarity (NMS), or the combination of these three measures. Our approach focuses on heterogeneous metamodel matching, and can handle one-to-one, one-to-many, many-to-one, and many-to-many matchings.

5.2 Transforming Metamodel Matching Problem into An Optimisation Problem

An optimisation problem attempts to find the optimal solution from all feasible solutions which can maximise or minimise one or more objectives subject to certain constraints. Optimisation problems are usually divided into single-objective optimisation problem and multi-objective optimisation problem. The single-objective problem typically has one objective function. The objective functions of a multi-objective optimisation problem may be mutually exclusive, i.e. when one solution maximises or minimises one objective, the opposite may be true for the other objective. Therefore, in order to satisfy multiple objectives simultaneously, multi-objective optimisation problems often result in a set of non-dominated solutions for which none of the corresponding objective value can be improved without sacrificing at least one of the other objectives, referred to as a Pareto optimal solution set [117].

Compared to some popular single-objective optimization algorithms, such as Simulated Annealing [118] and Particle Swarm Optimization [119], Differential Evolution (DE) algorithm [120] exhibits exceptional performance in global optimization problems, particularly when the objective function presents multiple local optima, making the quest for a global optimum challenging. Furthermore, DE algorithm often converges relatively quickly, making it well-suited for problems that require finding solutions close to the optimum within a limited time frame. These charac-

teristics align well with our requirements for metamodel matching. Deb and Jain proposed a multi-objective optimisation algorithm, NSGA-III [121], which is able to generate Pareto frontiers quickly and find solutions more efficiently compared to other recently suggested algorithms (MOEA/D [122], NSGA-II [123]) by introducing a hierarchical structure as well as a reference point strategy [121], [124].

In this approach, we employ both single-objective and multi-objective optimization approaches to select the results with the highest similarity measures for metamodel matching. For single-objective optimization, we utilize the DE algorithm to combine the three similarity measures into a single objective function. Conversely, for multi-objective optimization, we employ the NSGA-III to independently compute the three similarity measures as separate objectives.

5.3 Search Space Construction

A two-step search was performed, where the first step was on class matching level, when calculating the similarity between source class and target class, the second-step search was executed to calculate the source features and target features matching according to the three measures (DSS, NSS, NMS) respectively.

We employed the metamodel matching situations as individuals. For each class in the source metamodel, we constructed all its matching situations, including empty matching (1-0), only one target class/feature (1-1), and multiple target classes/features matched by it (1-n). Similarly, a target class/feature that had been matched

by a source class/feature may also be matched by other source classes/features, thus constituting n-1 as well as n-m possibilities.

Each metamodel matching situation was encoded as a unique integer. When a source metamodel contained n classes(first-step search)/features(second-step search) and a target metamodel contains m classes (first-step search) / features (second-step search), the lower limit of the search space lower boundary was 0 (the situation with no matching), the search space upper boundary was the species diversity (SD), which was also the total number of all possible matchings for a source class (first-step search)/feature(second-step search):

$$SD = \sum_{i=0}^{m} C_m^i$$

where C_m^i is the combinatorial operator.

The search space varied with steps, and an individual was a size n set with random integer range in [0, SD].

5.4 Objective Function Construction

The objective functions are the functions that to measure similarity between metamodels, the higher result for objective functions, the higher similarity for two metamodels in the specific measure. Here, we introduce three similarity measures DSS, NSS and NMS, for this approach.

5.4.1 Similarity Measures for Objective Function

The DSS and NSS were described in above chapters, however, the NMS in this approach is different. Previous NMS requires a thesaurus, which needs to be provided by users. The advent of Word2Vec [125] has opened up opportunities for calculating NMS without a thesaurus. Word2Vec is a technique for natural language processing, which can convert a series of words into high-dimensional vectors by training, and these vectors have some semantic connection in the vector space in which they are located. Similarity is defined by the frequency of the two words used in the same context. For example, when a Word2Vec model always learns that there is 'ID' in the context of 'Student' (e.g. 'Student ID') and that there is always 'Name' in the context of 'Student' (e.g. 'Student Name'), a certain similarity arises between 'ID' and 'Name', it can then generate a similarity value between 'Name' and 'ID'.

The names of features were tokenised by a Camel Case Splitter [113] first. After that, we used a pre-trained Word2Vec model, which was trained on Google News dataset (about 100 billion words), to calculate NMS between source and target metamodels. The Google News dataset covers a large number of news texts on a variety of topics and domains. Therefore, rich semantic relations and concepts can be captured by semantic similarity models trained on this dataset. On the other hand, models are often named according to real-life terminology, for example, when we create a class to describe a person, we usually name the class 'Person', instead of the name of the animal. Finally, if the names of the elements in the model do

not exist in the dataset, it will not be possible to compute the NMS, whereas the Google News dataset contains a huge amount of text and avoids missing names. Word2Vec can directly calculate similarity for two sets of strings, which happens to help us directly calculate the NMS of two tokenised feature names. The similarity of Word2Vec contains negative values, indicating that the two words present negative semantic correlation, for example NMS(label,outgoing)=-0.051, which helps us to better exclude irrelevant matchings. Similarly, for two empty classes, NMS is 1.

5.4.2 Objective Functions

Optimization algorithms typically aim to minimize or maximize objective functions. In our specific problem, the objective is clearly to maximize the objective function. Although the majority of common optimization algorithms are designed with a focus on achieving optima close to zero, they often provide interfaces to facilitate the maximization of objective functions. Thus, we can easily utilize these algorithms without significant modifications. In more detail, it involves simply negating the original minimization objective function within its specified value range, effectively transforming it into a maximization problem.

The similarity measures for this task are feature-based, so for the similarity of two metamodels, an overall similarity can be derived by summing the similarities of the matched features between metamodels. However, in a search-based method scenario, this poses a problem in that the more matchings there are, the higher the

sum of similarities would be, eventually leading to a situation that there is a matching between any features in the source and target metamodels, which is obviously unreasonable. Therefore, the sum of each feature similarity need to be divided by the number of matchings between source metamodel and target metamodel, this will effectively exclude matches with low similarity.

When there are k matchings between two metamodels, p_i ($i \in \{1, 2, ...k\}$) is a pair of source feature and target feature for one of these matchings, the three objective functions OF_{DSS}, OF_{DSS} and OF_{DSS} are:

$$OF_{DSS} = \frac{\sum_{i=1}^{k} DSS(p_i)}{k}$$

$$OF_{NSS} = \frac{\sum_{i=1}^{k} NSS(p_i)}{k}$$

$$OF_{NMS} = \frac{\sum_{i=1}^{k} NMS(p_i)}{k}$$

These three three objective functions can be used for multi-objective optimisation directly. However, for single-objective optimisation, the single-objective function S_OF needs to combine the three measures of similarity equi-weightedly:

$$S\_OF = \frac{OF_{DSS} + OF_{NSS} + OF_{NMS}}{3}$$

One important thing to note is that a reasonable value for 1-0 matching needs to be set. Although in theory it should be 0, this would lead to a higher similarity for matching any class/feature than for matching an empty class/feature, resulting in no empty matching occurring at all, but empty matching is very common in practice. We matched a number of metamodels from previous studies [20]–[22], [126], averaged the DSS, NSS and NMS for correct and incorrect matchings respectively, and then used the median of the correct and incorrect averages as the similarity of the matches to an empty class.

5.5 Selecting One Solution Using Machine Learning

For both single-objective and multi-objective optimisation problems, the results may not be unique. In practice, these results may all be considered as correct results, as there may genuinely exist multiple valid transformations between two metamodels. An example is UML to ER transformation [127], which have several valid and different transformations that could be defined from UML model to ER model.The choice between these transformations would need to be made by a user. An automated search procedure could produce all these solutions. However, developers often need to select one that aligns more closely with their specific requirements. The challenge is how to select one solution from these multiple solutions. When specific require-

ments are known, such as prioritizing DSS similarity, developers can easily choose the one with the highest DSS similarity from multiple optimal solutions. However, when these requirements are not well-defined, in addition to manual selection by the user, a more efficient and feasible approach is summarising preferences from past matchings, similar approach using machine learning for solution-selection has been proposed and proven to be feasible in other fields [128].

We proposed a machine learning-based approach to select one solution from multiple optimal solutions. We built a model for regression analysis and trained it with a number of metamodels with manually matched from previous studies [20]–[22], [126]. The purposed regression analysis model is to learn from previous matchings and summarise developers' preferences, enabling the selection of the solution that aligns most closely with developers' preference from multiple optimal solutions.

The four most popular regression analysis methods, Linear Regression (LR) [129], Support Vector Machine (SVM) [130], Automatic Relevance Determination Regression (ARD) [131], and Multilayer Perceptron (MLP) [132] were used to construct the models as well as for comparison. LR is a fundamental statistical technique, which seeks to establish linear relationships between dependent and one or more independent variables. Its objective is to predict the value of the dependent variable by fitting the best-fitting straight line. Linear regression is primarily employed when dealing with continuous dependent variables [129]. While commonly associated with classification, SVM's utility extends to regression tasks as well. SVM regression aims

to find a function that optimally fits the data while minimizing prediction errors. It relies on the concepts of support vectors and kernel functions, making it suitable for nonlinear regression problems [130]. ARD regression belongs to the Bayesian regression family and focuses on identifying the relevant independent variables within a model. It automatically determines which predictors influence the target variable, enhancing model interpretability [131]. MLP, an artificial neural network model, is frequently employed for nonlinear regression problems. Comprising multiple layers of neurons, each connected to the previous layer, MLP learns complex nonlinear relationships between inputs and outputs through training. It excels in capturing intricate data patterns [132].

5.6 Evaluation

The state-of-the-art approach for metamodel matching is GAMMA [18], which is also an SBSE method. It outperforms some previous work on ten pairs of metamodel matching cases [18]. We evaluated our approach on these ten cases as well to demonstrate our performance and compare it with the state-of-the-art. To avoid the bias caused by random search, we performed twenty searches and took the average value as our final result.

We first verified the accuracy of the trained regression analysis models. Because our model is designed to select the best fit solution from multiple optimal solutions that match past metamodel matching requirements, we conducted a comparison

against manual selection and calculated accuracy. Table 5.1 shows the accuracy of the trained regression analysis models. After validation, we chose the MLP-based model for subsequent evaluation.

Table 5.1: The accuracy of the trained regression analysis models

Regression Algorithms	Accuracy
LR [129]	69.3%
SVM [130]	72.2%
ARD [131]	70.2%
MLP [132]	82.4%

In addition, we evaluated the effectiveness of the matchings obtained by our multi-objective optimisation approach and single-objective optimisation approach (Table 5.2). We also utilized the Vargha-Delaney effect size comparison, a non-parametric measure for comparing the magnitude of differences between two independent groups across various metrics, to assess the performance differences between our multi-objective optimization approach and the single-objective optimization approach, particularly focusing on precision, recall, and F-score (Table 5.3). The results demonstrated that the multi-objective approach was better in terms of precision. Conversely, in terms of recall, the single-objective approach outperformed. The F-score comparison showed a more balanced performance between the two approaches, slightly leaning towards the multi-objective approach.

