Automatic Generation of Trace Links in Model-driven Software Development

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Math. Birgit Grammel
geboren am 27. September 1978 in Pretoria

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)
Prof. Dr. Richard Paige
(University of York, UK)

Tag der Verteidigung: Dresden, den 17. Februar 2014

Dresden im November 2014

Abstract

Traceability data provides the knowledge on dependencies and logical relations existing amongst artefacts that are created during software development. In reasoning over traceability data, conclusions can be drawn to increase the quality of software.

The paradigm of Model-driven Software Engineering (MDSD) promotes the generation of software out of models. The latter are specified through different modelling languages. In subsequent model transformations, these models are used to generate programming code automatically. Traceability data of the involved artefacts in a MDSD process can be used to increase the software quality in providing the necessary knowledge as described above.

Existing traceability solutions in MDSD are based on the integral model mapping of transformation execution to generate traceability data. Yet, these solutions still entail a wide range of open challenges. One challenge is that the collected traceability data does not adhere to a unified formal definition, which leads to poorly integrated traceability data. This aggravates the reasoning over traceability data. Furthermore, these traceability solutions all depend on the existence of a transformation engine. However, not in all cases pertaining to MDSD can a transformation engine be accessed, while taking into account proprietary transformation engines, or manually implemented transformations. In these cases it is not possible to instrument the transformation engine for the sake of generating traceability data, resulting in a lack of traceability data.

In this work, we address these shortcomings. In doing so, we propose a generic traceability framework for augmenting arbitrary transformation approaches with a traceability mechanism. To integrate traceability data from different transformation approaches, our approach features a methodology for augmentation possibilities based on a design pattern. The design pattern supplies the engineer with recommendations for designing the traceability mechanism and for modelling traceability data. Additionally, to provide a traceability mechanism for inaccessible transformation engines, we leverage parallel model matching to generate traceability data for arbitrary source and target models. This approach is based on a language-agnostic concept of three similarity measures for matching. To realise the similarity measures, we exploit metamodel matching techniques for graph-based model matching. Finally, we evaluate our approach according to a set of transformations from an SAP business application and the domain of MDSD.

Acknowledgements

The idea for this thesis arose during my time at SAP Research CEC Dresden. I am grateful to SAP Research for financing this work. This thesis would not have been written without the encouragement of my supervisor, Prof. Uwe Aßmann. Thank you for giving me the opportunity to work on this interesting and challenging topic, for all inspiring ideas and the help to shape mine.

Further, I want to express my gratitude to the colleagues of SAP Research, Dr. Andreas Rummler, Dr. Roger Kilian-Kehr and Henrik Lochmann for all discussions on shaping this thesis and critical comments, Dr. Christoph Pohl, from whom I learnt about hyperedges and Matthias Fritzsche, who provided me with knowledge on ATL. My very special thank you goes out to Konrad Voigt for all the endless discussions, proofreading and more discussions. You have a very sharp and quick mind. I am particular grateful to Stefan Kastenholz for his untiring commitment, while working on his minor thesis and diploma under my supervision. This work has highly profited from all our discussions.

My gratitude goes out to the software technology group at TU Dresden, Jendrik Johannes, Florian Heidenreich, Andreas Bartho, Mirko Seifert, Christian Wende, Julia Schröter and Simone Röttger for all the feedback and fruitful discussions at the Doktorandentage. I am especially grateful to Christoff Bürger for proofreading my thesis and for valuable comments.

I would like to extend a special thanks to the researchers of the AMPLE project, especially Steffen Zschaler and Ralf Mitschke, who all accompanied me through the first three years of my thesis with interesting ideas on traceability.

Last, but not least, I would like to thank my family and friends, who supported me through all of the ups and downs during this time, most surely not monotonous. Sincere thanks go out to my brother for he is an amazing person to be around with lots of positive energy. Without you all I would not have written a single line. Above all, I would like to thank M for accompanying me with so much love during the last two years of this thesis. You are the best.

I thank you all.

Heidelberg, October 2013

Publications

This thesis is based on the following peer-reviewed publications:

- B. Grammel, S. Kastenholz, and K. Voigt. Model Matching for Trace Link Generation in Model-driven Software Development. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MoDELS'12)*, Volume 7590, Springer Verlag, 2012.
- B. Grammel and S. Kastenholz. A Generic Traceability Framework for Facet-Based Traceability Data Extraction in Model-Driven Software Development. In *Proceedings of the Traceability Workshop of the 6th European Conference on Modelling Foundations and Applications (ECMFA-TW'10)*, ACM, 2010.
- B. Grammel and K. Voigt. Foundations for a Generic Traceability Framework in Model-driven Software Engineering. In *Proceedings of the Traceability Workshop of the 5th European Conference on Modelling Foundations and Applications (ECMFA-TW'09)*, 2009.
- B. Grammel. Towards a Generic Traceability Framework for Model-driven Software Engineering. In *Proceedings of the 1st International Workshop on Future Trends of Model-driven Development of the International Conference on Enterprise Information Systems (ICEIS'09)*, 2009.
- N. Anquetil, B. Grammel, I. Galvao, J. Noopen, S. S. Khan, H. Arboleda, A. Rashid, and A. Garcia. Traceability for Model-driven, Software Product Line Engineering. In *Proceedings of the Traceability Workshop of the 4th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-TW'08)*, 2008.
- H. Lochmann and B. Grammel. The Sales Scenario: A Model-driven Software Product Line. In *Proceedings of Software Engineering 2008 Workshopband: Fachtagung des GI-Fachbereichs Softwaretechnik*, 2008.
- C. Pohl, A. Charfi, W. Gilani, S. Göbel, B. Grammel, H. Lochmann, A. Rummler, and A. Spriestersbach. Aspect-Oriented Software Development in Business Application Engineering. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, 2008.
- C. Pohl, A. Rummler, and B. Grammel. Improving Traceability in Model-Driven Development of Business Applications. In *Proceedings of the Traceability Workshop of the 3rd European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-TW'07)*, 2007.

Contents

1

Introduction

In the IEEE Standard Glossary of Software Engineering Terminology [IEE90] the notion of traceability is defined as: *The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.* Traceability data in model-driven software development (MDSD) [SV06] can be understood as the runtime footprint of model transformation execution [CH06]. Essentially, trace links provide this kind of information by associating source and target model elements with respect to the execution of a certain model transformation. Trace links have a manifold application domain [RJ01, CH06]:

- *System comprehension* to understand system complexity by navigating via trace links along model transformation chains
- *Coverage analysis* to determine whether all requirements were covered by test cases in the development life cycle, where the traceability of models and model elements is an integral part
- *Change impact analysis* to analyze the impact of a model change on existing generated output
- *Orphan analysis* to find orphaned model elements with respect to a specified trace link
- *Model transformation debugging* to locate bugs during the development of transformation programs and later in shipped applications

According to [CH06, VBJB07] model transformation approaches either generate trace links *implicitly* or *explicitly*. In other words, in the former case, they provide an integrated traceability solution, or in the latter one, they require traceability-specific encoding for generating traceability data. Nevertheless, these traceability solutions still entail a wide range of open challenges. One challenge is that the collected traceability data does not adhere to a unified traceability metamodel or formal definition, which aggravates tool interoperability and reasoning over traceability data. At the same time, traceability data needs to be sufficiently expressive to account for the above-mentioned traceability scenarios, which is not necessarily compatible with a traceability metamodel claiming universality. Finding the sweet spot between these objectives is a non-trivial task according to the Center of Excellence for Software Traceability [ABE$^+$06, GHZ$^+$11]. Essentially, the *inexpressiveness* as well as the *aggravated standardization and integration* of traceability data result in the *problem of poor quality of traceability data*.

Furthermore, there is yet room for improving the means to populate a traceability metamodel, requiring manual encoding for individual transformations. This is likely to be a cost and time-intensive task, leading to the *problem of error-prone and time-consuming efforts* to achieve traceability.

Regarding the approaches for implicit and explicit generation, these traceability solutions all depend on the existence of a transformation engine. Yet, a transformation engine cannot be accessed in all cases of model-based development, for example, while taking into account proprietary transformation engines, or manually implemented transformations. In these cases, it is not possible to leverage the transformation engine for generating traceability data automatically. This leads to the *problem of lacking traceability data due to non-existing or inaccessible transformation engines.*

In view of these shortcomings, this thesis suggests several methods to generate traceability data automatically. In doing so, we propose a generic traceability framework for augmenting arbitrary model transformation approaches with a traceability mechanism. To tackle the problem of poor quality of traceability data, this framework is based on a traceability metamodel, presenting the formalization on integration conditions needed for implementing a traceability mechanism. Essentially, this metamodel provides a unified traceability metamodel, yet on the other hand accounts for an adequate expressiveness of traceability data needed for the traceability scenarios. To achieve this dual nature, the traceability metamodel is featured with an extensibility mechanism based on facets [Pri00b].

Regarding explicit and implicit generation, we advocate two possible augmentation methods to achieve a traceability mechanism, respectively: a) Augmentation of the transformation-engine logic based on aspect-oriented programming [EFB01] and b) Augmentation of the traceability-data output through the use of a model transformation. Once the augmentation method has been applied in either of the two cases, the transformation approach is featured with a traceability mechanism that generates traceability data conforming to the proposed traceability metamodel. In the case of explicit generation, the resulting traceability mechanism does not require manual effort in order to gain traceability data, as opposed to before.

In order to address the problem of lacking traceability data due to non-existing or inaccessible transformation engines, we propose to use model matching techniques as part of the traceability framework to generate trace links for arbitrary source and target models.

Based on the above-mentioned problem definition and solution approaches, we formulate the following hypotheses, to be validated in this thesis:

H1 The quality of traceability data is improved through:

  - a generic interface for integration of traceability data
  - a generic traceability metamodel
  - a facet-based extensibility mechanism of the traceability metamodel.

H2 Efforts to achieve traceability are minimized (for the explicit generation class) through augmentation of the transformation engine with the means of trace-link generation.

H3 Parallel model matching can be leveraged for the generation of trace links regarding inaccessible or non-existing transformation engines.

In the following, we provide an overview on the contributions of this thesis.

1.1. Thesis Contributions

The overall contribution of this thesis is a generic traceability framework for generating trace links automatically. The framework provides three major components, which each in itself reflect a contribution: A traceability metamodel (C1), a generic traceability interface for arbitrary transformation engines (C2) and a model matching component (C3). These contributions are described in the following.

C1: Facet-based Modelling of Traceability Data with CRUD Trace Links

To account for a sufficient expressiveness of traceability data, the specification of user-defined artefacts and link types in traceability metamodels is necessary. To allow for the extensibility of traceability metamodels, corresponding types are defined as facets. Examples are, a facet for the *life cycle* of artefacts, with faceted values requirement, design, code, or test; the *location*, where artefacts were created and the *stakeholder* of artefacts with values like project manager, or developer. Since facets can be varied independently and re-combined, the extensibility of traceability metamodels is achieved.

Furthermore, we define a minimal set of 4 elementary trace links based on the semantics of CRUD actions from database operations. This set of trace links builds the foundation on defining link types in a standardized way, yet may be extended through the use of facets as described above.