We also counted the running time for multi-objective optimisation (Table 5.4) and single-objective optimisation (Table 5.5), since the initial population size and

Table 5.2: Evaluation of multi-objective and single-objective optimization approaches on cases of GAMMA [18]

Case	Precision (GAMMA)	Recall (GAMMA)	F-measure (GAMMA)	Precision (MOO)	Recall (MOO)	F-measure (MOO)	Precision (SOO)	Recall (SOO)	F-measure (SOO)
EER 2 WebML	0.69	0.72	0.70	0.86	0.63	0.73	0.76	1.00	0.86
UML1.4 2 UML2.0	1.00	0.67	0.80	0.95	0.76	0.84	0.91	0.92	0.92
UML1.4 2 EER	0.71	0.72	0.71	0.89	0.65	0.76	0.64	0.84	0.72
UML1.4 2 WebML	1.00	0.84	0.91	0.86	0.96	0.91	0.68	0.79	0.73
UML1.4 2 Ecore	1.00	0.66	0.79	0.91	0.69	0.79	0.88	0.97	0.92
Ecore 2 WebML	0.82	0.69	0.74	0.88	1.00	0.93	0.91	0.89	0.90
Ecore 2 UML2.0	0.64	0.89	0.75	0.93	0.67	0.78	0.80	0.89	0.84
Ecore 2 EER	0.48	0.59	0.52	0.88	1.00	0.94	0.76	0.96	0.85
UML2.0 2 EER	0.67	0.72	0.68	0.56	0.50	0.53	0.32	0.90	0.47
UML2.0 2 WebML	0.91	0.73	0.81	0.82	0.92	0.87	0.71	0.92	0.80
Average	0.79	0.72	0.74	0.85	0.78	0.81	0.74	0.83	0.78

Table 5.3: Vargha-Delaney effect size comparison of multi-objective and single-objective optimization approaches

Evaluation metrics	Vargha-Delaney effect size
Precision	0.74
Recall	0.325
F-score	0.52

the maximum number of evolutionary generations were set to vary for different metamodel pairs as the population diversity was different. Here, we counted the execution time for a population size of 100 and a maximum number of generations

to evolve of 10. The search time is proportional to both the initial population size and the number of generations, which means that when the initial population size becomes 200 and the maximum number of generations evolves to 100, the time required is approximately 20 times the time in the table.

Table 5.4: Execution time for population size 100 with 10 generations for multi-objective optimisation

Case	Time (s)	Case	Time (s)
EER 2 WebML	22.45s	Ecore 2 WebML	24.74s
UML1.4 2 UML2.0	1521.15s	Ecore 2 UML2.0	814.61s
UML1.4 2 EER	37.00s	Ecore 2 EER	41.11s
UML1.4 2 WebML	27.96s	UML2.0 2 EER	58.86s
UML1.4 2 Ecore	395.79s	UML2.0 2 WebML	52.27s

In summary, with respect to $RQ2$, our results suggest that the metamodel matching approach utilizing a search-based optimization method exhibits the capability to not only accommodate large-scale metamodels but also enhance matching accuracy when compared to the Decision Support System (DSS) approach and GAMMA [18].

Regarding the time aspect, both single-objective and multi-objective optimiza-

Table 5.5: Execution time for population size 100 with 10 generations for single-objective optimisation

Case	Time (s)	Case	Time (s)
EER 2 WebML	3.13s	Ecore 2 WebML	5.94s
UML1.4 2 UML2.0	202.02s	Ecore 2 UML2.0	133.31s
UML1.4 2 EER	4.07s	Ecore 2 EER	8.32s
UML1.4 2 WebML	6.08s	UML2.0 2 EER	10.26s
UML1.4 2 Ecore	65.88s	UML2.0 2 WebML	8.66s

tion approaches demonstrate acceptable performance. It is evident that single-objective optimization is notably faster, but the multi-objective optimization outperforms it in terms of average F-measure. Notably, both single-objective and multi-objective optimization approaches achieve F-measure values surpassing those of GAMMA.

5.7 Conclusion

In this chapter, we proposed an approach to address the challenge of automated or semi-automated metamodel matching. To address the issue, we defined a search-

based metamodel matching approach combining single-objective or multi-objective optimisation and machine learning (MLP). The approach handles one-to-one, one-to-many, many-to-one, many-to-many matchings, which allows it to be applied to various metamodel matching. This approach also addresses the scalability limitation which cannot be addressed by DSS approach in Chapter 4

However, this study has its limitations. The Word2Vec model used in this study was trained on Google News dataset, and in the future we will investigate on building a model exclusively for MDE to obtain more accurate NMS. Both approaches for metamodel matching have a limitation that is precise mapping of feature values cannot be identified: different possible mappings of one integer value to another are not distinguished, also mappings of one string value to another are not distinguished. Only direct value mapping can be performed.

Chapter 6

Synthesis of Model Transformations from Metamodel Matching

6.1 Introduction

In this chapter, we will introduce the process of synthesis of model transformations after extracting correspondences by metamodel matching. The first step is using an intermediate language $\mathcal{TL}$ to formalise the extracted correspondences, including consistency and completeness checks. The second step is synthesising multiple MT language specifications from $\mathcal{TL}$. Figure 6.1 shows the process of the approach.

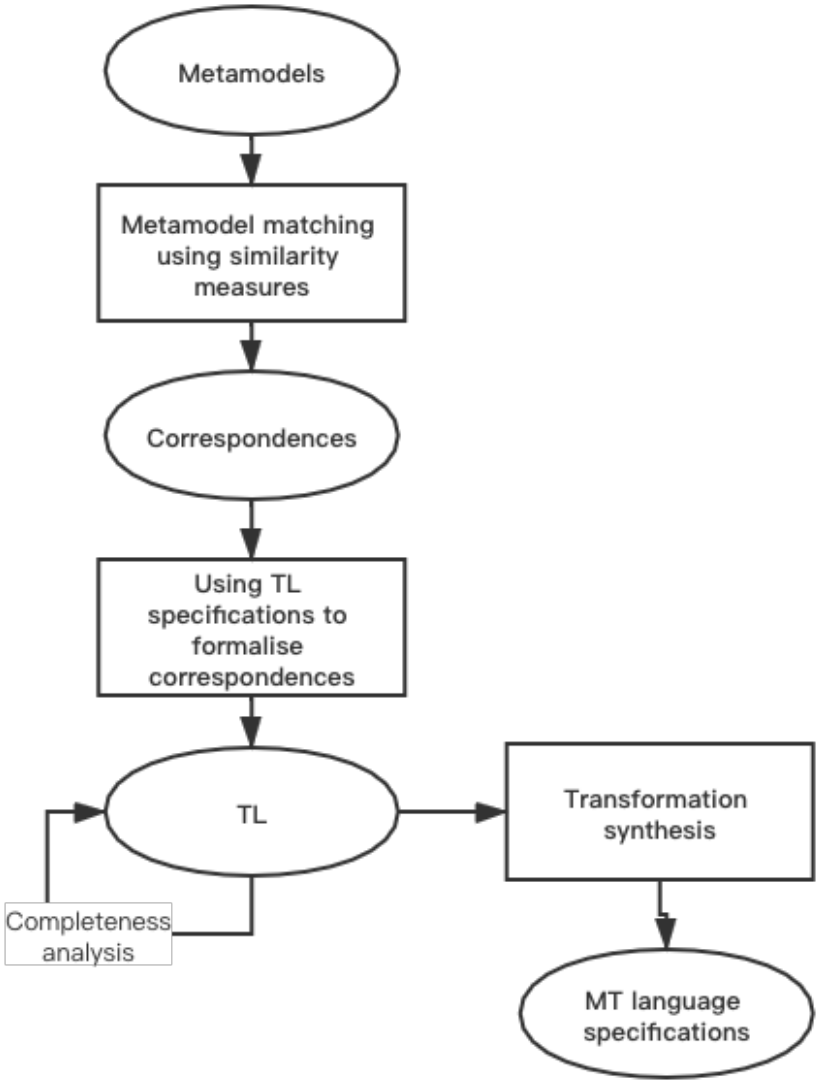


Figure 6.1: Synthesising multiple model transformation language specifications by metamodel matching

6.2 $\mathcal{TL}$ Specification

To enable the production of specifications in multiple MT languages from correspondences, we introduce an intermediate language $\mathcal{TL}$ that uses a simplified transformation notation to express transformation specifications in a language-independent manner, avoiding having separate mappings from correspondences to many languages, there is only one mapping to $\mathcal{TL}$.

The notation expresses class correspondences of E to F by the notation $E \longmapsto F$, and feature correspondences of f to g by the notation $f \longmapsto g$.

A $\mathcal{TL}$ specification consists of a set of rules of the form

$$
\begin{array}{l}
\{PreCond\}\ E \longmapsto F \\
\quad p_1 \longmapsto q_1 \\
\quad \vdots \\
\quad p_n \longmapsto q_n
\end{array}
$$

where the p_i are features (owned, inherited or composed) of E or OCL expressions in such features, and the q_i are distinct features of F. The optional *PreCond* is a Boolean-valued OCL expression in the p_i.

The meaning of a class mapping $\{C\}\ E \longmapsto F$ as a model transformation rule is that for every instance e of E that satisfies C, there is a corresponding instance e' of F. Distinct instances of E map to distinct instances of F. A feature mapping $p \longmapsto q$ of $\{C\}\ E \longmapsto F$ means that for corresponding instances $e : E$, $e' : F$, the value of $e'.q$ is the interpretation $(e.p)'$ of $e.p$ via the class mappings.

For example, based on the correspondences in the illustrative example, the $\mathcal{TL}$ notation for the transformation from Tree to Graph is:

$$
\begin{aligned}
&\mathit{Tree} \longmapsto \mathit{Node} \\
&\quad \mathit{label} \longmapsto \mathit{name} \\
&\quad \mathit{parent} \longmapsto \mathit{incoming.source}
\end{aligned}
$$

Having obtained such possible correspondences, the next step is typically to examine them for incompleteness or inconsistency. For example, a measure of incompleteness of the $\mathcal{TL}$ specification is the proportion of source classes in its class matchings, as a proportion of the total number of source classes. For example, if there are 10 source classes but only 5 appear in the matchings, we could infer that our matchings are likely to be incomplete and that some matchings are missing, in this case, we may need to check or modify the $\mathcal{TL}$ specification. Similarly, the proportion of source features that are matched by the specification. The proportion of target classes and features which are used by the matchings could also be measured. Our approach partially automates the completeness measure by recognising cases of incompleteness and proposing mapping strategies (additional mappings) to resolve them, and asking the user to confirm these solutions. Table 6.1 summarises the different checks which we defined in our previous work [133].

By using the intermediate $\mathcal{TL}$ representation, we facilitate production of transformation code in multiple MT languages, which is an improvement on existing ap-

Table 6.1: Consistency and completeness checks [133]

Issue	Correction
Class mapping $Sub \longmapsto T$ for Sub subclass of E, has T not subclass/or equal to F, where $E \longmapsto F$	Retarget Sub mapping, or introduce additional target splitting map $Sub \longmapsto F$
Two directions of bidirectional association r not mapped to mutually reverse target features	Modify one feature mapping to ensure consistency
Source, target have different multiplicities	Propose modified mappings
Unused target subclasses $FSub$ of F, where $E \longmapsto F$	Introduce discriminator condition $FSubC$ and mapping $\{FSubC\}E \longmapsto FSub$
Unused source or target feature f	Suggest class or feature mapping that uses f
Feature mapping $f \longmapsto r.g$ with $r : R$ of abstract type/element type	Propose concrete subclass $RSub$ of R for this instantiation

proaches. The developer effort goes into refining and improving the $\mathcal{TL}$ matchings, once this is completed, transformations in specific languages can be automatically generated without additional effort.

6.3 Generating ATL Specifications from $\mathcal{TL}$

Once a complete and consistent $\mathcal{TL}$ specification is obtained, the next step is to generate an implementation of this in a particular MT language. Different MT languages have different properties in different situations (ATL is the most popular in industry [134], ETL uses more program-like constructs[21]; ATL is effective for separate models transformations but poorly supports update-in-place transformations, for which ETL is more suitable; ATL is more declarative, whilst ETL is more imperative in style), developers may need to produce different transformations. On the other hand, integration of different languages into the same approach can increase the applicability of the approach. In this section we describe the steps of generating ATL specifications from $\mathcal{TL}$.

For each $\mathcal{TL}$ rule $\{Cond\}\ E \longmapsto F$, with F a concrete class, there is an ATL matched rule of the schematic form

```
1 rule E2F
2 { from ex : MM1!E ( ex.Cond )
3   to fx : MM2!F
4   ( ... )
5 }
```

In the case that there are two or more $\mathcal{TL}$ rules for E with overlapping (non disjoint) conditions $Cond$, these must be combined into a single ATL rule with

multiple output pattern elements.