C2: Design Pattern on Augmentation of Model Transformations with Trace-Link Generation

We propose a design pattern and a methodology to augment arbitrary model transformations with a specific traceability mechanism. The design pattern can be realised through a generic traceability interface for arbitrary model transformation engines. Depending on the approach of trace link generation (explicit or implicit), we claim two possible augmentation methods to achieve traceability as described above. In view of the generic traceability interface, these augmentation methods require the implementation of two different kinds of connectors. In this way, we propose *blackbox* and *invasive* connectors for implicit and explicit generation, respectively.

The design pattern is evaluated in terms of three connectors for QVT as well as ATL (blackbox connectors) and Xpand (invasive connector).

C3: Parallel Model Matching for Trace-Link Generation

We contribute a traceability mechanism based on model matching for inaccessible or non-existing transformation engines (blackbox systems). Thereby, blackbox systems can be augmented with a traceability mechanism.

To achieve this, we propose a novel, language-agnostic concept defining three similarity measures for model matching to generate trace links. Furthermore, as common to parallel matching systems, the matching results due to different matching algorithms are combined. In realisation of this concept, 8 matching algorithms are deployed to instantiate the three similarity measures. In doing so, we base our work on the metamodel matching system from Voigt et al. [Voi11] to apply specialised graph-based matching using planar graphs with the benefit of improving matching results.

Furthermore, we experiment with metamodel-matching techniques applied to model matching, such as filtering and blocking, defined under the term *metamodel-driven matching*. It turns out that the key-enabler for leveraging model matching for trace link generation is the exploitation of metamodel-matching techniques.

Finally, we contribute an evaluation of model-to-model and model-to-text transformations with respect to 29 mappings from a prominent set of transformations from the domain of MDSD, called ATL Zoo, as well as 12 mappings from an SAP business application.

1.2. Thesis Outline

The content of this work is structured into the following chapters. First, we give an overview on the basic concepts of our work in Chapter 2. As such, we introduce the notion of traceability in MDSD, typed attributed graphs—used as internal data model for matching, furthermore, model matching systems and facets.

After having introduced the foundations, we provide a motivating example for traceability in Chapter 3, to lead the way to our problem definition. In doing so, we introduce three categories of trace link generation, which we use throughout this thesis. Based upon these, we span the scope of our work in terms of our requirements analysis.

The chapters 4 to 7 describe our approach and reflect all components of the generic traceability framework. Chapter 4 gives an overview on its architecture and description of its main components, where the following three chapters are dedicated to each of the components. At first, the generic traceability metamodel is discussed in Chapter 5. We derive a minimal and elementary set of 4 trace-link types based on the semantics of CRUD actions from database operations. The derivation is founded on the classification of source-target relationships from [CH06]. Next, we define an extensibility mechanism for the traceability metamodel, achieved through the use of facets and facet hierarchies.

Secondly, in Chapter 6, we present a methodology for achieving the automatic generation of trace links. The methodology accounts for both explicit as well as implicit generation approaches. Core to the methodology is the generic traceability interface, which defines the contract between a given transformation and traceability engine. Depending on the generation approach, the methodology proposes two different kinds of connectors as part of this contract, namely *blackbox* and *invasive* connectors for implicit and explicit generation, respectively. Both connectors serve the purpose of augmenting model transformations for the sake of generating traceability data.

Finally, we introduce a matching system as part of the generic traceability framework in Chapter 7. We define three similarity measures upon which the proposed matching algorithms work. The matching system implies a parallel matching process by combining the matching results of numerous matching algorithms. As such, the implementation of the matching system incorporates 8 matching algorithms, all derived from the similarity measures.

To demonstrate the feasibility of our approach, an evaluation for both the methodology for augmenting transformations as well as the matching system is presented in Chapter 8. The evaluation scenarios are from an SAP business application and the ATL Zoo. In Chapter 9, we discuss related work of our approach in view of the latter methodology and matching system. Finally, we conclude this thesis in Chapter 10 with a summary and an outlook on future work.

2

Foundations

This chapter presents the foundations of our work and is structured as follows. First, we introduce the paradigm of Model-driven Software Development (MDSD) in Section 2.1 and the notion of traceability in Section 2.2. Since our work is based on model matching, we provide an overview on the basic concepts of model matching systems in Section 2.3. The introduced matching algorithms operate on graphs. Therefore, Section 2.4 is dedicated to the fundamentals of graph theory and typed attributed graphs. Finally, we define the concept of facets in preparation of defining traceability models based on facets in Section 2.5.

2.1. Model-driven Software Development

This section introduces the paradigm of Model-driven software development (MDSD). MDSD denotes the—partial or entire—generation of software out of models. The request for proposal of the Object Management Group on Query/Views/Transformations (QVT) in 2002 and the final adopted QVT specification [Obj11] in 2005 led to the development of numerous model transformation approaches, as summarized in [CH06].

A model is an abstraction of a system or its environment, or both [CH06]. Models are derived from a domain-specific modelling language (DSL). In general, a DSL is a language that provides domain-specific abstractions and language constructs.

Definition 1 (Model)**.** *A* ***model*** *defines formal knowledge about an application domain expressed in a domain-specific language [SV06].*

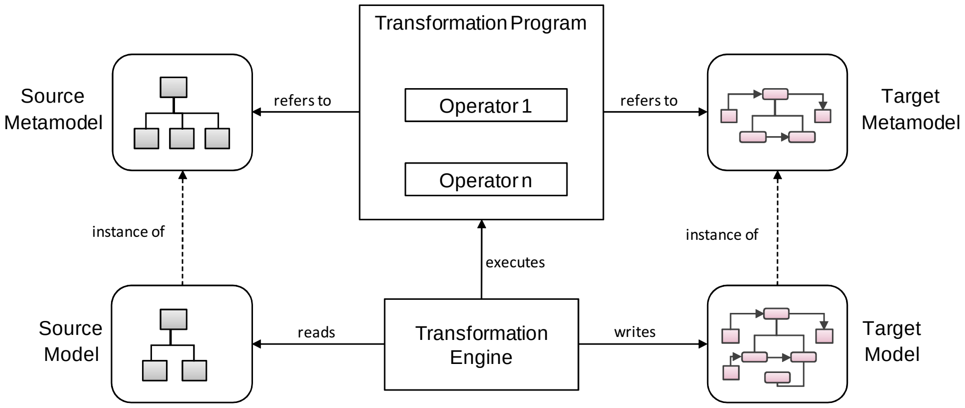


Figure 2.1.: Transformation Program

Metamodels are domain-specific languages and thus provide the formal definition of models. The formal definition of models is necessary, since the goal of MDSD is to *automatically* generate executable code (at least various parts) from one or more models.

The key to MDSD lies in the concept of model transformations. Through the execution of a model transformation a given source model is transformed into a target model. Thereby, the transformation engine is directed through the transformation program at runtime. A transformation program (cf. Figure 2.1) includes a finite set of *transformation rules*, which are defined on the basis of the source and target metamodel elements. Each transformation rule describes an instruction on how a set of elements of a certain instance of the source metamodel is transformed into a set of model elements of a certain instance of the target metamodel. Formally, we define a model transformations in analogy to [Kle08], as follows. Thereby, we define the *instance of* relationship between models and metamodels in terms of an element (a model) of a certain type (the metamodel) in analogy to the relationship an integer has to the set of integers.

Definition 2 (Model Transformation)**.** *Let S be an instance of the source metamodel S^M. Let T be an instance of target metamodel T^M.*

*Then, a **model transformation** is defined as a mapping MT from S^M to T^M.*

$$MT : S^M \to T^M \text{ with } S \mapsto T$$

In other words, a model transformation allows to transform a given source model S conforming to S^M into a target model T conforming to T^M.

Generally, model transformations may be applied to multiple source and target models, for example in the case of model merging or model weaving. The above definition may be extended accordingly. The transformation program is written in a certain language, called *transformation language* or *transformation program language*. The transformation rules included in the transformation program have different forms depending on the transformation language used.

Transformation rules may be distinguished into *hard-coded* or *model transformation* rules according to [Kle08]. Hard-coded transformation rules are written in a template language (e.g., the model-to-text transformation language XPand from oAW [Foug]), or programming language (e.g., Java). Model transformation rules are specified according to the QVT standard, for example the specification of the Object Management Group [Obj11].

Transformation rules may be characterised according to their domains, where a domain is defined as the part of a rule used for accessing the model(s) on which the rule operates. When the domain is used for accessing the source model, it is referred to as source domain, otherwise, if used to access the target model, target domain. We give examples by using the above classification to underline the variety of possible structures of domains:

- Hard-coded Rules: In case of template-based approaches, the rule includes variables (to bind elements from the source and/or target models) next to elements of the transformation language embedded in a string body. The source domain refers to the variables, while the target domain is the remaining string body.
- Model Transformation Rules: In QVT Relations, a rule incorporates a distinguished typed variable in the domain definition that can be matched in the processed model. In QVT Operations, the rule takes on the form of an imperative procedure. The domain definition then corresponds to the parameter and the code that accesses model elements by using the parameter as an entry point.

In the scope of this work, we will use sub-units of transformation rules, called *operators*, to have a more fine-grained view on how rules transform source elements into target elements. For example, an operator may be a query, e.g., in OCL, or coding for controlling the execution flow, such as iterative or conditional clauses.

Definition 3 (Transformation Operators)**.** *Given a transformation language TL and source and target metamodels S^M and T^M with source model S conforming to S^M and target model T conforming to T^M.*

*Then a **transformation operator** op is defined through the following characteristics:*

- *an operator is an element of TL conforming to the abstract syntax of TL, that is, $op \in TL$.*
- *an operator is a mapping from a set of source-metamodel elements $\{s_i^M : i = 1, \dots, n\} \subseteq S^M$ to a set of target-metamodel elements $\{t_j^M : j = 1, \dots, m\} \subseteq T^M$, such that:*

  $$\begin{aligned} &op(s_1^M, \dots, s_n^M) = (t_1^M, \dots, t_m^M) \text{ with} \\ &(s_1, \dots, s_k) \mapsto (t_1, \dots, t_l), s_i \in S, t_j \in T, i = 1, \dots, k,\ j = 1, \dots, l \end{aligned}$$

  In other words, the mapping allows to transform a set of source elements from S (where each source element s_i conforms to one of the metamodel elements from $\{s_i^M : i = 1, \dots, n\}$) into a set of target elements of T (where each target element t_j conforms to one of the metamodel elements from $\{t_j^M : j = 1, \dots, m\}$).

Remark We only provide a classification on the operators relevant to our work in Chapter 6, Section 6.2.

An operator may also be defined for an empty set, that is,

$$op(s_1^M, \dots, s_n^M) = (t_1^M, \dots, t_m^M) \text{ with } \{s_1^M, \dots, s_n^M\} = \emptyset$$

e.g., in model-to-text-transformations to generate static text. Analogously, a delete operator is defined as

$$op(s_1^M, \dots, s_n^M) = (t_1^M, \dots, t_m^M) \text{ with } \{t_1^M, \dots, t_m^M\} = \emptyset.$$

Since transformation rules can be interpreted as operators, we define a broader term, *transformers*, for both notions.

Definition 4 (Transformers)**.** *The notion of* ***transformers*** *comprises transformation rules and transformation operators. A transformer, thus, relates to a transformation rule, or a transformation operator.*

There are two types of model transformations, *model-to-model* (M2M) transformations and *model-to-text* (M2T) transformations [CH06, SV06]. While a M2M transformation creates a target model as an instance of the target metamodel, a M2T transformation creates text in terms of strings. Thus, the target of a M2T transformation is not formalized through a metamodel. Often, M2T transformations are referred to model-to-code transformations, if the target text represents a particular programming language and may be interpreted as a special case of a M2M transformation, provided a metamodel exists for the target programming-language [CH06].

Furthermore, model transformations may be categorized into *exogenous* resp. *endogenous* transformations [MG06]. Endogenous transformations are transformations between models expressed in the same language (if $S^M = T^M$), while exogenous transformations are transformations between models expressed using different languages.

Typical examples for exogenous transformations are [MG06]:

- *Synthesis* transforms a higher-level (more abstract) specification into a lower-level (more concrete) specification, for example, in code generation, where the source code is translated into bytecode or executable code, or where the design models are translated into source code.
- *Reverse engineering* is the inverse of synthesis and extracts a higher-level specification from a lower-level one.
- *Migration* transforms a program written in one language to another, yet the same level of abstraction is kept.

Typical examples for endogenous transformations are [MG06]:

- *Optimization* improves certain operational qualities (e.g., performance), while preserving the semantics of the software.

	horizontal	**vertical**
endogenous	Refactoring	Formal refinement
exogenous	Language migration	Code generation

Table 2.1.: Orthogonal model transformation types ([MG06])

- *Refactoring* is used for changing the internal structure of software to improve certain characteristics of software quality without changing its observable behaviour.
- *Simplification and normalization* decrease the syntactic complexity, e.g., by simplifying the language syntax for better readability.

The same taxonomy [MG06, GS04] distinguishes between *horizontal* and *vertical* transformations. The direction specification refers to the level of abstraction: For vertical transformations source and target remain on different levels, while for horizontal transformations they are on the same level. A prominent example for a horizontal transformation is *refactoring*. An example for a vertical transformation also referred to as refinement transformation, is code generation.

In fact, the above-mentioned transformation types are orthogonal [MG06] as presented in Table 2.1 with prominent examples.

In this work, we will use the Java-based implementation of MOF [Gro06], the Eclipse Modelling Framework (EMF) [SBPM08] as language for expressing models.

2.2. Traceability in MDSD

Traceability of artefacts provides the means of understanding the complexity of logical relations existing among artefacts generated during the software development lifecycle. With the inception of model-driven software development (MDSD) [SV06] entailing the use of models for different domains, such as business processes, system requirements, architecture, design and tests, the scope of artefacts has clearly been diversified. Maintaining and defining relations and dependencies between artefacts is a nontrivial task and has been a known challenge since the early 1970s. Questions concerning design decisions, or whether a certain requirement is necessary and what its effect on system level would be if deleted, all urge the need for continuous traceability of artefacts throughout the development lifecycle.

Although the advantages of traceability have been identified, its realization in practice has only taken hold scarcely [Ale02, EH91]. The commonly stated reasons are: (1) the high cost of manual creation and maintenance of traceability information; (2) lacking heuristics to what kind of trace link information should be tracked; (3) discrepancies between traceability stakeholders (those creating links and those using links); (4) lack of adequate tool support; (5) artefacts are written in different languages (natural vs.

programming languages) (6) and they describe a software system at different levels of abstraction (design versus implementation).

In the IEEE Standard Glossary of Software Engineering Terminology [IEE90] the notion of traceability is defined as:

Definition 5 (Traceability)**.** *The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.*

Related to MDSD, traceability is defined as follows:

Definition 6 (Traceability in MDSD)**.** *Traceability information in MDSD can be understood as the runtime footprint of the execution of a model transformation [CH06].*

Based on this definition, we interpret the runtime footprint of a transformation as the tracking of corresponding source and target artefacts of *transformers* in the transformation program. Thus, we arrive at the following definition of traceability in MDSD.

Definition 7 (Traceability in MDSD)**.** *Traceability information in MDSD is the tracking of corresponding source and target artefacts of transformers in the transformation program.*

Essentially, trace links provide this kind of information by associating source and target model elements with respect to the execution of a certain model transformation.

Definition 8 (Trace Link)**.** *Given a model transformation MT from source model* S *to target model* T*. Let* $\{s_i : i = 1, \ldots, n\}$ *be the set of all model elements of* S *and* $\{t_j : j = 1, \ldots, m\}$ *be the set of all model elements of* T*.*

Then, the set of trace links for MT is defined as a binary relation ***trace_rel*** *between* $\{s_i\}$ *and* $\{t_j\}$*, that is, as subset of the Cartesian product of the sets* $\{s_i\}$ *and* $\{t_j\}$*:*

$$\boldsymbol{trace\_rel} \subseteq \{s_i\} \times \{t_j\}$$

Generally, we classify trace links into implicit and explicit link types, as proposed by [POKZ08].

- Implicit trace links—referring to trace links between artefacts that arise due to a MDSD operation, e.g., a model transformation as in Definition 8.
- Explicit trace links—referring to trace links explicitly defined between artefacts using one or more languages. Explicit trace links can be further categorised into trace links between models and corresponding model elements (e.g., a dependency relationship drawn between two UML packages) or between models and artefacts, that do not conform to a metamodel (e.g., between a model and a requirement).

Furthermore, any set of collected trace links is referred to as *traceability information*, or *traceability data*.

2.2.1. Traceability Scenarios

In the following section, we list possible use cases for traceability data, called traceability scenarios. We introduce each scenario regarding its general applicability in software development and then state its relevance to MDSD.

System Comprehension

Traceability data may be used to derive logical and functional dependencies between artefacts, being from the same development phase (e.g., design, implementation and testing), or different phases within a software development project. In [OO07] this is also referred to as *trace inspection*.

Concerning MDSD in particular, the following traceability scenario is conceivable. The paradigm of MDSD raises the level of abstraction by encapsulating implementation details into transformations [SV06]. In doing so, the total system complexity concerns, firstly, the mapping of model to target language (i.e., generation, or interpretation) and secondly, the complexity of the source model(s) as such. The advantage of this separation of concerns is, that a developer only interested in the modelling part of a system, not necessarily needs to know the technical details of the implementation. Consequentially, we have different stakeholders in an MDSD process, a) those using the transformation and b) those writing a transformation program, which are not necessarily the same person.

To provide the stakeholder using the transformation and likely being unfamiliar with the transformation language with an adequate system comprehension and still uphold the above-mentioned advantage of separation of concerns, traceability data may be useful. For example, trace links capturing the logical and functional dependencies between representations of design artefacts and generated code may aid the developer in comprehending the transformation easier. This is especially useful for the system comprehension of a chain of model transformations and large-scale development.

Coverage Analysis

A coverage analysis is used to determine the degree to which artefacts are followed up by other artefacts in a development process [OO07]. For example, to answer question, whether all market requirements have been implemented, or whether all software requirements were covered by test cases in the development life cycle.

There are slight variations to the notion of coverage analysis, depending on the types of artefacts involved. A special case of a coverage analysis is referred to as *code coverage* or *test coverage*, dealing with the coverage of source code through test cases as part of software testing [MM63, LHL02, SW08].

Regarding an MDSD process, the traceability of models and model elements is an integral and necessary part for a coverage analysis. For instance, a coverage analysis may be useful for the following scenario motivated through a system compliance check. Assuming a bug was found and corrected in a low-level artefact and one needs to know, which test cases to re-run to check whether the system is still correct. Re-running all test cases may not be a viable solution for a large-scale test suite where time is a limiting factor. A coverage analysis with respect to the changed artefact allows to find the requirements (or features), which are covered up by this changed artefact. This is achieved by calculating the transitive closure on all trace links from the changed artefact to the requirement(s). From this given set of requirements another coverage analysis renders all associated test cases again by transitive closure on the corresponding trace links, yet in the opposite direction.

Another use case for a coverage analysis bound to MDSD is for checking, whether all necessary parts of the input model are actually utilised by a transformation [OO07], for example, in case of a model-to-text transformation generating code documentation from a specific model. For large models this can be a tedious task due to cross-references. A coverage analysis allows for checking to which extent all relevant model elements have been transformed.

A coverage analysis is used in [CH06] for the feature *target incrementality*, that is, the ability to update existing target models based on changes in the source models. A target-incremental transformation creates the target models if they are missing on the first execution, in other words, if corresponding source models are not covered or followed up on yet. To detect whether a certain target model and corresponding model elements already exist, trace links can be used.

Change Impact Analysis

A change impact analysis allows to determine how changing one artefact would affect other artefacts [HJD11, AB93]. For example, a high-level artefact needs to be updated, e.g., by adding a new alternative flow to a use case. Afterwards one needs to provide an estimate for the effort required for the implementation of the change. Transitive closure on trace links from the high-level artefact provides an upper bound understanding of what lower-level artefacts could be impacted, allowing an estimate of the ”size” of the change.

Related to MDSD, traceability data can be useful in analyzing how changing one model would affect other related models (and corresponding model elements) [CH06], or existing generated code due to model-to-text transformations [OO07]. Re-generating artefacts to propagate a model change is not always a viable option, since it is generally not possible to generate all artefacts automatically within an MDSD process, consequently, manually added artefacts need to be maintained. Secondly, generated artefacts may have cross-dependencies. Thus a change impact analysis is useful in finding an estimate on potentially impacted artefacts before deciding on re-generating the transformation, especially in the case of large-scale model transformation chains.

Orphan Analysis

An orphan analysis is to find orphaned model elements with respect to a specified trace link [OO07]. A so-called orphan refers to one or more artefacts that are not referenced by, or reference other artefacts. Orphans are created in case of artefact deletion, e.g. while deleting a model element, the corresponding code might be outdated. Re-running the model transformation as a result of an orphan analysis to render updated traceability data is one part of the analysis in addition to the handling of manually created trace links.

Model Transformation Debugging

According to [CH06], trace links may be used for debugging of model transformations, or model-based debugging (i.e., by mapping the execution of an implementation back to its high-level model in numerous steps) to locate bugs during the development of transformation programs and later in shipped applications. For example, if a problem was discovered in the generated file of a model-to-text transformation, a developer can utilize the trace links to identify the snippet in the transformation program and model data that caused the generation of the defective part.

Further Scenarios

In the following, we list other traceability scenarios found in literature [OO07].