For example, rules $E \longmapsto F1$ and $E \longmapsto F2$ with concrete $F1$, $F2$ would be implemented by:

```
rule E2F1F2
{ from ex : MM1!E
  to f1x : MM2!F1
  ( ... ),
  f2x : MM2!F2
  ( ... )
}
```

This means also that source expressions e of element type E must be disambiguated when used on the RHS of bindings: $t \leftarrow e$ refers to e converted to $F1$ elements via $E2F1F2$, whilst $t \leftarrow thisModule.resolveTemp(e, `f2x')$ refers to e converted to $F2$ elements via the rule.

A feature mapping $f \longmapsto g$ is represented by an ATL binding $g \leftarrow ex.f$ in cases where f and g are not composed. Source compositions $r.f$ for f of 1 multiplicity, r not of 1 multiplicity, are evaluated as $ex.r \rightarrow collect(\_x \mid \_x.f)$.

In the case of a feature mapping $f \longmapsto r.g$, where r is of 1-multiplicity or 0..1 multiplicity, of concrete class element type R, a new output pattern element $rx : MM2!R$ is defined:

```
rule E2F
```

```
{ from ex : MM1!E ( ex.Cond )
  to fx : MM2!F
  ( r <- rx ),
  rx : MM2!R
  ( ... )
}
```

The bindings for rx implement the feature mappings for $f \longmapsto g$, and for other mappings $k \longmapsto l$, for each $k \longmapsto r.l$ which is a feature mapping of the $E \longmapsto F$ mapping.

In the case that there is a direct mapping $f \longmapsto r$ and also composed mappings $g \longmapsto r.h$, a *do* clause is needed for the composed mappings:

```
rule E2F
{ from ex : MM1!E ( ex.Cond )
  to fx : MM2!F
  ( r <- ex.f )
  do
  { fx.r.h <- ex.g; }
}
```

where r is of 1 multiplicity.

For 0..1 and for *-multiplicity r, a *do* clause implementation of $g \longmapsto r.h$ instead has the form

```
for (rx in fx.r)
{ rx.h <- ex.g; }
```

If there is no direct mapping $f \longmapsto r$, r is of $*$ multiplicity, and the only feature mappings $f \longmapsto r.g$ of $E \longmapsto F$ are such that f's upper multiplicity is 1[1], then r can be defined by an output pattern $rx : MM2!R$ which has bindings $g \leftarrow ex.f$ for each $f \longmapsto r.g$.

If there are cases of mappings $f \longmapsto r.g$ where f and r are of *-multiplicity, then instead sets of R elements are created using unique lazy or called rules. In this case, if f has a class element type $E1$, the mapping $f \longmapsto r.g$ in cases where r and f have * multiplicity can be implemented by introducing a new unique lazy rule:

```
rule E2F
{ from ex : MM1!E ( ex.Cond )
  to fx : MM2!F
  ( r <- ex.f->collect( e1x |
            thisModule.MapE12Rg(e1x) )  )
}

unique lazy rule MapE12Rg
{ from e1x : MM1!E1
  to rx : MM2!R
```

[1]ie., f is of 1 or 0..1 multiplicity

```
11    ( g <- e1x )
12  }
```

The effect of this approach is to produce a set of R objects, one for each element $e1x$ of $ex.f$. Again, R must be a concrete class for this to be valid. An example of this situation is the *supertypes* $\longmapsto$ *generalization.parent* mapping in the MOF2UML case.

Further updates to the r with additional mappings $k \longmapsto r.l$ with l of higher or equal upper multiplicity to k must be handled in the *do* clause of the $E2F$ rule, via a statement

```
1  for (rx in fx.r)
2  { rx.l <- ex.k; }
```

Further updates to the r with additional mappings $k \longmapsto r.l$ with l of smaller upper multiplicity than k require the creation of additional R objects:

```
1  ( r <- ex.k->collect( e2x |
2            thisModule.MapE22Rl(e2x)) )
```

If instead f has a value element type T, the mapping $f \longmapsto r.g$ can be implemented by introducing a new called rule:

```
1  rule E2F
2  { from ex : MM1!E ( ex.Cond )
3    to fx : MM2!F
```

```
4     ( r <- ex.f->collect( tx |
5                 thisModule.MapT2Rg(tx) )  )
6  }
7
8  rule MapT2Rg(tx : T) : MM2!R
9  { to rx : MM2!R
10    ( g <- tx )
11 }
```

Table 6.2 summarises the translation from $\mathcal{TL}$ to ATL for composed target feature mappings $f \longmapsto r.g$, where there is no direct mapping $h \longmapsto r$.

Regarding ATL limitations, ATL does not currently support operators $\rightarrow$*closure*, $\rightarrow$*unionAll*, $\rightarrow$*before* or $\rightarrow$*after*, which are used by the correspondence construction and enhancement process, as detailed in Section 6.2. Thus these must be implemented by appropriate library operations in terms of OCL operators supported by ATL.

The synthesised ATL satisfies the quality recommendations of ATL manual [134]:

- Use standard and unique lazy rules in preference to lazy rules – lazy rules are not needed by our translation.
- Avoid imperative constructs and the use of *resolveTemp* – imperative code and called rules are only needed in cases of composite target features. *resolveTemp*

Table 6.2: Generated ATL for composite target feature mappings $f \longmapsto r.g$ of $E \longmapsto F$ (for different multiplicity situations)

r	f	g	ATL implementation
1 or 0..1	any	any	$r \leftarrow rx$ for new OutPatternElement $rx : MM2!R$ (...) rx binding is $g \leftarrow ex.f$ for g upper multiplicity $\geq f$ upper multiplicity, $g \leftarrow$ *if* $ex.f \rightarrow notEmpty()$ *then* $ex.f \rightarrow any()$ *else null endif* otherwise. All bindings due to $E \longmapsto F$ mappings $k \longmapsto r.l$ are defined in the rx pattern.
*	1 or 0..1	any	$r \leftarrow rx$ for new OutPatternElement $rx : MM2!R$ (...) rx binding is $g \leftarrow ex.f$
*	*	any	$r \leftarrow ex.f \rightarrow collect(e1x \mid thisModule.MapE12Rg(e1x))$ Additional $k \longmapsto r.l$ with upper multiplicity $k \leq$ upper multiplicity l: *for* $(rx$ *in* $fx.r)$ { $rx.l \leftarrow ex.k$; } Additional $k \longmapsto r.l$ with upper multiplicity $k >$ upper multiplicity l: $r \leftarrow ex.k \rightarrow collect(e2x \mid thisModule.MapE22Rl(e2x))$

is only needed in cases of vertical entity splitting.

- Use *collect* in preference to *iterate* – *iterate* is only needed to define the *closure* operator, otherwise *collect* is used.

More precisely, the situation with regard to the quality flaws [13] is that cycles of calling dependencies ($CBR_2 > 0$) are not possible, while errors of excessive fan-in, fan-out and calling dependencies can arise only because of calls to data conversion operators or to auxiliary unique lazy/called rules in the case of composite target features. Duplicate code may arise if two subclasses $E1$, $E2$ of a source class E have common feature mappings – this duplication could be removed by using ATL rule

inheritance. Excessive rule and transformation size may occur due to the size of the metamodels.

We use the case study of transformation between Tree and Graph to illustrate the approach:

```
rule Tree2Node
transform tree_x : MM1!Tree
to node_x : MM2!Node
{
if (tree_x.parent.isDefined()) {
node_x.incoming= Set{tree_x.parent.equivalent('
   MapTree2Edgesource')
}; }
node_x.name = tree_x.label;
}

@lazy
rule MapTree2Edgesource
transform tree_x : MM1!Tree
to incoming_edge_x : MM2!Edge
{
incoming_edge_x.source = (tree_x).equivalent('Tree2Node');
      }
```

6.4 Generating ETL Specifications from $\mathcal{TL}$

ETL is a widely-used hybrid transformation language within a general MDE and MT framework, Epsilon [80]. However it lacks a formal semantics, and published ETL specifications seem to have a high frequency of flaws or code 'smells' [13], [135], in particular due to the use of implicit rule invocation (the *equivalent*() operator).

Examining ETL cases, we find that the powerful hybrid nature of the language sometimes leads specifiers to (i) overuse imperative coding, which makes code more complex and increases the error probability; (ii) use complex ad-hoc navigations in the target model to link target elements, with cycles of mutually-dependent rules, which make it difficult to understand; (iii) overuse implicit invocation, which makes logic unclear, and brings uncertain problem. For example, the uml2Simulink transformation (https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/examples/org.eclipse.epsilon.examples.uml2simulink).

An automated synthesis procedure for ETL could help to reduce such problems by (i) using declarative constructs where possible; (ii) using standard strategies for navigating the target model; (iii) always using the version of *equivalent* parameterised by an explicitly-named rule to be invoked.

We assume that the essential and additional constraints of Section 6.2 hold. For each $\mathcal{TL}$ rule $\{Cond\}\ E \longmapsto F$, with E a concrete class, we generate an ETL concrete rule of the schematic form

```
1 rule E2F
```

```
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
  ...
}
```

A rule for abstract E is instead defined as an abstract ETL rule:

```
@abstract
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
  ...
}
```

When these ETL rules are used to convert the data of an E-typed MM_1 feature r to data of an F-typed MM_2 feature rr, the conversion of data is referred to as $r.equivalent('E2F')$. Because of the construction process of the $\mathcal{TL}$ specification τ, distinct rules of τ always either have distinct source classes or (for class-splitting cases) distinct target classes. Hence, there can only be one ETL rule with the given name.

ETL supports rule inheritance, this can be used in cases where both a general rule $\{Cond\}\ E \longmapsto F$ and a specialised rule $\{Cond1\}\ ESub \longmapsto F1$ are present in

the $\mathcal{TL}$ specification τ, with *ESub* a subclass of E and $F1$ equal to F or a descendant of F. The specialised rule is implemented as:

```
rule ESub2F1
  transform esubx : MM1!ESub
  to f1x : MM2!F1
  extends E2F {
  guard: ( esubx.Cond1 )
  ...
}
```

Feature mappings which are common to the generalised and specialised rules do not need to be explicitly implemented in $ESub2F1$. If $Cond1$ is the same as $Cond$, the guard of $ESub2F1$ can be omitted.

A feature mapping $f \longmapsto g$ of class mapping $\{Cond\}\ E \longmapsto F$ is represented by an ETL assignment $fx.g = ex.f$; in cases where f and g are of value types and are not composed. Source compositions $r.f$ are evaluated as $ex.r.f$. If f and g are of class types, then $f \longmapsto g$ is implemented by

```
  fx.g = ex.f.equivalent('ERef2FRef');
```

where f and g are not composed, and *ERef* is the class type of f, and *FRef* the class type of g[2]. Likewise for composed source features $r.f$ of class type:

[2]If there is no class mapping $ERef \longmapsto FRef$ then the most specific mapping with source class equal to or an ancestor of *ERef* and target equal to or a descendant of *FRef* is quoted. Such a mapping must exist because of the *Metamodel matching* constraints of Section 6.2.

```
1   fx.g = ex.r.f.equivalent('ERef2FRef');
```

If f or $r.f$ is of 0..1 multiplicity, the assignments are guarded:

```
1   if (ex.f.isDefined())
2   { fx.g = ex.f.equivalent('ERef2FRef'); }
```

and

```
1   if (ex.r.f.isDefined())
2   { fx.g = ex.r.f.equivalent('ERef2FRef'); }
```

In the case of a feature mapping $f \longmapsto r.g$, where r is of 1-multiplicity or 0..1 multiplicity, of concrete class type R, a new variable rx of type $MM2!R$ is defined:

```
1  rule E2F
2    transform ex : MM1!E
3    to fx : MM2!F {
4    guard: ( ex.Cond )
5      if (ex.f.isDefined())
6      { var rx = new MM2!R;
7        fx.r = rx;
8        rx.g = ex.f.equivalent('ERef2G');
9        ...
10     }
11 }
```

The conditional test is included if f is of 0..1 multiplicity. Assignments to rx features implement the feature mapping $f \longmapsto g$, and other mappings $k \longmapsto l$, for each $k \longmapsto r.l$ which is a feature mapping of the $E \longmapsto F$ mapping.