- Model Synchronization: While transformation source and target evolve, trace links may be used to keep both source and target in synchronization [CH06]. For example, while using model transformation approaches that account for preserving user edits in the target. After changing the model input, re-generating the target does not automatically update preserved edits, potentially leading to updated model elements, yet outdated edits. Trace links can be used to resolve this issue of discrepancy. Another scenario on the synchronization of cardinality-based feature models and their specializations is given in [KC05].
- Unprotected Block Checking: For model-to-text transformation approaches providing support for unprotected blocks (to preserve user edits) in the generated code, the following functionality is conceivable. After unprotected blocks have been implemented, the corresponding traceability data needs to be updated to show their completion. A status report on completed and uncompleted unprotected regions may be useful for more efficient project and resource planning.
- Traceability Model Merging: Merging of traceability models may be used, when several different transformations are executed from the same source model. Merging these traceability models provides a more compact view on trace links refer-

encing the same source elements, yet different target elements, for example while generating both source code as well as documentation from a source model.

- Traceability Model Evolution: The evolution of traceability models may be used to understand the evolution of a corresponding transformation source with respect to its target.

2.2.2. Traceability in MDSD

We look into the possible ways on how trace links can be generated in MDSD. According to Vanhooff et al. [VBJB07] transformation approaches either generate trace links *implicitly* or *explicitly*, also referred to as *automatic* or *manual* generation according to Czarnecki and Helsen [CH06]). In the former case, an integrated support for traceability (e.g., QVT [Obj11], MOFScript [Fouf]) is already provided, while the latter case requires effort to encode traceability as a regular output model from transformation to obtain a traceability solution. The explicit trace link generation may be achieved in two ways according to Olsen and Oldevik [OO07], either by writing traceability-specific rules into the transformation program for each transformation (e.g., ATL [Foud], oAW [Foug]), or to make use of a higher-order transformation on the transformation model, as described in [Jou05] for ATL.

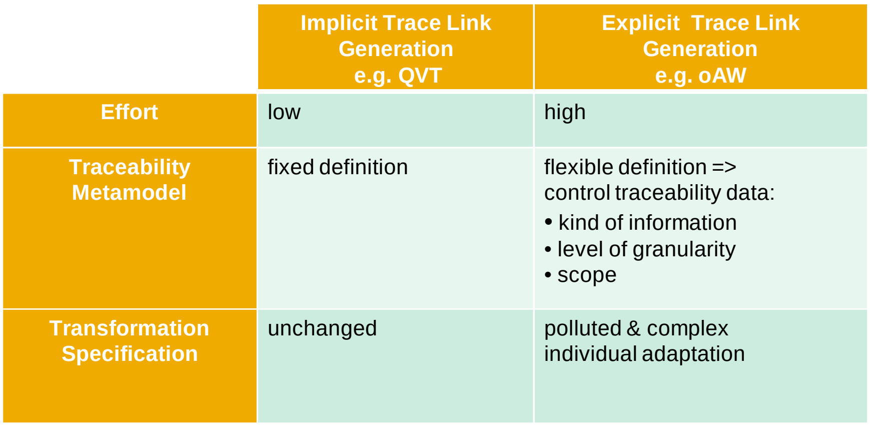
In analogy to the classification above, we adopt the terms explicit and implicit trace-link generation and define the following two classes of model transformation approaches.

Definition 9 (Implicit Trace-Link Generation Class)**.** *The implicit trace link generation class refers to all model transformation approaches with an integrated traceability mechanism to generate trace links.*

Definition 10 (Explicit Trace-Link Generation Class)**.** *The explicit trace link generation class refers to all model transformation approaches with a traceability mechanism to generate trace links requiring the encoding of traceability. Encoding of traceability is the process of either writing traceability-specific rules into the transformation program, or making use of a higher-order transformation on the transformation model to achieve this.*

Next, we summarise the advantages and disadvantages of the above two trace link generation classes, as tabularized in Figure 2.2. The major advantage of implicit trace-link generation is the fact that no additional effort is necessary to obtain trace links between input and output models, as they are generated automatically in parallel to the actual model transformation. A disadvantage is, that the traceability metamodel is fixed and since most transformation approaches use different metamodels, standardization among different approaches is aggravated. Essentially, the implicit generation allows less flexibility to control traceability data due to the fixed traceability metamodel. According to [CH06] traceability data might be controlled with respect to:

- The kind of data, e.g., the links between source and target elements, the operators that created them.
- The granularity level of data, e.g., tracing only on file level instead of more concrete entities in the file body. Setting the granularity of trace links which may differ from one traceability scenario to another has been identified as a challenge by the Center of Excellence for Traceability [ABE$^+$06]. When tracing all model element references, the number of trace links might become incomprehensible and hence less useful to the developer. Furthermore, it might be a performance issue when handling large and complex model transformations.
- The scope for which the data is recorded, e.g., tracing for specific rules, or subset of the source model. Not all model information might be allowed to be traced for security reasons, mandated for instance by customer requests.



	Implicit Trace Link Generation e.g. QVT	**Explicit Trace Link Generation e.g. oAW**
Effort	low	high
Traceability Metamodel	fixed definition	flexible definition => control traceability data: • kind of information • level of granularity • scope
Transformation Specification	unchanged	polluted & complex individual adaptation

Figure 2.2.: Implicit and Explicit Trace-Link Generation Class

Alternatively, as in the case of explicit trace link generation, it is possible to treat traceability as a regular output model of the transformation and incorporate additional transformation rules into the transformation template to generate the traceability model (e.g., ATL [Foud], oAW [Foug]). The choice of metamodel is then completely at the discretion of the developer and does not depend on the transformation engine. Hence, control over the traceability data modelling is given. The drawback however, is, that additional effort is required to add traceability-specific transformation rules, which may also pollute the implementation. As this task is generally done manually, it is likely to be error-prone and time consuming. Moreover, this effort is repeated for each transformation.

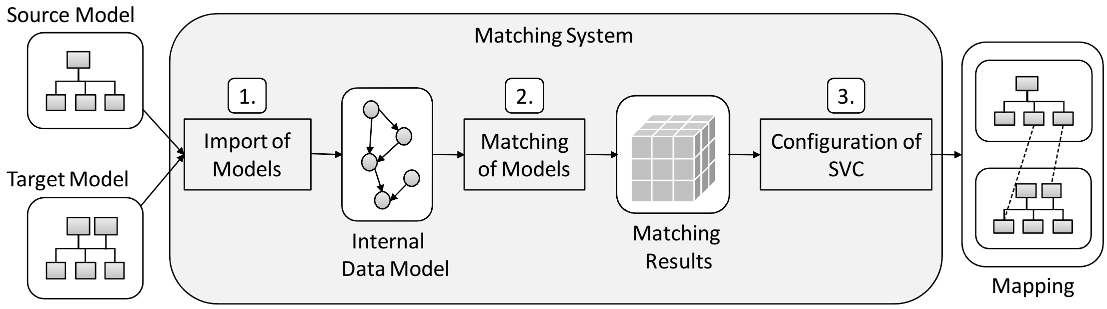


Figure 2.3.: Common Matching Process

2.3. Model Matching

The process of model matching is to find semantic correspondences between model elements. Such correspondences are called *matches*. The general matching process receives a source and target model and creates a mapping between elements of these models, following the definition of Rahm and Bernstein in [BLP00].

Definition 11 (Matching)**.** *The* ***matching process*** *is defined through a mapping from a source S and target model T to a set of n-ary mappings, where each mapping is defined from a set of source elements of S to a set of target elements of T. Each mapping specifies that a set of elements of S corresponds, that is, matches to a set of elements in T. The semantics (e.g., type and value equality) of a correspondence can be described by an expression attached to the mapping.*

The matching process is implemented through a matching system. There are three kinds of matching systems as described later in Section 2.3.1. Commonly, a matching system involves the following steps as depicted in Figure 2.3:

1. *Import* of models: to transform the models that need to be matched into the internal data model of the matching system
2. *Matching* of models: calculates a similarity value between all pairs of source and target elements
3. *Configuration* of similarity value cube: aggregates the matcher results to create an output mapping

1. **Import of models**: During the import, the source and target models that need to be matched are transformed into the internal data model of a matching system to have a common basis for arbitrary matching algorithms. In some systems e.g., [LTLL08], pre-processing steps are applied, that is, properties of the input models are exploited to adjust the matching process. For example, in case the element names of the input models differ to a large extent, the weights of name-based matching techniques are adjusted.

2. **Matching of models**: Different matching algorithms are applied to the imported source and target model in order to identify semantic correspondences (matches) between source and target elements. A pair of identified source and target elements is called a *match*. We define a matching algorithm as follows:

Definition 12 (Matching Algorithm)**.** *Let* $\{s_i : i = 1, \ldots, n\}$ *be the set of model elements of source model* S *and* $\{t_j : j = 1, \ldots, m\}$ *be the set of model elements of target model* T*.*

*A **matching algorithm** is defined through a 2-ary mapping **matchAlg** from* $\{s_i\}$ *and* $\{t_j\}$ *to the interval of real values between* 0 *and* 1*, that is:*

$$\boldsymbol{matchAlg} : \{s_i\} \times \{t_j\} \rightarrow [0, 1] \subset \mathbb{R}$$

A given output value of a matching algorithm represents the degree of similarity of the related source and target element.

Definition 13 (Similarity Value)**.** *Each output value of matchAlg is called a **similarity value**, where* 1 *is the greatest possible similarity.*

The matching system can be configured by choosing which matching algorithms should be involved in the matching process. For a classification on different matching algorithms refer to Section 2.3.2.

Definition 14 (Matcher)**.** *A **matcher** implements a matching algorithm.*

3. **Configuration of similarity value cube**: Each matching algorithm provides separate results for a certain source and target model, where the results describe a similarity value for all source and target model element combinations, called *similarity value matrix*. All of these matrices are arranged into a cube, called *similarity value cube* (SVC). To derive a mapping (or matches) in terms of Definition 11 between source and target elements out of these results, the similarity value cube needs to be configured, that is, different strategies [BMPQ04, DR07, VIR10, PMK10] are applied to aggregate the matcher results. Common strategies are e.g., to form an *aggregation matrix* by calculating the average of similarity values for a given source and target element, or *selection matrix* by selecting all elements exceeding a certain threshold. More details on the configuration strategies follow in Appendix B.3.

Definition 15 (Similarity Value Cube)**.** *Let* S *and* T *be a source and target model, respectively.*

Let $\{s_i : i = 1, \ldots, n\}$ *be the set of model elements of* S *and* $\{t_j : j = 1, \ldots, m\}$ *be the set of model elements of* T *and* $\{m_k : k = 1, \ldots, l\}$ *be a set of matchers.*

Then, a similarity value cube SVC *is a* 3*-dimensional data cube defined through the mapping:*

$$SVC : \{s_i\} \times \{t_j\} \times \{m_k\} \rightarrow [0, 1] \subset \mathbb{R}$$

2.3.1. Classification of Matching Systems

If the matchers in the matching system are applied in parallel, this implies a parallel matching system. Essentially, there are three kinds of matching systems as mentioned above:

- Parallel matching systems
- Sequential matching systems
- Hybrid matching systems

Parallel matching systems, e.g., [DR07, VIR10], apply each matcher in parallel and independently on the input models. After execution of the matchers, their results are aggregated.

Sequential matching systems, e.g., [FV07, FHLN08], apply matchers in sequential order, that is, the results of a matcher serve as input for the succeeding matcher. This allows for an incremental refinement of matching results, however may potentiate an existing error.

Another possibility is a combination of the above-mentioned systems in hybrid matching systems, for instance [HZL$^+$07] use fix-point calculations by incrementally executing parallel matchers to use their results as input for the same set of matchers. In [PBR10], user-specific compositions of matchers are applied. A composition results in a set of matchers that are combined through operators that have an order. This allows, for instance, for an intersection or union of matcher results. Our work, however, is based on a parallel matching system.

2.3.2. Classification of Matching Algorithms

Several classifications on matching algorithms have been proposed. Prominent examples from schema matching and ontology matching are, the classification of Rahm and Bernstein [RB01] as well as Shvaiko [SE05, CSH06]. A consolidated view on the latter classifications can be found in [Voi11] referring to metamodel matching as summarized in the following.

Matching algorithms may be divided into two classes, that is, matching is either applied on *element level* or *structure level*. While the former case deals with the comparison of elements and their properties, e.g., comparing their names, the latter case is based upon the graph structure.

Element-level Algorithms

Element-level matching algorithms make use of information available as properties of elements, for example the name or data type of an element. The following classes exist:

- **String-based** String-based algorithms cover similarity calculation using string information. For example, an element's name may be used, or metadata such as documentation or annotations, etc. Furthermore, the similarity calculation may be based on common prefixes or suffixes. More advanced processes are based on the edit-distance by calculating the number of edit operations necessary to transform one string into another, e.g., the Levenshtein-distance [WF74].
- **Constraint-based** Constraint-based matching techniques use information of elements which define a certain constraint on an element, e.g., data types, keys, or cardinalities. Data types are used to derive a similarity of elements based on the data type's similarity. For simple types such as integer or float a static type conversion table can be applied, while for complex types such as structures more advanced techniques are needed.
- **Linguistic resources** Matching algorithms may rely on external sources such as linguistic resources. These resources can be dictionaries, a common knowledge thesaurus or a domain-specific dictionary. An example for a domain-specific dictionary is WordNet [Fel98] a publicly available dictionary used for matching.
- **Mapping reuse** Algorithms with mapping reuse base their calculation on existing mappings, for instance, by using transitivity as an indicator for similarity, that is, existing mappings may conclude mappings for their transitive dependent elements. A more sophisticated approach is used in [DR02], which generalizes this idea to reuse match results at the level of entire schemas or schema fragments.

Structure-level Algorithms

Matching algorithms using the graph structure to derive similarities between elements are based on the premise that relations (e.g., inheritance and containment) between elements and their position (building up the structure) are similar for similar elements. Numerous algorithms operating on this structure-level have been proposed:

- **Graph-based** Graph-based algorithms either use the overall graph (global graph-based algorithms), or only parts of the graph, that is, local graph-based and region graph-based algorithms:
  - Global graph-based: The former case, distinguishes between exact or inexact algorithms. Exact algorithms calculate a mapping from one node (element) to another as well as a mapping for edges, as for example in sub-graph isomorphism algorithms allowing structure-preserving bijection. In contrast, inexact algorithms, e.g., the graph edit distance or maximum common sub-graph algorithms, allow for an error-tolerant approach, that is, allow certain distances between sub-graphs.

    Inexact algorithms such as the graph edit distance or maximum common subgraph algorithms apply a sequence of edit operations, composed of: add, remove, and relabel (rename). A sequence of such operations defines a mapping from one graph onto another, thus calculating the maximal common subgraph along with the necessary operations.

  - Local graph-based: In contrast, graph algorithms may regard the context of an element as opposed to the whole graph, that is, the relation of this element to its neighbours in the graph. Local graph-based matching algorithms operate on a tree use the children, leaf, sibling and parent relationship [VIR10, DR02], relative to a given element to derive mappings. An extension of these algorithms is to generalize a graph's spanning tree and use the neighbours in the graph for matching.
  - Region graph-based: Region graph-based algorithms make use of regions within a graph, that is, sub-graphs of a given graph. The algorithm calculated the frequency of such sub-graphs in the given graphs. The underlying premise of the algorithm is that sub-graphs sharing a high frequency are more similar. Thus, this frequency is used to derive a similarity between the sub-graphs' elements. An example of region graph-based algorithms is the graph mining matcher in [Voi11] or the filtered context matcher in [DR07].

- **Taxonomy-based** Taxonomy-based matching algorithms operate on trees and commonly are specialized on this structure, for instance name path matching. A name path matcher is an extension of the name matcher and calculates similarities of model elements with the help of their path, where a path is the concatenation of the element names from the root to the current element.
- **Repository of structures** This approach uses a repository of structures containing mappings, corresponding models and co-efficients denoting similarities between models. The storage of similarities allows for a faster retrieval of mappings. The co-efficients are metrics, for example, structure name, root name and maximal path length. These numbers act as an index for a set of models, thus allowing an efficient retrieval.
- **Logic-based** Logic-based matching algorithms make use of additional constraints defined on models. In doing so, matching is based on these constraints in a logic language, or by adding mappings upon reasoning via post-processing. For instance, if a mapping exists between attributes, it follows that a mapping between the containing classes has to exist as well, since attributes need a containing element.

2.4. Graph Theory

In the following section, we introduce concepts of graph theory relevant to our work. From the field of graph transformation, we introduce *typed attributed graphs*, which have been used as a modelling technique in software engineering and as a metalanguage to specify and implement visual modelling techniques such as UML [EEPT06]. In the following, we align our work with [EPT04, EEPT06] to derive the definition of typed attributed graphs.

2.4.1. Graph Definitions

A graph is a structure consisting of nodes and edges, where each edge links two nodes. To define the start node (source) as well as the end node (target) of an edge, we introduce two functions, the source and target function.

Definition 16 (Graph)**.** *A* ***graph*** $G = (V, E, s, t)$ *consists of a set* V *of nodes (also called vertices), a set* E *of edges and two functions* $s, t : E \longrightarrow V$*, the source and target function.*

A graph is considered directed, if every edge has a distinguished start node (its source) and end node (its target). To represent undirected graphs, for each undirected edge between two nodes v and w we add both directed edges (v, w) and (w, v) to the set E of edges. A graph is finite, if V is a finite set of nodes. Otherwise, if V is an infinite set, the graph is infinite. In this work, we will consider finite graphs.

Definition 17 (Cardinality of Nodes and Edges)**.** *The* ***cardinality*** *of the set of nodes of a graph, denoted as* $|V|$*, is defined as the number of nodes* n *of* V*. Thus, we write:* $|V| = n$*. Analogously, the cardinality of the set of edges is defined as the number of edges* m *of* E*, that is:* $|E| = m$*.*

Graphs are related by graph morphisms, which map the nodes and edges of a graph to those of another one, preserving the source and target of each edge.

Definition 18 (Graph Morphism)**.** *Given graphs* G_1, G_2 *with* $G_i = (V_i, E_i, s_i, t_i)$ *for* $i = 1, 2$*.*

Then, a graph morphism $f : G_1 \longrightarrow G_2$*,* $f = (f_V, f_E)$ *consists of two functions* $f_V : V_1 \longrightarrow V_2$ *and* $f_E : E_1 \longrightarrow E_2$ *that preserve the source and target functions, i.e.* $f_V \circ s_1 = s_2 \circ f_E$ *and* $f_V \circ t_1 = t_2 \circ f_E$*.*

A graph morphism f *is injective (or surjective), if both functions* f_V*,* f_E *are injective (or surjective, respectively);* f *is called isomorphic if it is bijective, which means both injective and surjective.*

In order to model attributed graphs with attributes for nodes and edges, we have to extend the classical notion of graphs (see Definition 16) to *E-Graphs*. An E-graph has two different kinds of nodes, representing the graph and data nodes as well as three kinds of edges, the usual graph edges and special edges used for the node and edge attribution.

Definition 19 (E-Graph)**.** *An E-graph G with* $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}})$ *consists of the sets:*

- V_G *and* V_D*, called the graph and data nodes (or vertices), respectively;*
- E_G*,* E_{NA}*, and* E_{EA} *called the graph, node attribute, and edge attribute edges, respectively;*

and the source and target functions:

- $source_G : E_G \longrightarrow V_G, target_G : E_G \longrightarrow V_G$ *for graph edges;*
- $source_{NA} : E_{NA} \longrightarrow V_G, target_{NA} : E_{NA} \longrightarrow V_D$ *for node attribute edges;*
- $source_{EA} : E_{EA} \longrightarrow E_G, target_{EA} : E_{EA} \longrightarrow V_D$ *for edge attribute edges.*

Definition 20 (E-Graph Morphism)**.** *Consider the E-graphs* G^1 *and* G^2*. An E-graph morphism* $f : G_1 \longrightarrow G_2$ *is a graph morphism holding for E-Graphs* G^1 *and* G^2*.*

Remark The main difference between E-graphs and graphs is that we allow edge attribute edges, where the source of these edges is not a graph node but a graph edge.

2.4.2. Typed Attributed Graphs

An attributed graph is an E-graph combined with an algebra over a data signature. In the signature, we distinguish a set of *sorts*, where the elements of a sort are attribute values, e.g., the sorts {natural numbers, strings}. The corresponding carrier sets in the algebra can be used for the attribution. We give a definition of an algebra as well as a final algebra and corresponding signature with examples in the following, before we derive the definition of typed attributed graphs.

An algebraic signature is a syntactical description of an algebra and consists of sorts (of attribute values) and operation symbols.

Definition 21 (Algebraic Signature)**.** *An algebraic signature* $\Sigma = (S, OP)$ *consists of a set* S *of sorts* s_i *and a set* OP *of n-ary operation symbols op on* S*, that is,* $OP \subseteq \{op : s_0 \dots s_i \dots s_n \longrightarrow s : 0 \leq i \leq n, s \in S, s_i \in S\}$

Remark To define constant symbols, we include the empty set $\emptyset$ in S. Hence, if $s_0 \dots s_n$ is an empty word, that is, $s_i \in \emptyset$ for $i = 0, \dots, n$, then $op : \longrightarrow s$ is called a constant symbol.

Example Let $S = \mathbb{N}$ be the set of natural numbers and

$OP = \{zero : \longrightarrow nat, succ : nat \longrightarrow nat, add : nat\ nat \longrightarrow nat, mult : nat\ nat \longrightarrow nat\}$

be a set of operations called constant, successor, addition, and multiplication operation, respectively. Then $\Sigma = (S, OP)$ is the data signature for natural numbers.

In the following, we define the concept of Σ-algebras also referred to as universal algebras. An algebra is a semantic model of a signature.