If there is a direct mapping $f \longmapsto r$ and also composed mappings $g \longmapsto r.h$, assignments are needed for the composed mappings:

```
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
    fx.r = ex.f.equivalent('ERef2R');
    fx.r.h = ex.g.equivalent('G2H');
}
```

where r is of 1 multiplicity and f is of type $ERef$.

For other multiplicity r, a *for* loop implementation of $g \longmapsto r.h$ is used:

```
  for (rx in fx.r)
  { rx.h = ex.g.equivalent('G2H'); }
```

If there is no direct mapping $f \longmapsto r$, r is of $*$ multiplicity, and the only feature mappings $f \longmapsto r.g$ of $E \longmapsto F$ are such that f's upper multiplicity is 1[3], then r can be defined by an additional variable $var\ rx = new\ MM2!R;\ rx.g = ex.f.equivalent('ERef2G');$ which has additional assignments $rx.l = ex.k.equivalent('K2L');$

[3]ie., f is of 1 or 0..1 multiplicity

for each $k \longmapsto r.l$, k of type K, l of type L. The rx creation is conditional on $ex.f.isDefined()$ if f is 0..1 multiplicity. Likewise, additional assignments are conditional if k is of 0..1 multiplicity.

If there are cases of mappings $f \longmapsto r.g$ where f and r are of *-multiplicity, then instead sets of R elements are created using lazy rules or operations. In this case, if f has a class type $ERef$, and $r.g$ has class type G, the mapping $f \longmapsto r.g$ in cases where r and f have * multiplicity can be implemented by introducing a new lazy rule:

```
1  rule E2F
2    transform ex : MM1!E
3    to fx : MM2!F {
4    guard: ( ex.Cond )
5    fx.r = ex.f.equivalent('MapERef2Rg');
6  }
7
8  @lazy
9  rule MapERef2Rg
10   transform erefx : MM1!ERef
11   to rx : MM2!R {
12     rx.g = erefx.equivalent('ERef2G');
13 }
```

The effect of this approach is to produce a set of R objects, one for each element *erefx* of $ex.f$. R must be a concrete class for this to be valid.

Further updates to the r with additional $E \longmapsto F$ mappings $k \longmapsto r.l$ with $l.upper = 0$ or $l.upper \geq k.upper$ must be handled in further *for* loops of the $E2F$ rule:

```
for (rx in fx.r)
{ rx.l = ex.k.equivalent('K2L'); }
```

Further updates to the r with additional $E \longmapsto F$ mappings $k \longmapsto r.l$ with $l.upper \neq 0$ and $k.upper = 0$ require the creation of additional R objects via a new lazy rule $MapE22R1$:

```
  fx.r.addAll(ex.k.equivalent('MapE22R1'));
```

If instead f has a value type T, the mapping $f \longmapsto r.g$ can be implemented by introducing a new called operation:

```
rule E2F
  transform ex : MM1!E
  to fx : MM2!F {
  guard: ( ex.Cond )
  fx.r.addAll(f.MapT2Rg());
}

operation T MapT2Rg()
```

```
{ var rx = new MM2!R;
  rx.g = self;
  return rx;
}
```

The generated code has a systematic structure, and all calls of rules via *equivalent* have been made explicit. Duplication of code due to common feature mappings of inheritance related $\mathcal{TL}$ rules is avoided by using ETL rule inheritance. However, excessive rule and transformation size may occur due to the size of the metamodels.

Theorem 1. Assuming all constraints of Section 6.2, the generated ETL specification is consistent with the semantics of the $\mathcal{TL}$ specification [133].

Proof. As noted above, ETL semantics is only informally defined in Epsilon [80], so we will only be able to provide an informal argument for correctness in terms of that semantics.

For each $\mathcal{TL}$ rule $\{Cond\}\ E \longmapsto F$, there is a corresponding generated ETL rule which maps E to F under condition $Cond$. The ETL rule is abstract iff the $\mathcal{TL}$ rule is, and the same structure of rule inheritance is used in both specifications. Feature mappings $f \longmapsto g$ of class mapping $\{Cond\}\ E \longmapsto F$ with f and g of value types are directly implemented by corresponding assignments in the ETL rule. As described above, feature mappings of class-typed f and g, with g direct, are implemented by

```
1 fx.g = ex.f.equivalent('ERef2FRef');
```

where $f.type = ERef$, $g.type = FRef$, or by a conditional assignment in the case f is of 0..1 multiplicity. In ETL, the element(s) *erefx* of $ex.f$ are converted to corresponding *FRef* elements *frefx* by looking *erefx* up in the traces of rule $ERef2FRef$ or (if $ERef2FRef$ is abstract) of its concrete specialisations [80]. If a trace exists, the corresponding *frefx* element(s) are returned. Otherwise, the most specific applicable concrete rule (lazy or non-lazy) for *erefx* is executed to obtain *frefx*. This corresponds to the $\mathcal{TL}$ object resolution semantics, provided that the rule execution in ETL terminates. For our generated ETL this property is ensured.

In the case of feature mappings $f \longmapsto r.g$ with composed target features, $r.type = R$, $g.type = G$, the ETL code defines a particular strategy for ensuring that $fx.r.g = (ex.f)'$ for corresponding instances $ex : E$, $fx : F$, $fx = ex'$. There are several cases to consider, depending on the multiplicities of r, g and f (Table 6.3). For simplicity we only consider multiplicities with upper bound either 0 (*) or 1.

In the first two cases the creation of a single R instance rx and the assignments to its features ensure that $fx.r.g = (ex.f)'$ in terms of the $\mathcal{TL}$ semantics (Section 6.2). In the third case, the strategy for combining multiple mappings $f \longmapsto r.g$, $k \longmapsto r.l$ with l of 1 or 0..1 multiplicity and k of * multiplicity, is to create a new set

Table 6.3: Generated ETL for composite target feature mappings $f \longmapsto r.g$ of $E \longmapsto F$ (for different multiplicity situations)

r	f	g	ETL implementation
1 or 0..1	1 or *	any	$var\ rx\ =\ new\ MM2!R;\ fx.r = rx;$ $rx.g = ex.f.equivalent('ERef2G');$
1 or 0..1	0..1	any	$if\ (ex.f.isDefined())\ \{$ $var\ rx\ =\ new\ MM2!R;\ fx.r = rx;$ $rx.g = ex.f.equivalent('ERef2G');\ ...\ \}$
*	1	any	$var\ rx\ =\ new\ MM2!R;\ fx.r = rx;$ $rx.g = ex.f.equivalent('ERef2G');$
*	0..1	any	$if\ (ex.f.isDefined())\ \{$ $var\ rx\ =\ new\ MM2!R;\ fx.r = rx;$ $rx.g = ex.f.equivalent('ERef2G');\ ...\}$
*	*	any	$fx.r.addAll(ex.f.equivalent('MapERef2Rg'));$ Additional $k \longmapsto r.l$ with $k.upper = 1$: $for\ (rx\ in\ fx.r)\ \{\ rx.l = ex.k.equivalent('K2L');\ \}$ Additional $k \longmapsto r.l$ with $k.upper = 0$, $l.upper = 1$: $fx.r.addAll(ex.k.equivalent('MapERef2Rl'));$ Additional $k \longmapsto r.l$ with $k.upper = 0$, $l.upper = 0$: $for\ (rx\ in\ fx.r)\ \{\ rx.l = ex.k.equivalent('K2L');\ \}$

of r elements r_x for each $x \in ex.k$, with $r_x.l$ set to x'. This ensures that the equation $fx.r.l = (ex.k)'$ holds, provided that other $fx.r$ elements (created for $f \longmapsto r.g$ or other mappings to r features) have an *null* value for their l feature (equivalent to an empty collection [136]), so do not contribute to the $fx.r.l$ collection.

Similarly, we use the case study of transformation between Tree and Graph to verify the approach. The ETL transformation from Tree to Graph is an example of case 10 in Table 4.5:

```
rule Tree2Node
    transform t : Tree!Tree
    to n : Graph!Node {
    n.label := t.label;

    if (t.parent.isDefined()) {
    var edge := new Graph!Edge; edge.source := n;
    edge.target := t.parent.equivalent();}
}
```

6.5 Evaluation

To evaluate the quality of model transformation, it is necessary to find an alignment of quality measures to MT quality, i.e., whether a lower/higher value of a measure improves/worsens MT quality. Previous works [13], [14] have proposed a number of effective quality measures, as they found widespread quality flaws in MT specifications based on investigating the characteristics of technical debt in model transformations, analysing a range of MT cases in different MT languages. Quality measures defined in previous work [13] will be evaluated on the generated code, these measures include:

ETS: Excessive transformation size ($c(\tau) > 1000$, or length > 500 LOC), where c

is the syntactic complexity measure defined in previous work [13].

ENR: Excessive number of rules ($nrules > 10$).

ENO: Excessive number of helpers/operations ($nops > 10$).

ERS: Excessive rule size ($c(r) > 100$ or length greater than 50 LOC).

EHS: Excessive helper size ($c(h) > 100$ or length > 50 LOC).

EPL: Excessive parameter list (for transformations, rules, and helpers): > 10 parameters including auxiliary rule/operation variables.

EFO: Excessive fan-out of a rule/operation (> 5 different rules/operations called from one rule/operation).

EFI: Excessive fan-in of a rule/operation (> 5 different rules/operations call one rule/operation).

CC: Excessive cyclomatic complexity (of rule logic or of procedural code), taken as 1 + the number of elementary Boolean conditions in the control flow/logic of the specification (> 10).

CBR_1: Excessive coupling between rules (the number of rule/operation explicit or implicit calling relations is greater than $nrules + nops$).

CBR_2: Excessive self/mutual dependency between rules (the number of rules/ operations involved in cycles of dependencies in the rule/operation call graph).

DC: Duplicate expressions/code (duplicate expressions or statements x with token count $t(x) > 10$).

UEX: Excessive use of undefined execution orders/priorities between rules (> 10 undefined orderings).

To compute the number of flaws in a transformation, we count 1 for each of ETS, ENR, ENO, UEX, CBR1 over the thresholds, plus ERS + EHS + CC + EPL + EFO + DC + CBR2 [13].

6.5.1 Generating ATL Transformations

We evaluated the approach using several cases from the ATL transformation zoo [22]. The procedure detailed in Chapter 6 was followed with interactive enhancement of the initially derived metamodel correspondences. In Table 6.4 we compare our solutions to the manual solutions of the zoo cases, in terms of their size (LOC) and the number of technical debt flaws in the transformation according to the quality flaw categories [13]. In terms of quality, the synthesised transformations have lower numbers of flaws, and lower flaw density (the average flaw density of the generated transformation versions is 0.008 flaws/LOC, compared to 0.0172 for the original versions).

Table 6.5 shows some estimates of the relative amount of work involved in the manual and automated construction of the ATL zoo case versions. We estimate effort in terms of how many manual changes are necessary to the automatically-synthesised

Table 6.4: Evaluation on ATL zoo cases [22]

Case	Original size (LOC)	F-measure flaws	New size (LOC)	New flaws
Ports	31	0	28	0
PetriNet2PathExp	70	1	27	0
Class2Relational	97	2	35	0
PathExp2PetriNet	104	0	40	0
SimpleClass2 SimpleRDB	302	10	38	0
Ant2Maven	324	4	317	2
Maven2Ant	360	3	332	2
MOF2UML	585	8	255	3
MySQL2KM3	613	13	48	1
UML2MOF	935	18	220	3

versions (additional to changes identified by the interactive improvement process), and the size of the original versions, as numbers of class and feature mappings. The execution time of the automated synthesis is also shown (times are the same for a transformation and its inverse because our tool generates these together).

These results show that a relatively small amount (less than 10%) of transformation content needs to be manually modified or created, for the versions produced by our transformation synthesis process.