Definition 22 (Σ-Algebra)**.** *For a given signature* $\Sigma = (S, OP)$*,*
a Σ*-algebra* $A = ((A_s)_{s \in S}, (op_A)_{op \in OP})$ *is defined by:*

- *for each sort* $s \in S$*, a set* A_s*, called the carrier set;*
- *for a constant symbol* $c :\longrightarrow s \in OP$*, a constant* $c_A \in A_s$*;*
- *for each operation symbol* $op : s_1 \dots s_n \longrightarrow s \in OP$*,*
  a mapping $op_A : A_{s_1} \times \dots \times A_{s_n} \longrightarrow A_s$*.*

Example The $\mathbb{N}$-algebra D is the algebra defined over the signature of natural numbers:

- $D_{nat} = \mathbb{N}$
- $zero_D = 0 \in D_{nat}$
- $succ_D : D_{nat} \longrightarrow D_{nat},\ x \mapsto x + 1$
- $add_D : D_{nat} \times D_{nat} \longrightarrow D_{nat},\ (x, y) \mapsto x + y$
- $mult_D : D_{nat} \times D_{nat} \longrightarrow D_{nat},\ (x, y) \mapsto x \cdot y$

Example The STRING-algebra D is the algebra defined over the signature of stings, where strings are words over characters:

- $D_{char} = \{a, \dots, z, A \dots, Z, 0 \dots, 9\}$
- $D_{string} = D^*_{char}$
- $a_D = A \in D_{char}$ (for constant value a)
- $empty_D = \lambda \in D_{string}$ (for the empty string λ)
- $next_D : D_{char} \longrightarrow D_{char}$,
  $a \mapsto b, \dots, z \mapsto A, A \mapsto B, \dots, Y \mapsto Z, Z \mapsto 0, 0 \mapsto 1, \dots, 9 \mapsto a$ (for implying an ordering on characters)
- $concat_D : D_{string} \times D_{string} \longrightarrow D_{string},\ (s, t) \mapsto st$ (for the concatenation of two strings)
- $add_D : D_{char} \times D_{string} \longrightarrow D_{string},\ (x, s) \mapsto xs$ (for adding a character to a string)
- $first_D : D_{string} \longrightarrow D_{char},\ \lambda \mapsto A, s \mapsto s_1$ with $s = s_1 \dots s_n$ (for returning a character from a string)

Next, we define a special kind of algebra, called final algebra, which we use for the attribution of an attributed type graph.

Definition 23 (Final Algebra). *Given a signature $\Sigma = (S, OP)$, the final Σ-algebra Z is defined by:*

- $Z_s = \{s\}$ *for each sort* $s \in S$;
- $c_Z = s \in Z_s$ *for a constant symbol* $c :\longrightarrow s \in OP$;
- $op_Z : \{s1\} \times \cdots \times \{s_n\} \longrightarrow \{s\} : (s_1 \cdots s_n) \mapsto s$ *for each operation symbol* $op : s_1 \cdots s_n \longrightarrow s \in OP$.

Example The final algebra Z for the signature STRING is defined as follows:

- $Z_{char} = \{char\}$
- $Z_{string} = \{string\}$
- $a_Z = char \in Z_{char}$
- $empty_Z = string \in Z_{string}$
- $next_Z : Z_{char} \longrightarrow Z_{char}$, $char \mapsto char$
- $concat_Z : Z_{string} \times Z_{string} \longrightarrow Z_{string}$, $(string, string) \mapsto string$
- $add_Z : Z_{char} \times Z_{string} \longrightarrow Z_{string}$, $(char, string) \mapsto string$
- $first_Z : Z_{string} \longrightarrow Z_{char}$, $string \mapsto char$

Different algebras can implement the same signature, corresponding to different semantics. To analyze relations between algebras, we define homomorphisms.

Definition 24 (Algebra Homomorphism). *Given a signature $\Sigma = (S, OP)$ and Σ-algebras A and B, a homomorphism $h : A \longrightarrow B$ is a family $h = (h_s)s \in S$ of mappings $h_s : A_s \longrightarrow B_s$ such that the following properties hold:*

- *for each constant symbol* $c :\longrightarrow s \in OP$, *we have* $h_s(c_A) = c_B$ *where* $c_A \in A_s$ *and* $c_B \in B_s$
- *for each operation symbol* $op : s_1 \ldots s_n \longrightarrow s \in OP$ *it holds that* $h_s(op_A(x_1, \ldots, x_n)) = op_B(h_{s_1}(x_1), \ldots, h_{s_n}(x_n))$ *for all* $x_i \in A_{s_i}$.

Remark If, for $x_i \in A_{s_i}$ and $x_j \in A_{s_j}$, s_i and s_j are different sorts (e.g. nat and string), that is $A_{s_i} \neq A_{s_j}$, it follows that h_{s_i}, h_{s_j} are different mappings. For $s_i = s_j$, it follows that $h_{s_i} = h_{s_j}$.

An attributed graph is an E-graph combined with an algebra over a data signature. In the signature, we distinguish a set of attribute value sorts, where the corresponding carrier sets in the algebra can be used for the attribution.

Definition 25 (Attributed Graph). *Let $\Sigma = (S_D, OP_D)$ be a data signature with attribute value sorts $S'_D \subseteq S_D$. An attributed graph $AG = (G, D)$ consists of an E-graph G together with a Σ-algebra D such that $\bigcup_{s \in S'_D} D_s = V_D$.*

Remark Attributes graphs allow us to bestow graph nodes as well as edges with certain attributes. These attributes are also referred to as labels and hence, attributed graphs as a labelled graphs.

Example We give an example of an *attributed graph* defining an object-oriented class specification for a class named *Person*, depicted as E-Graph in Figure 2.4 (lower graphic). The *Person* Class contains two Fields, namely *name* and *age*, and according getter methods, namely *getName* and *getAge*. We make use of the following graphical notation: The solid nodes and arrows refer to the graph nodes V_G and edges E_G, respectively. The dashed nodes are the (used) data nodes V_D and dashed arrows represent node and edge attribute edges E_{NA} and E_{EA}.

Let $G_{targetModel}$ be an E-Graph with

- $G_{targetModel} = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}})$
- $V_G = \{class, field_1, field_2, method_1, method_2\}$
- $V_D = \{Person, name, age, getName, getAge\}$
- $E_G = \{fields_1, fields_2, method_1, method_2\}$
- $E_{NA} = \{cname, fname_1, fname_2, mname_1, mname_2\}$
- $E_{EA} = \varnothing$
- $source_G : E_G \to V_G : x \mapsto \left\{ \begin{array}{rcl} class & : & x \in \{fields_1, fields_2, methods_1, methods_2\} \end{array} \right.$
- $target_G : E_G \to V_G : x \mapsto \left\{ \begin{array}{rcl} field_1 & : & x = fields_1 \\ field_2 & : & x = fields_2 \\ method_1 & : & x = methods_1 \\ method_2 & : & x = methods_2 \end{array} \right.$
- $source_{NA} : E_G \to V_G : x \mapsto \left\{ \begin{array}{rcl} class & : & x = cname \\ field_1 & : & x = fname_1 \\ field_2 & : & x = fname_2 \\ method_1 & : & x = mname_1 \\ method_2 & : & x = mname_2 \end{array} \right.$
- $target_{NA} : E_G \to V_G : x \mapsto \left\{ \begin{array}{rcl} Person & : & x = cname \\ name & : & x = fname_1 \\ age & : & x = fname_2 \\ getName & : & x = mname_1 \\ getAge & : & x = mname_2 \end{array} \right.$
- $source_{EA} : E_G \to V_G : x \mapsto \varnothing$
- $target_{EA} : E_G \to V_G : x \mapsto \varnothing$

Furthermore, let D be an algebraic signature with
$D =$ STRING as defined in Example 2.4.2.

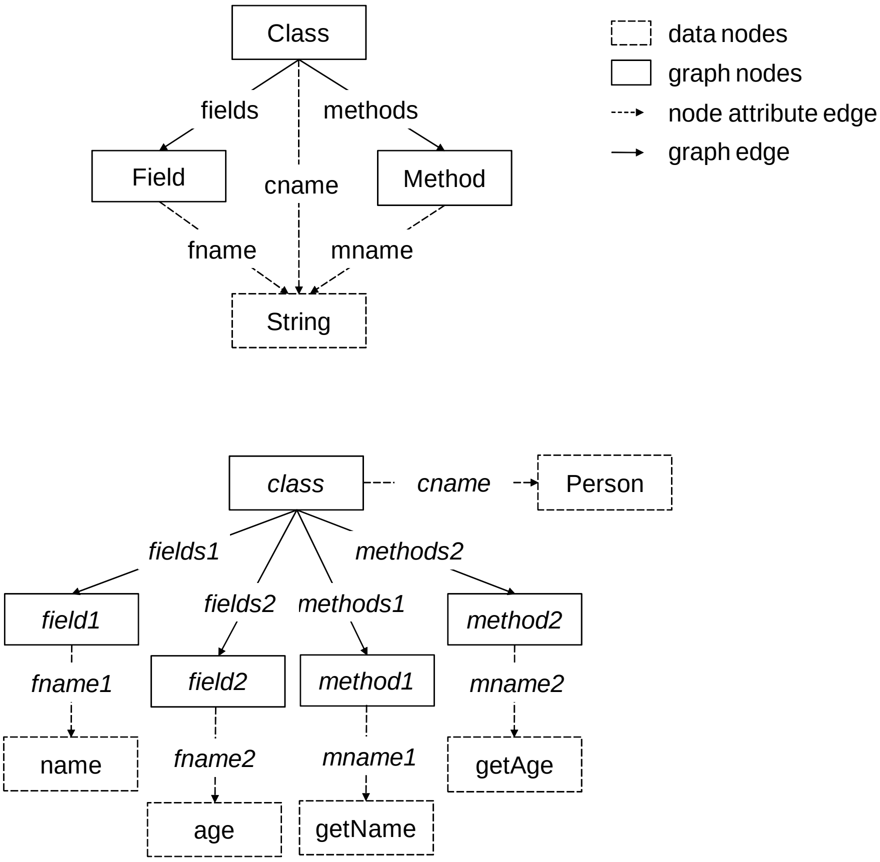


Figure 2.4.: Attributed Graph (lower) and Attributed Type Graph (upper)

Then $AG_{targetMetamodel}$ is an attributed graph with
$AG_{targetMetamodel} = (G_{targetMetamodel}, D)$.

For the typing of attributed graphs, we use a distinguished graph (called attributed type graph) attributed over the final Σ-algebra Z. This graph defines the set of all possible types.

Definition 26 (Attributed Type Graph)**.** *Given a data signature Σ, an attributed type graph is an attributed graph $ATG = (TG, Z)$, where Z is the final Σ-algebra.*

Example We give an example of an *attributed type graph* as formal definition of an object-oriented class. A class consists of Fields and Methods, owning a name attribute of type String including the class, depicted as E-Graph in Figure 2.4 (upper graphic). Essentially, this is the formal definition of the class *Person* in the example AG from above.