6.5.2 Generating ETL Transformations

We evaluated the approach using several published ETL cases [21]. The procedure detailed in Section 6 was followed with interactive enhancement of the initially derived metamodel correspondences. In Table 6.6 we compare transformations gen-

Table 6.5: Effort of manual/automated versions of ATL cases [22]

Case	Execution time (ms)	Changed/ deleted feature maps	Changed/ deleted class maps	Original feature maps	Original class maps
Ports	70	0	0	4	3
PetriNet2PathExp	133	0	0	8	3
Class2Relational	60	0	0	22	6
PathExp2PetriNet	133	0	0	16	3
SimpleClass2 SimpleRDB	285	2	0	12	4
Ant2Maven	326,769	1	0	99	37
Maven2Ant	326,769	1	6	82	30
MOF2UML	66,953	30	2	173	11
MySQL2KM3	859	3	2	117	11
UML2MOF	66,953	30	1	152	13

erated by our approach to the manual transformations of the cases, in terms of their LOC and the number of technical debt flaws in the transformation according to the quality flaw categories and data of previous work [13]. On average, code size has been reduced by 27%, and the number of flaws reduced by a factor of 3.9.

Flaw density has been reduced from 0.108 flaws/LOC to 0.038 (a improvement factor of 2.8). Our solutions for copy transformations such as CopyTVApp and CopyOO have significantly fewer flaws than the original versions because we utilise rule inheritance, thus avoiding duplicated subclass rule code mapping superclass features. The number of rule-rule dependences has also in general been reduced to the minimum necessary for type-correct element mapping. The average f-measure score of our generated solutions is 0.883, indicating high accuracy in recapturing the

original intent of the transformation cases.

Table 6.6: Evaluation on ETL cases [21]

Case	Original size (LOC)	Original flaws [13]	New size (LOC)	New flaws
Tree2Graph	15	1	12	2
Competition2TVApp	30	1	22	0
Flowchart2Html	31	1	25	0
CopyTVApp	48	14	67	2
CopyFlowchart	57	7	48	3
RSS2Atom	88	6	70	2
ArgoUML2Ecore	96	13	55	1
CopyOO	110	23	127	6
OO2DB	142	6	80	3
uml2Simulink	148	11	52	2
Average	76.5	8.3	55.8	2.1

Table 6.7 shows some estimates of the relative amount of work involved in the manual and automated construction of the ETL case versions. We estimate effort in terms of how many manual changes are necessary to the automatically-synthesised versions (additional to changes identified by the interactive improvement process), and the size of the original versions, as numbers of class and feature mappings. The execution time of the automated matching synthesis is also shown.

These results show that a relatively small amount (6.5% with respect to the original feature mappings and 8% with respect to original class mappings) of transformation content needs to be manually modified, for the case versions produced by our transformation synthesis process. Execution time is also within a practical range, for the relatively small cases considered here.

Table 6.7: Effort of manual versus automated versions of ETL cases [21]

Case	Execution time (ms)	Changed/ deleted feature maps	Changed/ deleted class maps	Original feature maps	Original class maps
Tree2Graph	157	0	0	3	1
Competition2TVApp	219	0	0	5	4
Flowchart2Html	258	0	0	4	4
CopyTVApp	156	0	0	13	7
CopyFlowchart	115	0	0	16	5
RSS2Atom	10,667	0	1	10	9
ArgoUML2Ecore	199,214	5	3	20	7
CopyOO	410	0	0	49	10
OO2DB	3,494	5	1	39	10
uml2Simulink	290	1	0	9	5

6.6 Conclusion

We have described a process for synthesizing ATL and ETL transformations from metamodel correspondences, based on an analysis of the consistency and completeness of these correspondences. The correspondences are formalised through an intermediary language denoted as $\mathcal{TL}$. The synthesis approach is novel in attempting to formally emulate the processes which a software engineer would informally undertake when creating a transformation. In addition, it is definetly worth to exlpore inverse transformations from tranformations to $\mathcal{TL}$, however implementing inverse transformations would involve substantial new work which is outside of the main scope of the thesis. Nonetheless, we will try to implement this in the future work.

Chapter 7

Model Transformation by Examples Approach

7.1 Introduction

With the rapid development of machine learning, more and more problems can be solved by this advanced technology. Machine learning is a process that learns from multiple 'input-output' pairs to identify patterns. for example, machine translation. Models are defined by metamodels, and there are correspondences between metamodels which also define model transformations. Therefore, there may be mapping relationships between the source model and the target model, and these mapping relationships include source class to the target class mappings (CM) and source feature to target feature mapping (FM).

Model transformation by-example (MTBE) is an approach for the automated or semi-automated construction of transformations based on examples of expected input and output models. We found that this is also in line with the idea of machine learning, that is, when suitable examples are available, a machine learning model can be constructed with the source models as input and the target models as output, and the machine learning model can be trained so that it can implement automated model transformation.

For homogeneous models, mappings can be identified by similar terminology easily. However, for two models that are heterogeneous, it is difficult to identify mappings by terminology. Although some mappings could be identified by source and target features having the same values, some feature values in the model transformation will also change during the transformation.

In the literature, there is only one related research using the machine learning technique for MTBE [19], however, their approach is based on LSTM and is completely black-box without any interpretability and does not support feature value transformation.

In this study, we propose a framework for automating model transformation. Our framework includes three parts, the first part is a parser, which is used to pre-process the model, and the second part is a class and feature transformation model, which is for transforming the source class/feature (SC/SF) to the target class/feature (TC/TF). The third part is a set of feature value transformation

models for identifying different feature value transformation. Decision tree and multiple linear regression analysis techniques are mainly applied in this framework.

7.2 Background

7.2.1 Regression Analysis

Regression analysis examines the relationship between the dependent variable and the independent variable. This technique is commonly used in predictive analysis. Regression analysis is divided into simple regression analysis and multiple regression analysis according to the number of variables involved; and linear regression analysis and non-linear regression analysis according to the type of relationship between the independent and dependent variables.

In this study, we use the elements of the source model as independent variables and the elements of the target model as dependent variables. Since each model has multiple elements, the regression analysis of model transformation is a kind of multiple regression analysis.

Multiple linear regression is a type of regression analysis which mathematically models the independent and dependent variables as linear equations like:

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \ldots + \beta_p X_{ip} \qquad i = 1, \ldots, n$$

where Y_i is dependent variable, X_i is independent variable, and β_i is coefficient.

In the model transformation, there is often a linear model, e.g. $father_{age} = age+25$. We have also tried to use neural networks to regress non-linear relationships between the feature values, however the results were poor, due to the complexity of the non-linear relationship and the fact that there are few relevant examples.

7.2.2 Decision Tree

Decision tree is a machine learning method. A decision tree is a tree-structure prediction model that uses layers of reasoning to achieve the final outcome prediction. The decision tree is made up of the following elements: Root node: contains the full set of examples. Internal nodes: corresponding to the feature analysis. Leaf nodes: represent the outcome of the decision.

When predicting, a judgement is made at an internal node of the tree with a certain attribute value, and depending on the result of the judgement, a decision is made as to which branch node to enter until the leaf node is reached and the prediction result is obtained. It is therefore a supervised learning algorithm based on 'if-then-else' rules, and these rules for the decision tree are obtained through training, rather than being formulated manually.

The advantages of decision tree method include the ease of extracting rules and the rule extraction process is interpretable and can be analysed visually. The decision tree model structure is simple and relatively fast to execute, producing

feasible and effective results in a short time for large-scale data. All of this is in line with our expectations for MTBE.

7.2.3 Bag-of-Words Model

Machine learning cannot directly process raw text; it needs to be transformed into a numerical vector representation. The Bag-of-Words (Bow) model is a method of extracting features from text by transforming it into a fixed-dimensional vector. It does not consider the order of words in a sentence, but only the number of occurrences of words in the corpus in that sentence. The dimensions of the transformed vector are the total number of different words in the corpus, where the value of each dimension represents the number of occurrences of a word in the sentence. The disadvantage of this model is that when the corpus is large, the dimension of the vector becomes high as well. And because it does not take contextual relationships into account, it leads to a lack of word association and location information. However, for model transformation, proper partitioning of the model elements can effectively control the size of the vocabulary list. Also, the same-level elements in the model are unordered, which would instead reduce the quality of feature extraction if a sequential correlation feature extraction method is used. Therefore, the BoW model is suitable to be used as a model for feature extraction.

7.3 Methodology

We divide the model transformation by example process into two steps. The first step is the class and feature transformation, followed by value transformation after the mapping relationships between source features (SF) and target features (TF) obtained.

7.3.1 Class and Feature Transformation

The model transformation is mainly based on mapping of features, although classes may be related by one-to-one, one-to-many, many-to-one, many-to-many transformations, this is also because the SF in different source classes (SC) are mapped to the TF of different target classes (TC).

Therefore, in this study, we define the basic unit of transformation as a feature, and to distinguish between features in different classes, we combine the class with each of its features to form unique transformation elements:

$$SourceElement = SC_{i}\_f_{k}$$

$$TargetElement = TC_{j}\_f_{k}$$

where $i \in \{1, 2, ..., m\}$, $j \in \{1, 2, ..., n\}$, $k \in \{1, 2, ..., q\}$, n and m are the number of SC and TC in each model respectively, and q is the number of features in each

class.

Due to the existence of mapping relationships, when a *SourceElement* appears in the source model, its corresponding *TargetElement* must be in the target model. Therefore, after BoW counting the number of occurrences of each class and feature, the mapping relationship between *SourceElement* and *TargetElement* can be concluded according to the regression analysis by decision tree model for class and feature transformation. The decision tree model can identify one-to-one, one-to-many, many-to-one, many-to-many relationships very well.

7.3.2 Value Transformation

For each *SourceElement* and *TargetElement*, there is a feature value. Feature values can be of various types, such as String, Boolean, int, etc. In addition, there are different feature value transformation modes for different types.

The most common feature value transformation is a direct transformation, which means that the values remain the same before and after the transformation, eg:

$$SC_{1\_}f_1 = X \longmapsto TC_{1\_}f_1 = X$$

For Boolean type values, different Boolean values may result in different TC and TF being mapped to, eg:

$$SC_{2\_}f_1 = True \longmapsto TC_{2\_}f_1 =''$$

$$SC_{2\_}f_1 = False \longmapsto TC_{3\_}f_1 = ''$$

For numerical values, the mathematical transformation always appears, eg:

$$SC_{4\_}f_1 = 1 \longmapsto TC_{4\_}f_1 = 2$$

where the transformation could be either by multiplying the SF value by two or possibly by adding 1 to the *SF* value, or by other mathematical functions. Therefore, more examples are needed to figure out what the exact mathematical function is.

For the transformation of String values, in addition to direct mapping, the mapping of string transformations, such as prefixing, also always appears, eg:

$$SC_{5\_}f_1 = good \longmapsto TC_{5\_}f_1 = verygood$$

where a prefix 'very' is added during the transformation.

The scope of our approach is restricted to the direct transformation of all value types. In addition, for the mathematical transformation, multiple linear transformation can be performed. We have experimented with techniques such as neural networks to implement non-linear model transformations, but as the non-linear relationships are too complex to be generalisable even for neural networks, our study only focuses on multiple linear model transformation. For String value transformation, prefixing, suffixing, combination and decomposition have been implemented.

A separate decision tree model is constructed for each Source Element, which can easily learn direct transformation. Since there are only two types of Boolean value, *True* and *False*, we connected them directly with features as two separate features, thus turning the value relationship of Boolean into a direct transformation. For example, the decision tree can generate two rules: if '$SC_{2\_}f_{1\,True}$', then '$TC_{2\_}f_1$'; if '$SC_{2\_}f_{1False}$', then '$TC_{3\_}f_1$'.