Let $G_{targetMetamodel}$ be an E-Graph with

- $G_{targetMetamodel} = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j\in\{G,NA,EA\}})$
- $V_G = \{Class, Field, Method\}$
- $V_D = D_{string}$
- $E_G = \{fields, methods\}$
- $E_{NA} = \{cname, fname, mname\}$
- $E_{EA} = \varnothing$
- $source_G : E_G \to V_G : x \mapsto \begin{cases} Class & : \quad x = fields \\ Class & : \quad x = methods \end{cases}$
- $target_G : E_G \to V_G : x \mapsto \begin{cases} Field & : \quad x = fields \\ Method & : \quad x = methods \end{cases}$
- $source_{NA} : E_G \to V_G : x \mapsto \begin{cases} Class & : \quad x = cname \\ Field & : \quad x = fname \\ Method & : \quad x = mname \end{cases}$
- $target_{NA} : E_G \to V_G : x \mapsto \begin{cases} string & : \quad x = cname \\ string & : \quad x = fname \\ string & : \quad x = mname \end{cases}$
- $source_{EA} : E_G \to V_G : x \mapsto \varnothing$
- $target_{EA} : E_G \to V_G : x \mapsto \varnothing$

Furthermore, let D be an algebraic signature with
$D =$ STRING as defined in Example 2.4.2.
Then $AG_{targetMetamodel}$ is an attributed graph with
$AG_{targetMetamodel} = (G_{targetMetamodel}, D)$.
Let Z be the final D-algebra.
Then $ATG_{targetMetamodel}$ is an attributed type graph with
$ATG_{targetMetamodel} = (AG_{targetMetamodel}, Z)$.

Definition 27 (Attributed Graph Morphism). *Given two attributed graphs $AG_1 = (G^1, D^1)$, $AG_2 = (G^2, D^2)$. An attributed graph morphism $f : AG^1 \longrightarrow AG^2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^1 \longrightarrow G^2$ and an algebra homomorphism $f_D : D^1 \longrightarrow D^2$.*

Definition 28 (Typed Attributed Graph). *A typed attributed graph (AG, t) over an attributed type graph ATG consists of an attributed graph AG together with an attributed graph morphism $t : AG \longrightarrow ATG$.*

Example Let t be a graph morphism $t : AG_{targetModel} \to ATG_{targetMetamodel}$ with

- $t = (t_{G,V_G}, t_{G,V_D}, t_{G,E_G}, t_{G,E_{NA}})$
- $t_{TG,V_G}(class) = Class$

- $t_{TG,V_G}(field_1) = t_{TG,V_G}(field_2) = Field$
- $t_{TG,V_G}(method_1) = t_{TG,V_G}(method_2) = Method$
- $t_{TG,V_D}(Person) = t_{TG,V_D}(name) = t_{TG,V_D}(age) = t_{TG,V_D}(getName)$
  $= t_{TG,V_D}(getAge) = string$
- $t_{G,E_G}(fields_1) = t_{G,E_G}(fields_2) = fields$
- $t_{G,E_G}(methods_1) = t_{G,E_G}(methods_2) = methods$
- $t_{G,E_{NA}}(cname) = cname$
- $t_{G,E_{NA}}(fname_1) = t_{G,E_{NA}}(fname_2) = fname$
- $t_{G,E_{NA}}(mname_1) = t_{G,E_{NA}}(mname_2) = mname$

Then $TAG_{targetModel}$ is a typed attribute graph over $ATG_{targetMetamodel}$ with $TAG_{targetModel} = (AG_{targetModel}, t)$.

2.4.3. Model representations through TAGs

According to [EEPT06] models and their corresponding metamodels can be expressed through TAGs. We will later use this formalism for model matching and import models (cf. Section 2.3) into an internal data model based on TAGs. Hence, we describe the representation of models and metamodels in terms of TAGs. The modelling conventions are based on the OMG Meta Object Facility (MOF) specification [Gro06].

Graph-based representation

We will represent models as attributed graphs and their corresponding metamodels as attributed type graphs. Thus, we need to show, how model or metamodel elements are mapped to nodes and data nodes in terms of a node mapping, secondly, how relationships between model or metamodel elements are mapped to edges, which we refer to as edge mapping. In the following, we describe these mappings for metamodels and model instances:

Node Mapping

- Metamodels: Regarding metamodels, each package, class and attribute (including operations) as well as an enumeration is mapped to a *graph node*. The labels are derived from the element names through attribution as described in Section 2.4.2. For example, in the upper part of Figure 2.5, the graph representation of a metamodel is given. The metamodel consists of a package P, which contains a class named A with an attribute called a. The graph representation thus consists of three corresponding *graph nodes*, labelled P, A and a. Furthermore, a data type is mapped to a *data node* and its label is set to the data type, e.g., data type *String* in Figure 2.4.

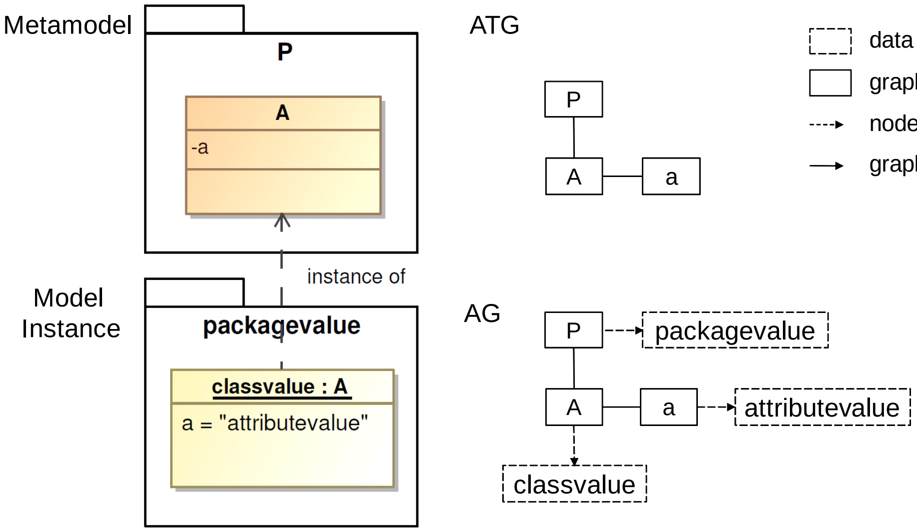
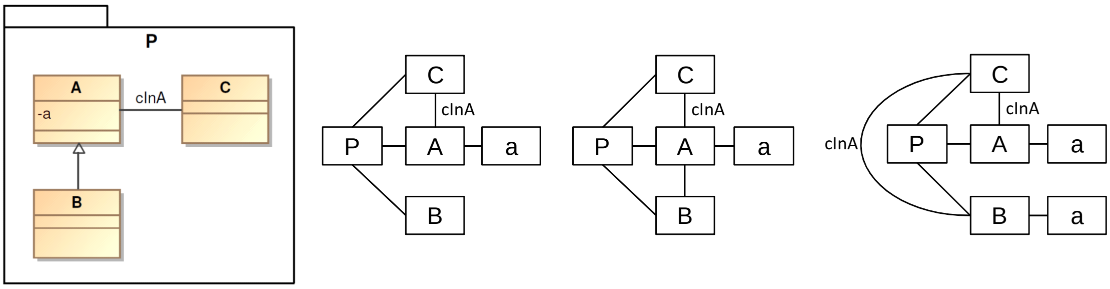


Figure 2.5.: Graph Representation of a Metamodel and Model Instance

- Model Instances: The node mapping follows the same principle as for metamodels above, except that no data type nodes occur. However, on instance-level, we need to map the values of model elements. These may be from a package, class, attribute etc. and are mapped to a *data node*. For a graph node, the label is derived from the name of the metamodel element to which the model element (represented as graph node) conforms to. Data nodes receive their labels from the values. For example, in the lower part of Figure 2.5, the graph representation of a model instance is described. On instance level, class A (of the depicted metamodel) is instantiated with the name *classvalue* contained in the package called *packagevalue*. Additionally, the attribute a is set to *attributevalue*. The graph nodes are labelled with their corresponding metamodel types, P, A and a. For each of these graph nodes, there exists a data node with a corresponding label derived from the value. Thus, the graph nodes P, A and a, refer to the data nodes *packagevalue*, *classvalue* and *attributevalue*, respectively.

Edge mapping

- Metamodels: Regarding the edge mapping three kinds of relationships need to be considered: References in general, containment references and inheritance. The inheritance relationship can be represented in two ways: a) through an edge representing the inheritance relation or implicitly by copying all inherited members into the corresponding subclasses. We give more insight into these representations in the following remark. References and containment can be mapped onto individual edge types. All three relationships correspond to graph edges, whereas an edge between a data node (due to a data type node) and graph node is represented as a *node attribute edge*.



Metamodel a) No Inheritance b) Inheritance Edges c) Flattened Inheritance

Figure 2.6.: Different Representations of a Inheritance Relation

- Model Instances: On instance-level, we consider reference and containment relationships, which are mapped onto separated edge types corresponding to *graph edges*. We additionally use an *node attribute edge* for the relationship between instances and their values, e.g., in the lower graph of Figure 2.4 between the graph node *class* and data node *Person*.

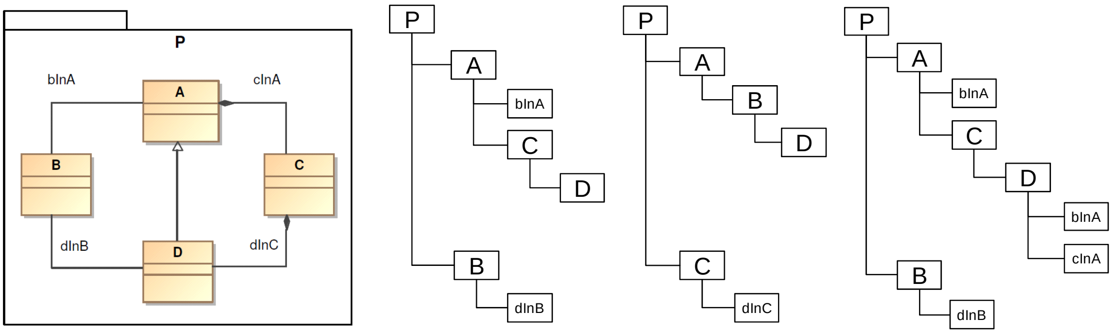
Remark A mapping between a metamodel and a graph is not unique due to the different representations of inheritance relations. Apart from the two above-mentioned options, alternatively the inheritance relationship can be neglected. In Figure 2.6 an example metamodel is given with three different representations of the inheritance relationship. The metamodel contains three classes, A, B and C, where A and B are related by an inheritance relation and A is related to C in terms of cInA. Finally, A includes an attribute, a.

Tree-based representation

Numerous matching techniques work more efficiently on a tree, for example, the parent, children, or leaf matcher described in Section 7.2.2. Thus, we need to consider a mapping from models (or metamodels) to trees. A tree is a special class of a graph with the constraint that each node is part of a parent-child relationship and the graph has a distinguished node, called *root* node. Formally, we define a tree as follows.

Definition 29 (Tree)**.** *A directed graph G is called a directed* **tree***, if the following holds:*

- *G has no simple cycles. A simple cycle is a cyclic traversal of nodes, where each node is traversed once.*
- *G is connected.*
- *Each node has at most one incoming edge.*



Metamodel a) Containment b) Reference c) Flattened Containment

Figure 2.7.: Different Tree Representations of a Metamodel

Regarding the mapping of a metamodel to a tree, we follow the approach proposed in [VIR10]. The node mapping works the same as for the graph-based representation. Additionally, certain references need to be mapped to nodes, the reason being, to resolve the graph into a tree structure. For the edge mapping, we again consider references in general, containment references and inheritance. To map these relationships onto a parent-child hierarchy of a tree, relates to the problem of mapping a graph onto a tree. To solve this, the algorithm of Kruskal [Kru56] is followed to find the *Minimal Spanning Tree* (MST) of a metamodel, being the closest related approach amongst others. A MST is a subgraph of a graph connecting all nodes, thus forming a tree, where the sum of the costs of all edges is minimal.