For String value transformation, a substitution value δ was introduced, which replaces the same part of the source String and the target String. For example, $SC_{5\_}f_1 = good \longmapsto TC_{5\_}f_1 = verygood$ would be replaced by $SC_{5\_}f_1 = \delta \longmapsto TC_{5\_}f_1 = very\delta$, then the decision model can generate a rule that: if '$SC_{5\_}f_1$' and 'δ', then '$TC_{5\_}f_1$' and '$very\delta$'.

Instead of using a decision tree, the mathematical transformation directly employs multiple linear regression. For example, for two models:

$$SC_{6\_}f_1 = 1, SC_{6\_}f_2 = 5 \longmapsto TC_{6\_}f_1 = 23$$

$$SC_{6\_}f_1 = 3, SC_{6\_}f_2 = 11 \longmapsto TC_{6\_}f_1 = 53$$

the multiple linear regression can generate a rule: if '$SC_{6\_}f_1 = a, SC_{6\_}f_2 = b$', then '$TC_{6\_}f_1 = 3a + 4b$'.

7.4 Framework

7.4.1 Framework Overview

Figure. 7.1 shows the overview of our proposed framework. *SourceModelParser* is used for transforming model file to the *SourceElement* form we defined earlier. It also transforms the source model to a *SourceElement* : *Values* version. The *Classandfeaturetransformationmodel* identifies the mapping relationships between SE and TE. An individual *Featurevaluetransformationmodel* is constructed and trained for each SE. Each *SourceElement* : *Values* is input into *Featurevaluetransformationmodels* one by one, then the corresponding *TargetElement* : *Values* is output to revert to the target model.

7.4.2 Training and Validation

We constructed the dataset with pairs of source model and target model examples. 80% of the dataset was used for training and the remaining 20% for validation. During the training process, the target model examples were presented in the same processing sequence as the source model and used as output to train the individual models in the framework. After the models were trained, the source models from the validation set are fed into the framework, and the output of the framework is compared with the target models in the validation set to check the accuracy of the model transformation.

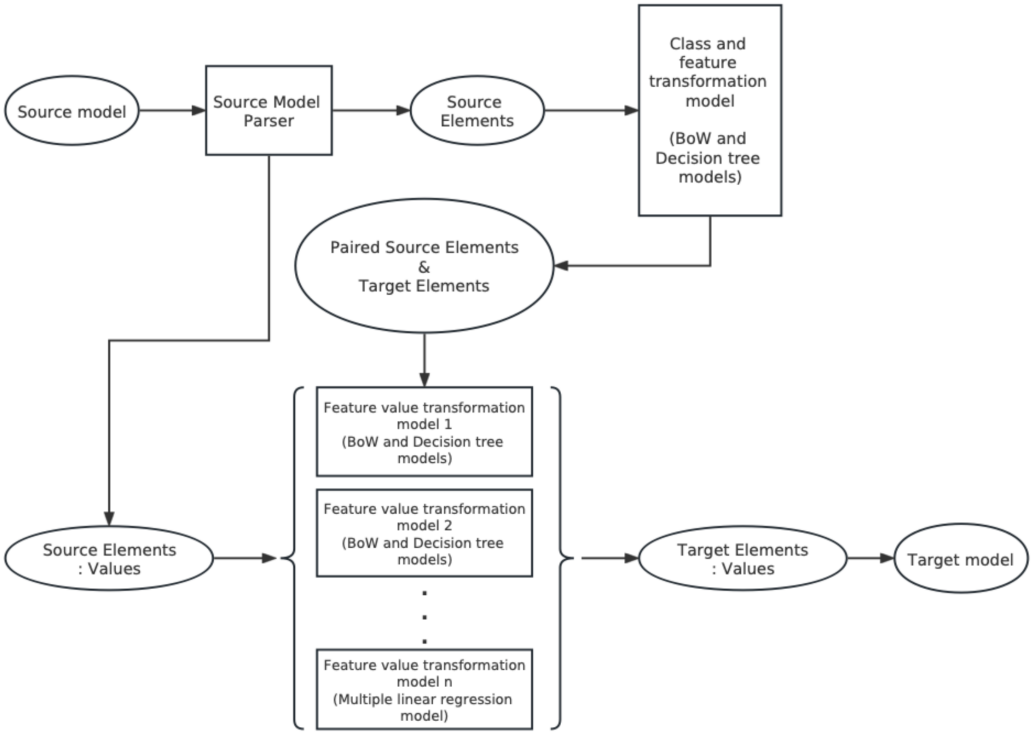


Figure 7.1: MTBE framework overview

7.5 Evaluation

We validated our MTBE approach on different sizes of model transformation examples of the benchmark cases [18]. We only considered the types of feature value transformations that can be recognised in this approach. With the number of models in the training set changes, the average accuracy of the validation set and the length of training time changes as shown in the Figure 7.2 and Figure 7.3. The state-of-the-art approach using machine learning for MTBE [19] requires around

800 examples to reach 100% accuracy, while our approach requires fewer examples to reach this. In addition, the training time of our approach is significantly reduced.

To avoid the discrepancy caused by different model size, we calculated the average transformation time for different numbers of *SourceElement*2*TargetElement* (Figure 7.4).

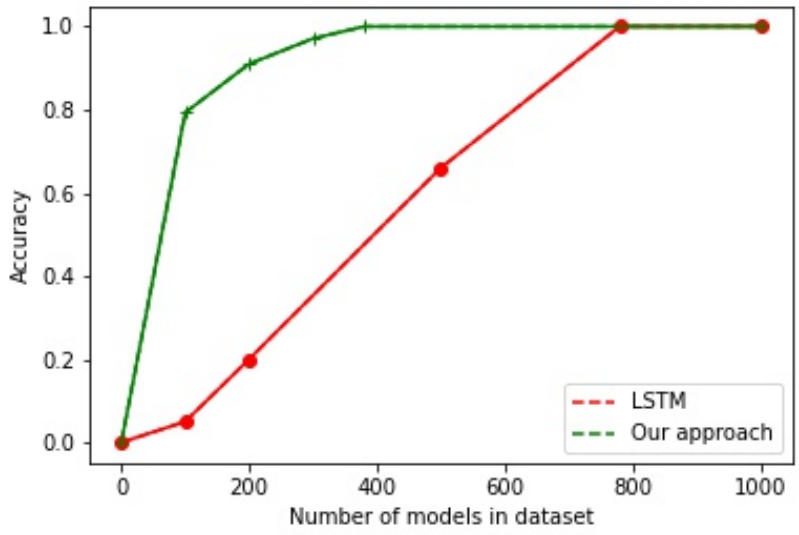


Figure 7.2: Variation of accuracy during training

Overall for $RQ4$, we can conclude that after training with sufficient examples, the machine learning framework for MTBE in this approach can not only provide model transformation with high accuracy, but is also quite fast in training and transformation. Compared to the state-of-the-art approach [19], our approach outperforms in terms of the number of required examples, training time and transformation time.

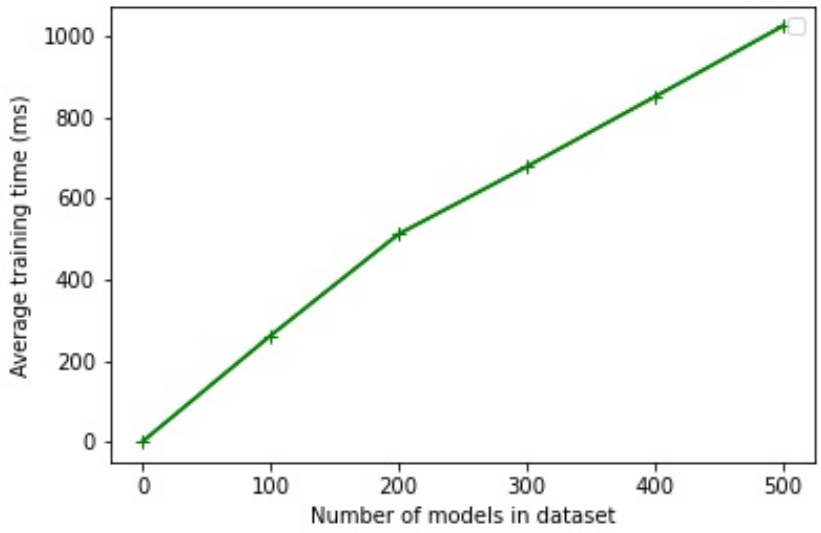


Figure 7.3: Training time for different number of models

7.6 Conclusion

In this chapter, we proposed a novel framework for model transformation by examples using machine learning technique. Decision tree and multiple linear regression analysis techniques are mainly applied in this framework. This approach can handle feature value transformations including direct mapping, linear numeric mapping and changeable String mapping. The introduction of decision trees ensures that all mappings that occur in examples can be correctly transformed, but brings the disadvantage that it is difficult to predict the correct result for types values that do not occur in examples. When the variability of the examples is obvious and most of the mapping relationships are covered, a very high accuracy can be achieved with only a few examples. However, when the variability of the examples is not obvious

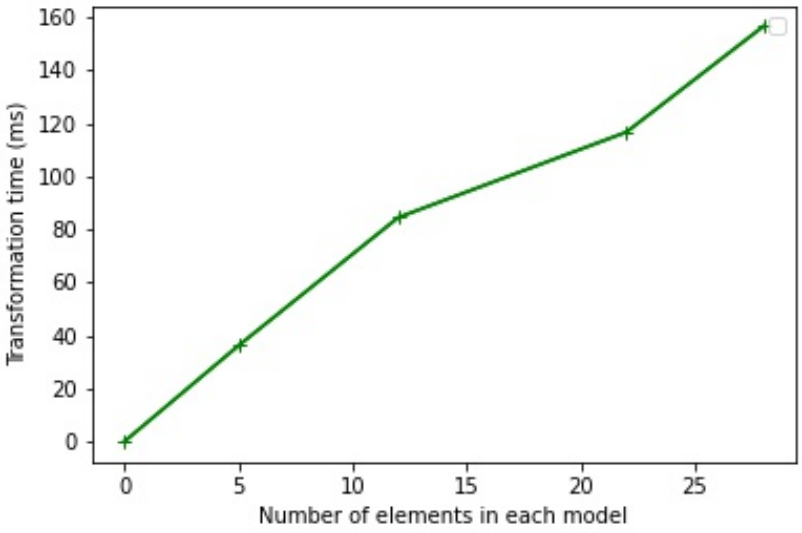


Figure 7.4: Impact of the size of the models when transforming

enough, a large number of examples are needed to achieve a high transformation accuracy.

Chapter 8

Conclusion and Future work

8.1 Conclusion

This thesis has proposed novel approaches to address the challenge of reducing the development time and effort needed to develop model transformation, and improve model transformation quality. To address the issue, this thesis has defined several approaches for synthesis of model transformations from metamodels and examples.

DSS approach for metamodel matching has first been introduced. DSS approach is an exhaustive-search approach, which has handled one-to-one, one-to-many and many-to-one matchings by calculating data structure similarity between source and target metamodel elements. However, the proposed DSS approach cannot handle n-m matchings yet. In addition, there is a scalability problem for large metamodels. These are due to a lot of calculations, and thus increase the matching time.

These disadvantages are unavoidable with the adaptation of the exhaustive-search approach.

To solve the disadvantages, we proposed another metamodel matching approach based on search-based approach. To be more specific, the approach has combined single-objective optimisation or multi-objective optimisation and machine learning (MLP) techniques. The approach has handles one-to-one, one-to-many, many-to-one, many-to-many matchings, which allows it to be applied to various metamodel matching. This approach also addresses the scalability limitation which cannot be addressed by DSS approach.

After extracting correspondences from metamodels using metamodel matching, we have investigated how to synthesise model transformations from correspondences automatically. We have introduced an intermediate language $\mathcal{TL}$ which uses a simplified transformation notation to express transformation specifications in a language-independent manner. We have described a process for synthesising ATL and ETL transformations from the correspondences formalised by $\mathcal{TL}$. The evaluation has shown that our approach can generate different kinds of out-place transformations, including migration, evolution, refinement, abstraction and semantic mapping automatically.

With the development of machine learning, more and more problems can be solved by this intelligent technology. We have combined model transformation problem with machine learning in our research. A novel framework for model transforma-

tion by examples using machine learning technique has been proposed in this thesis. Decision tree and multiple linear regression analysis techniques have been mainly applied in this framework. This approach can handle feature value transformations including direct mapping, linear numeric mapping and changeable String mapping.

Compared with related works in Chapter 3, the approaches in this thesis significantly distinguish themselves by conscientiously addressing the inherent shortcomings. Our DSS approach is the only approach to prioritise DSS. The search-based optimisation approach introduced in this thesis pioneers the adoption of multi-objective optimisation for metamodel matching. Our model transformation synthesis approach is the only one which includes consistency and completeness checks of matchings using correspondence patterns. The MTBE approach in this thesis is the first to implement precise feature value transformations using machine learning techniques. The evaluation results have shown that the approaches proposed in this thesis can significantly reduce the development time and effort needed to develop model transformations. Moreover, compared to manually constructed transformations, the quality of the automated synthesised transformations with our approaches has been improved.

8.2 Future Work

In terms of metamodel matching approaches, although our multi-objective optimisation approach has addressed the shortcomings of DSS approach, multi-objective

optimisation approach still needs to be improved. The existing multi-objective optimisation approach only employs three similarity measures, and more similarity measures are needed. We have trained an MLP model for selecting the best solution from Pareto optimal solution set, however, due to the limited available resources, the number of data for training MLP model and Word2Vec model is still not large enough. In addition, the Word2Vec model used in this study was trained on the Google News dataset.

Future work will continue to focus on using multi-objective optimisation approach for extracting correspondences from large-scale metamodels. More objectives, including graph-related similarity, will be introduced in the future. We will also explore more multi-objective optimisation algorithms, such as MOPSO [137], DMOPSO [138], SMPSO [139], MOEA/D [122], SPEA2[140], PESA2 [141], to improve the efficiency of the search. More metamodels will be mined and used to train the MLP model for selecting the best solution from Pareto optimal solution set. In the future, we will investigate on building a Word2Vec model exclusively for MDE to obtain more accurate NMS.

Both approaches for metamodel matching have a limitation that is precise mapping of feature values cannot be identified: different possible mappings of one integer value to another are not distinguished, also mappings of one string value to another are not distinguished. Only direct value mapping can be performed. Although the MTBE approach proposed in this thesis has addressed this partially, there are still

several kinds of feature value mappings that need to be addressed. In addition, the approach still requires a large number of examples for machine learning.

In the future, we will also continue to investigate on precise mapping of feature values. In addition to the existing Numeric, String and Boolean types, we will explore more feature value types, including Collections, Enumerations, etc. We will also explore how to accomplish precise feature value transformations with fewer examples in the future. More artificial intelligence techniques, such as Convolutional Neural Network (CNN) [142], Transformer-based models [143], [144], Natural Language Processing (NLP) [145], [146], will be used to facilitate better learning of these complex feature value transformations.

Currently, we have only implemented transformation from $\mathcal{TL}$ to MT in different languages. In the future we will implement reverse tranformation from MT to $\mathcal{TL}$, thus providing more transformation possibilities.

After improving the multi-objective optimisation approach and MTBE approach, we will fuse these approaches into a formal model transformation synthesis approach. We expect that given a source metamodel, a target metamodel, and a small number of examples, the fusion approach will automatically generate high-quality model transformations with precise feature value mappings.

Bibliography

[1] S. Kent, "Model driven engineering," in *International Conference on Integrated Formal Methods*, Springer, 2002, pp. 286–298.

[2] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[3] X. Hei, L. Chang, W. Ma, J. Gao, and G. Xie, "Automatic transformation from uml statechart to petri nets for safety analysis and verification," in *2011 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering*, IEEE, 2011, pp. 948–951.

[4] M. Wang and L. Lu, "A transformation method from uml statechartto petri nets," in *2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, IEEE, vol. 2, 2012, pp. 89–92.

[5] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 396–414, 2016.

[6] L. Burgueño, J. Cabot, and S. Gérard, "The future of model transformation languages: An open community," *Journal of Object Technology*, vol. 18, no. 3, 2019.

[7] K. Lano, Q. Xue, and S. Kolahdouz-Rahimi, "Agile specification of code generators for model-driven engineering," *Proceedings of ICSEA 2020*, pp. 9–15, 2020.

[8] K. Lano, S. Kolahdouz-Rahimi, and S. Fang, "Model transformation development using automated requirements analysis, metamodel matching, and transformation by example," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–71, 2021.

[9] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin, "Managing model adaptation by precise detection of metamodel changes," in *European Conference on Model Driven Architecture-Foundations and Applications*, Springer, 2009, pp. 34–49.

[10] M. D. Del Fabro and P. Valduriez, "Towards the efficient development of model transformations using model weaving and matching transformations," *Software & Systems Modeling*, vol. 8, no. 3, pp. 305–324, 2009.

[11] S. Schwichtenberg, C. Gerth, Z. Huma, and G. Engels, "Normalizing heterogeneous service description models with generated qvt transformations," in

European Conference on Modelling Foundations and Applications, Springer, 2014, pp. 180–195.

[12] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Cope-automating coupled evolution of metamodels and models," in *European Conference on Object-Oriented Programming*, Springer, 2009, pp. 52–76.

[13] K. Lano, S. Kolahdouz-Rahimi, M. Sharbaf, and H. Alfraihi, "Technical debt in model transformation specifications," in *International Conference on Theory and Practice of Model Transformations*, Springer, 2018, pp. 127–141.

[14] M. Wimmer, S. M. Perez, F. Jouault, and J. Cabot, "A catalogue of refactorings for model-to-model transformations.," *Journal of Object Technology*, vol. 11, no. 2, pp. 2–1, 2012.

[15] D. Lopes, S. Hammoudi, J. De Souza, and A. Bontempo, "Metamodel matching: Experiments and comparison," in *2006 International Conference on Software Engineering Advances (ICSEA'06)*, IEEE, 2006, pp. 2–2.

[16] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut, "Metamodel matching for automatic model transformation generation," in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2008, pp. 326–340.

[17] L. Lafi, S. Hammoudi, and J. Feki, "Metamodel matching techniques in mda: Challenge, issues and comparison," in *Model and Data Engineering: First*

International Conference, MEDI 2011, Óbidos, Portugal, September 28-30, 2011. Proceedings 1, Springer, 2011, pp. 278–286.

[18] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.

[19] L. Burgueño, J. Cabot, S. Li, and S. Gérard, "A generic lstm neural network architecture to infer heterogeneous model transformations," *Software and Systems Modeling*, vol. 21, no. 1, pp. 139–156, 2022.

[20] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Towards an expressivity benchmark for mappings based on a systematic classification of heterogeneities," in *Proceedings of the First International Workshop on Model-Driven Interoperability*, 2010, pp. 32–41.

[21] Epsilon, Website, `https://www.eclipse.org/epsilon/examples/`, 2020.

[22] EclipseATLZoo, Website, `www.eclipse.org/atl/atlTransformations`, 2020.

[23] L. Addazi, A. Cicchetti, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Semantic-based model matching with emfcompare," in *ME@ MODELS*, 2016, pp. 40–49.

[24] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[25] S. R. Schach, *Software Engineering*. Aksen associates, 1990.

[26] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003.

[27] D. S. Frankel, "Model driven architecture: Applying mda to enterprise computing," *Google Scholar Google Scholar Digital Library Digital Library*, 2003.

[28] A. Kleppe, J. Warmer, and W. Bast, "Mda explained: The model driven architecture (tm): Practice and promise," *Addison-Wesley Professional. Retrieved November*, vol. 5, p. 2010, 2003.

[29] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Software Engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[30] C. Atkinson and T. Kuhne, "Model-driven development: A metamodeling foundation," *IEEE Software*, vol. 20, no. 5, pp. 36–41, 2003.

[31] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering (FOSE'07)*, IEEE, 2007, pp. 37–54.

[32] O. Pastor, S. España, J. I. Panach, and N. Aquino, "Model-driven development," *Informatik-Spektrum*, vol. 31, no. 5, pp. 394–407, 2008.

[33] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Systems Journal*, vol. 45, no. 3, pp. 451–461, 2006.

[34] R. Soley *et al.*, "Model driven architecture," *OMG White Paper*, vol. 308, no. 308, p. 5, 2000.

[35] A. W. Brown, "Model driven architecture: Principles and practice," *Software and Systems Modeling*, vol. 3, no. 4, pp. 314–327, 2004.

[36] F. N. Place, "Object management group," *Mars*, vol. 2005, pp. 06–12, 2000.

[37] E. Seidewitz, "What models mean," *IEEE Software*, vol. 20, no. 5, pp. 26–32, 2003.

[38] A. M. Starfield, K. A. Smith, and A. L. Bleloch, *How to model it: Problem solving for the computer age*. Interaction Book Company, 1994.

[39] J. Bézivin and O. Gerbé, "Towards a precise definition of the omg/mda framework," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, IEEE, 2001, pp. 273–280.

[40] K. Lano, *Model-driven software development with UML and Java*. Course Technology Press, 2009.

[41] R. Pooley and P. King, "The unified modelling language and performance engineering," *IEE Proceedings-Software*, vol. 146, no. 1, pp. 2–10, 1999.

[42] K. Lano, *The B language and method: a guide to practical formal development*. Springer Science & Business Media, 2012.

[43] J. M. Spivey and J. Abrial, *The Z notation*. Prentice Hall Hemel Hempstead, 1992, vol. 29.

[44] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[45] P. Hudak, "Domain-specific languages," *Handbook of Programming Languages*, vol. 3, no. 39-60, p. 21, 1997.

[46] B. Henderson-Sellers, C. Gonzalez-Perez, O. Eriksson, P. J. Ågerfalk, and G. Walkerden, "Software modelling languages: A wish list," in *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*, IEEE, 2015, pp. 72–77.

[47] M. Richters and M. Gogolla, "On formalizing the uml object constraint language ocl," in *International Conference on Conceptual Modeling*, Springer, 1998, pp. 449–464.

[48] J. Cabot and M. Gogolla, "Object constraint language (ocl): A definitive guide," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, Springer, 2012, pp. 58–90.

[49] OMG, *Unified modeling language*, 2001.

[50] OMG, *Object constraint language*, Website, `https://www.omg.org/spec/OCL`, 2014.

[51] E. Cariou, C. Ballagny, A. Feugas, and F. Barbier, "Contracts for model execution verification," in *European Conference on Modelling Foundations and Applications*, Springer, 2011, pp. 3–18.

[52] J.-M. Favre, "Towards a basic theory to model model driven engineering," in *3rd Workshop in Software Model Engineering, WiSME*, Citeseer, 2004, pp. 262–271.

[53] R. F. Paige, P. J. Brooke, and J. S. Ostroff, "Metamodel-based model conformance and multiview consistency checking," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 3, 11–es, 2007.

[54] J. Overbeek, "Meta object facility (mof): Investigation of the state of the art," M.S. thesis, University of Twente, 2006.

[55] I. Poernomo, "The meta-object facility typed," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006, pp. 1845–1849.

[56] H. A. Handley, W. Khallouli, J. Huang, W. Edmonson, and N. Kibret, "Maintaining the consistency of sysml model exports to xml metadata interchange (xmi)," in *2021 IEEE International Systems Conference (SysCon)*, IEEE, 2021, pp. 1–8.

[57] W3C, *Extensible markup language (xml)*, Website, `http://www.w3.org/XML/`, 2016.

[58] OMG, *Mof 2 xmi mapping*, Website, `http://www.omg.org/spec/XMI/2.4.1/`, 2011.

[59] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[60] M. Belaunde, C. Casanave, D. DSouza, *et al.*, *Mda guide version 1.0.1*, 2003.

[61] M. Kempa and Z. A. Mann, "Model driven architecture," *Informatik-Spektrum*, vol. 28, no. 4, pp. 298–302, 2005.

[62] T. Clark, A. Evans, and R. France, "Object-oriented theories for model driven architecture," in *International Conference on Object-Oriented Information Systems*, Springer, 2002, pp. 235–244.

[63] J. Osis, E. Asnina, and A. Grave, "Computation independent modeling within the mda," in *IEEE International Conference on Software-Science, Technology & Engineering (SwSTE'07)*, IEEE, 2007, pp. 22–34.

[64] J. L. Garrido, M. Noguera, M. González, M. V. Hurtado, and M. L. Rodríguez, "Definition and use of computation independent models in an mda-based groupware development process," *Science of Computer Programming*, vol. 66, no. 1, pp. 25–43, 2007.

[65] G. Benguria, X. Larrucea, B. Elvesæter, T. Neple, A. Beardsmore, and M. Friess, "A platform independent model for service oriented architectures," in *Enterprise Interoperability*, Springer, 2007, pp. 23–32.

[66] J. Criado, L. Iribarne, and N. Padilla, "Resolving platform specific models at runtime using an mde-based trading approach," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, Springer, 2013, pp. 274–283.

[67] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[68] R.-J. Back and J. Wright, *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.

[69] N. Wirth, "Program development by stepwise refinement," in *Pioneers and Their Contributions to Software Engineering*, Springer, 2001, pp. 545–569.

[70] G. Hinkel, "An approach to maintainable model transformations with an internal dsl," Ph.D. dissertation, PhD thesis. National Research Center, 2013.

[71] S. Götz, M. Tichy, and R. Groner, "Claimed advantages and disadvantages of (dedicated) model transformation languages: A systematic literature review," *Software and Systems Modeling*, vol. 20, no. 2, pp. 469–503, 2021.

[72] L. Tratt, "Model transformations and tool integration," *Software & Systems Modeling*, vol. 4, no. 2, pp. 112–122, 2005.

[73] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[74] OMG, *Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification*, `http://www.omg.org/spec/QVT/1.1`, Last accessed 17 Feb 2019.

[75] K. Lano and S. Kolahdouz-Rahimi, "Specification and verification of model transformations using uml-rsds," in *International Conference on Integrated Formal Methods*, Springer, 2010, pp. 199–214.

[76] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "ATL: a QVT-like transformation language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ACM, 2006, pp. 719–720.

[77] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[78] F. Jouault and W. Piers, *Atl user guide*, `https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language`, 2009.

[79] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," in *International Conference on Theory and Practice of Model Transformations*, Springer, 2008, pp. 46–60.

[80] D. Kolovos, L. Rose, R. Paige, and A. Garcıa-Domınguez, "The epsilon book," *Structure*, vol. 178, pp. 1–10, 2010.

[81] L. Lúcio, M. Amrani, J. Dingel, *et al.*, "Model transformation intents and their properties," *Software & Systems Modeling*, vol. 15, no. 3, pp. 647–684, 2016.

[82] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *Proceedings 18th International Conference on Data Engineering*, IEEE, 2002, pp. 117–128.

[83] K. Voigt and T. Heinze, "Metamodel matching based on planar graph edit distance," in *International Conference on Theory and Practice of Model Transformations*, Springer, 2010, pp. 245–259.

[84] G. A. Miller, "Wordnet: A lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[85] P. N. Mendes, M. Jakob, and C. Bizer, "Dbpedia: A multilingual cross-domain knowledge base.," in *LREC*, Citeseer, 2012, pp. 1813–1817.

[86] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," *UPGRADE, The European Journal for the Informatics Professional*, vol. 9, no. 2, pp. 29–34, 2008.

[87] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *the VLDB Journal*, vol. 10, pp. 334–350, 2001.

[88] P. Mitra, G. Wiederhold, and J. Jannink, "Semi-automatic integration of knowledge sources," in *Proceedings of Fusion*, vol. 99, 1999, pp. 1–9.

[89] P. Mitra, G. Wiederhold, and M. Kersten, "A graph-oriented model for articulation of ontology interdependencies," in *International Conference on Extending Database Technology*, Springer, 2000, pp. 86–100.

[90] T. Milo and S. Zohar, "Using schema matching to simplify heterogeneous data translation," in *Vldb*, vol. 98, 1998, pp. 24–27.

[91] L. Palopoli, D. Sacca, and D. Ursino, "An automatic technique for detecting type conflicts in database schemes," in *Proceedings of the seventh international conference on Information and knowledge management*, 1998, pp. 306–313.

[92] L. Palopoli, D. Saccá, G. Terracina, and D. Ursino, "A unified graph-based framework for deriving nominal interscheme properties, type conflicts and object cluster similarities," in *Proceedings Fourth IFCIS International Conference on Cooperative Information Systems. CoopIS 99 (Cat. No. PR00384)*, IEEE, 1999, pp. 34–45.

[93] S. Castano and V. De Antonellis, "Global viewing of heterogeneous data sources," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 2, pp. 277–297, 2001.

[94] D. Beneventano, S. Bergamaschi, S. Castano, *et al.*, "Information integration: The momis project demonstration," in *Vldb*, 2000, pp. 611–614.

[95] J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic schema matching with cupid," in *vldb*, vol. 1, 2001, pp. 49–58.

[96] W.-S. Li and C. Clifton, "Semint: A tool for identifying attribute correspondences in heterogeneous databases using neural networks," *Data & Knowledge Engineering*, vol. 33, no. 1, pp. 49–84, 2000.

[97] W.-S. Li, C. Clifton, and S.-Y. Liu, "Database integration using neural networks: Implementation and experiences," *Knowledge and information systems*, vol. 2, pp. 73–96, 2000.

[98] D. Varró, "Model transformation by example," in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 410–424.

[99] D. Varró and Z. Balogh, "Automating model transformation by example using inductive logic programming," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, 2007, pp. 978–984.

[100] S. Muggleton and L. De Raedt, "Inductive logic programming: Theory and methods," *The Journal of Logic Programming*, vol. 19, pp. 629–679, 1994.

[101] Z. Balogh and D. Varró, "Model transformation by example using inductive logic programming," *Software & Systems Modeling*, vol. 8, no. 3, pp. 347–364, 2009.

[102] S. Gulwani, "Programming by examples," *Dependable Software Systems Engineering*, vol. 45, no. 137, pp. 3–15, 2016.

[103] S. Gulwani, "Programming by examples: Applications, algorithms and ambiguity resolution," in *Principles and Practice of Declarative Programming*, 2017.

[104] L. Burgueño, J. Cabot, and S. Gérard, "An lstm-based neural network architecture for model transformations," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, 2019, pp. 294–299.

[105] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. B. Omar, "Search-based model transformation by example," *Software & Systems Modeling*, vol. 11, no. 2, pp. 209–226, 2012.

[106] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, pp. 65–85, 1994.

[107] D. Beyer, C. Lewerentz, and F. Simon, "Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems," in *International Workshop on Software Measurement*, Springer, 2000, pp. 1–17.

[108] S. Fang and K. Lano, "Extracting correspondences from metamodels using metamodel matching," in *STAF-JRC 2019, ser. CEUR Workshop Proceedings*, vol. 2405, 2019, pp. 3–8.

[109] O. Macindoe and W. Richards, "Graph comparison using fine structure analysis," in *2010 IEEE Second International Conference on Social Computing*, IEEE, 2010, pp. 193–200.

[110] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, 1966, pp. 707–710.

[111] D. Kless and S. Milton, "Comparison of thesauri and ontologies from a semiotic perspective," in *Proc. of the 6th Australian Ontology Workshop (AOW 2010), T. Meyer, MA Orgun and K. Taylor (eds.) Adelaide, AU: Australian Computer Society*, Citeseer, 2010, pp. 35–44.

[112] A. Maedche and S. Staab, *Comparing ontologies-similarity measures and a comparison study*. AIFB, 2001.

[113] A. Corazza, S. Di Martino, and V. Maggio, "Linsen: An efficient approach to split identifiers and expand abbreviations," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2012, pp. 233–242.

[114] G. Kappel, H. Kargl, G. Kramler, *et al.*, "Matching metamodels with semantic systems-an experience report.," in *BTW Workshops*, 2007, pp. 38–52.

[115] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[116] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[117] Y. Censor, "Pareto optimality in multiobjective problems," *Applied Mathematics and Optimization*, vol. 4, no. 1, pp. 41–59, 1977.

[118] R. A. Rutenbar, "Simulated annealing algorithms: An overview," *IEEE Circuits and Devices magazine*, vol. 5, no. 1, pp. 19–26, 1989.

[119] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, IEEE, vol. 4, 1995, pp. 1942–1948.

[120] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, pp. 341–359, 1997.

[121] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2013.

[122] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, 2007.

[123] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[124] Z. Fan, Y. Fang, W. Li, J. Lu, X. Cai, and C. Wei, "A comparative study of constrained multi-objective evolutionary algorithms on constrained multi-objective optimization problems," in *2017 IEEE congress on evolutionary computation (CEC)*, IEEE, 2017, pp. 209–216.

[125] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[126] K. Lano and S. Fang, "Automated synthesis of atl transformations from metamodel correspondences," in *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, SCITEPRESS, 2020, pp. 263–270.

[127] Y. Ou, "On mapping between uml and entity-relationship model," in *The Unified Modeling Language: Technical Aspects and Applications*, Springer, 1998, pp. 45–57.

[128] E. Hancer, B. Xue, M. Zhang, D. Karaboga, and B. Akay, "Pareto front feature selection based on artificial bee colony optimization," *Information Sciences*, vol. 422, pp. 462–479, 2018.

[129] X. Yan and X. G. Su, "Linear regression analysis," *Theory and Computing*, 2003.

[130] N. Cristianini, J. Shawe-Taylor, *et al.*, *An introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[131] D. Wipf and S. Nagarajan, "A new view of automatic relevance determination," *Advances in Neural Information Processing Systems*, vol. 20, 2007.

[132] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric Environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998.

[133] K. Lano, S. Fang, and S. Kolahdouz-Rahimi, "Tl: An abstract specification language for bidirectional transformations," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020, pp. 1–10.

[134] Eclipse, *Atl user guide*, Website, `https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language`, 2019.

[135] N. Bonet, K. Garcés, R. Casallas, M. E. Correal, and R. Wei, "Influence of programming style in transformation bad smells: Mining of etl repositories," *Computer Science Education*, vol. 28, no. 1, pp. 87–108, 2018.

[136] OMG, *Object constraint language specification v2.4*, Website, `https://www.omg.org/spec/OCL/2.4/PDF`, 2014.

[137] C. C. Coello and M. S. Lechuga, "Mopso: A proposal for multiple objective particle swarm optimization," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, IEEE, vol. 2, 2002, pp. 1051–1056.

[138] K.-B. Lee and J.-H. Kim, "Dmopso: Dual multi-objective particle swarm optimization," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2014, pp. 3096–3102.

[139] A. J. Nebro, J. J. Durillo, J. Garcia-Nieto, C. C. Coello, F. Luna, and E. Alba, "Smpso: A new pso-based metaheuristic for multi-objective optimization," in *2009 IEEE Symposium on Computational Intelligence in Multi-criteria Decision-making (MCDM)*, IEEE, 2009, pp. 66–73.

[140] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," *TIK-Report*, vol. 103, 2001.

[141] D. W. Corne, N. R. Jerram, J. D. Knowles, and M. J. Oates, "Pesa-ii: Region-based selection in evolutionary multiobjective optimization," in *Proceedings*

of the 3rd annual conference on genetic and evolutionary computation, 2001, pp. 283–290.

[142] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[143] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[144] F. A. Acheampong, H. Nunoo-Mensah, and W. Chen, "Transformer models for text-based emotion detection: A review of bert-based approaches," *Artificial Intelligence Review*, vol. 54, no. 8, pp. 5789–5829, 2021.

[145] K. Chowdhary, "Natural language processing," *Fundamentals of Artificial Intelligence*, pp. 603–649, 2020.

[146] J. Hirschberg and C. D. Manning, "Advances in natural language processing," *Science*, vol. 349, no. 6245, pp. 261–266, 2015.