The mapping onto a tree structure starts with a package and the elements contained. In order to compute the MST first all references or containment references are handled as edges of a graph. Depending on the type of relationship and the choice of flattening for inheritance, different MST are computed. We give three different MSTs in the following example, derived from [VIR10]. In Figure 2.7 a), a containment MST is presented. In this case, the containment relations lead the way to structuring the MST. Thus, P contains A, while D is contained in C and C in A. Furthermore, P contains B, due to the package containment. In case b), the references direct the structure of the MST from A to B to D. In both cases, a) and b), the remaining references are mapped to elements, e.g., in Figure 2.7 a), dInB of D. Additionally, inheritance relations need to be considered. As explained above, the import may neglect these relations, or copy all inherited members into the corresponding subclasses. In the latter case, therefore, the containment and reference MST are enhanced with these inherited members. We show this in case c) as flattened containment. The inherited members are bInA and cInA.

For models, the mapping to trees is calculated analogously to metamodels, except that no inheritance relations need to be considered. Thus, option a) and b) hold.

2.5. Facets

The inception of facets stems mainly from the interest in classification.

Definition 30 (Classification). *A classification is the systematic arrangement in groups or categories according to established criteria. Commonly, these groups are referred to as classes and are arranged hierarchically*[1] *[MW05].*

Classifications may be differentiated by their structure into hierarchies or polyhierarchies.

Definition 31 (Hierarchy versus Polyhierarchy). *A classification structured as a hierarchy (also called monohierarchy) possesses a tree structure, such that each class owns one and only one parent node. In the case of a polyhierarchy, a class may have more than one parent node.*

The early fields of application for facets are library and information science, dating back to the work of Shiyali Ramamrita Ranganathan[2] [Tun09]. Ranganathan recognized the challenges inherent in using a single taxonomy to represent a diverse collection of knowledge [Tun09]. We give a formal definition of a taxonomy.

Definition 32 (Taxonomy). *A taxonomy refers to a hierarchical classification.*

Remark Mathematically, a taxonomy is a tree structure of classifications for a given set of objects.

The above-mentioned challenges by Ranganathan concern the rigidity of taxonomical schemes and unstructured indexes [Tun09], as listed below. Let us assume an exemplary taxonomy on a book catalog in the following. The taxonomy classifies books according to geographical, literary type and temporal classes as depicted in Figure 2.8 on the left side. Thereby, books are firstly classified according to their geography distinguished by the classes of the continents, secondly their literary genre of poetry, drama and prose and lastly, their temporal occurance in the 17th to 19th century.

- **Rigidity of taxonomical schemes**: The key property of a taxonomy requests that every node in the taxonomy has a unique path to the root node. Therefore, every class exists uniquely within a taxonomy. This has implications on the design of taxonomies and may be too ”rigid”, as described in [Tun09], when classifying compound classes, that is, classes that are compound of existing classes in the taxonomy. For example, let us assume a compound class [3] *American Poetry of the 19th Century.* The designer of the classification has to decide, whether this class belongs under *America* and then *Poetry* and *19th Century*, or first under the

---

[1] adapted from Merriam-Webster’s dictionary

[2] Ranganathan, an Indian mathematician and librarian, lived from 1892 to 1972.

[3] extracted from [Pri00b]

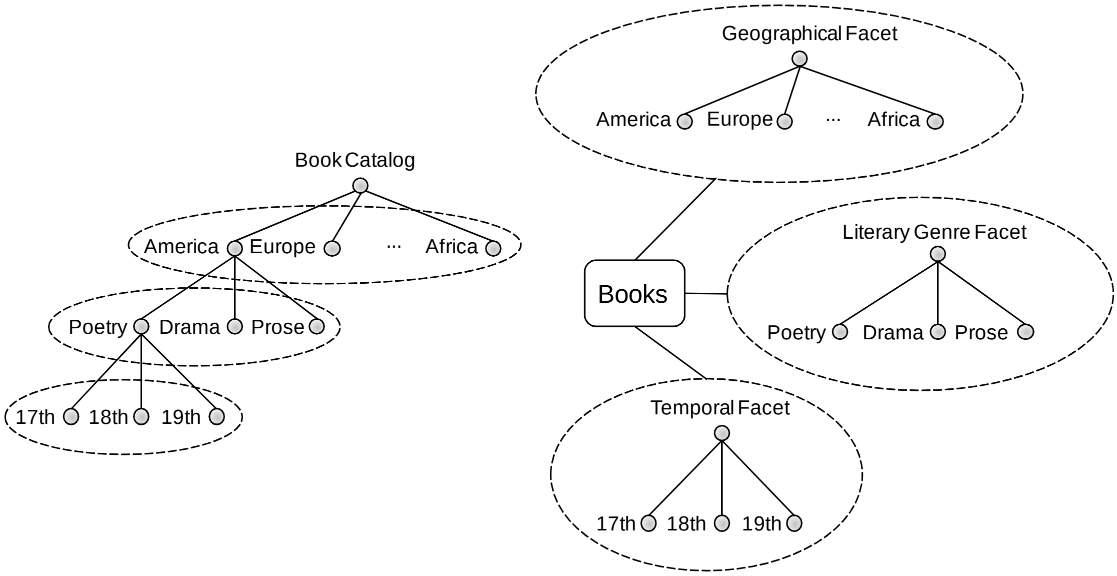


Figure 2.8.: Taxonomy and Facets for Book Catalog

temporal class, then the geographical and last the literary genre class [Pri00b], or any other permutation of the classes. In general, $3! = 6$ combinations are possible. Because of the key property mentioned above, it is impossible to classify *American Poetry of the 19th Century* as a child of *America*, *Poetry* and *19th Century* at the same time, thus the rigidity to incorporate compound classes at multiple points.

A work-around to the above shortcoming is a polyhierarchy. Yet, polyhierarchies are difficult to maintain, particularly, when having to maintain potential cross-references, when moving a node with its subtree [Tun09].

- **Scalability of taxonomical schemes**: When adding a new class to a taxonomy, this necessitates the adaption of all other classes with corresponding child class occurrences. For example, when adding a new temporal class to the taxonomy in Figure 2.8, the *Poetry*, *Drama* and *Prose* class, would have to be adapted with the new temporal child node, effectively for every geographic class. Thus redundant structures in the sense of common subtaxonomies result in effort during the maintenance of taxonomies.

Ranganathan introduced the faceted classification scheme, which he called *colon classification* in 1933 [Ran33] (first edition), providing a more elegant solution to the above-mentioned challenges. Revisiting the example above, in a faceted classification scheme including three facets, that is, a geographical, literary genre and temporal facet, as depicted in Figure 2.8, the design decision on incorporating the class *American Poetry of the 19th Century* does not have to be made. A book from the class *American Poetry of the 19th Century*, would be classified under *America* in the geographical facet, under

19th Century in the temporal facet and lastly, under *Poetry* in the literary genre facet. The facets would then be postcombined at the time of retrieval.

In summary, facets tackle the challenge of a) rigidity and b) scalability inherent to taxonomies. In the case of a) a faceted classification has the flexibility to incorporate compound classes at multiple points, which Ranganathan called ”hospitality at many points”, due to the post-combination of facets as described above.

In the case of b) facets factorize the hierarchical classification, and thus simplify the classification scheme, as to factor out common subtaxonomies as facets, or ”re-occurring features” as Priss [Pri08] states. In factoring out these common sub-taxonomies, redundancy is avoided in the classification scheme, thus simplifying the scheme. For example, a new temporal class from our example, would only have to be added to the temporal facet itself, leaving all remaining classes unaffected.

We next look into different definitions of the notion facet.

Definition 33 (Facet)**.** *Facets are viewpoints or aspects of classification schemes [Pri00b].*

In computer science, facets are used as means of structuring class hierarchies, where the notion of *facet* usually refers to ”subdivisions of a class hierarchy based on re-occurring features” [Pri08]. A more formal definition given by Priss is:

Definition 34 (Facet)**.** *Facets are relational structures consisting of finite sets of units, relations and/or other (constituent) facets combined for a certain purpose [Pri00a].*

According to Priss [Pri08], the notion of facets has several slightly different definitions and is used under different names in information processing disciplines. Therefore, we will state the characteristics a facet owns in the context of our work:

Definition 35 (Facet)**.**

1. *Facets own a model which consists of a hierarchy of faceted values and/or facets.*
2. *Facets may form facet hierarchies.*
3. *Every facet model contains a faceted value denoting an unknown faceted value.*
4. *Facets are independent of each other because any facet can be combined with any other facet and modifying a single facet does not have impact on other facets.*

Remark The difference between facet and faceted values lies in the level of abstraction. Facets are abstract, whereas faceted values are concrete as per choice of their implementation.

Remark Without using facets all possible combinations of faceted values are precombined, whereas with faceted classifications the combination of faceted values only takes place when classification takes place and are thus post-combined.

In the following, we summarize the advantages of facets resp. faceted classifications.

- Flexibility: Faceted classifications allow for more flexibility to classify compound classes, due to the post-combination of faceted values at the time of retrieval.
- Simplification: Facets factorize inheritance hierarchies and thus simplify the classification scheme.
- Extensibility: Faceted classifications do not require the structure of classification to be fixed a priori. The classification may be adapted by adding facets and/or faceted values without having to maintain redundant structures (common subtaxonomies factorized out into independent facets) and without impact on other facets.

3

Requirements Analysis

In the following chapter, we are going to present our requirements analysis. The underlying approach of this analysis is a problem analysis approach called *Zielorientierte Projektplanung* (ZOPP) [HG97] or *Objectives-oriented Project Planning* in English, promoted by the Deutsche Gesellschaft für Technische Zusammenarbeit (German Technical Cooperation). We start off by giving a motivating example for our work in Section 3.1. Then, we span the scope of our problem domain by defining three categories of trace link generation in Section 3.2, on the basis of which we derive our problem definition and corresponding requirements in Section 3.3.

3.1. Motivating Example

In the following section, we provide a motivating example for the problem definition in Section 3.2 underlying this work.

A software development process is commonly defined through the standard development phases, namely, requirements analysis, design, implementation and test phase [Int08] as portrayed in Figure 3.1, showing an excerpt of the SAP product development process, called Product Innovation Life Cycle [PRG07]. Within all phases artefacts are created, yet, when deploying an MDSD process, the advantage of generating artefacts *automatically* may be leveraged.

Given a project manager conducting a change impact analysis (cf. Section 2.2.1), the following scenario is conceivable. The project manager needs to analyse the effects of changing a software requirement as part of a customer request. Hence, he needs to analyse the impact this change will have on all development phases and understand,

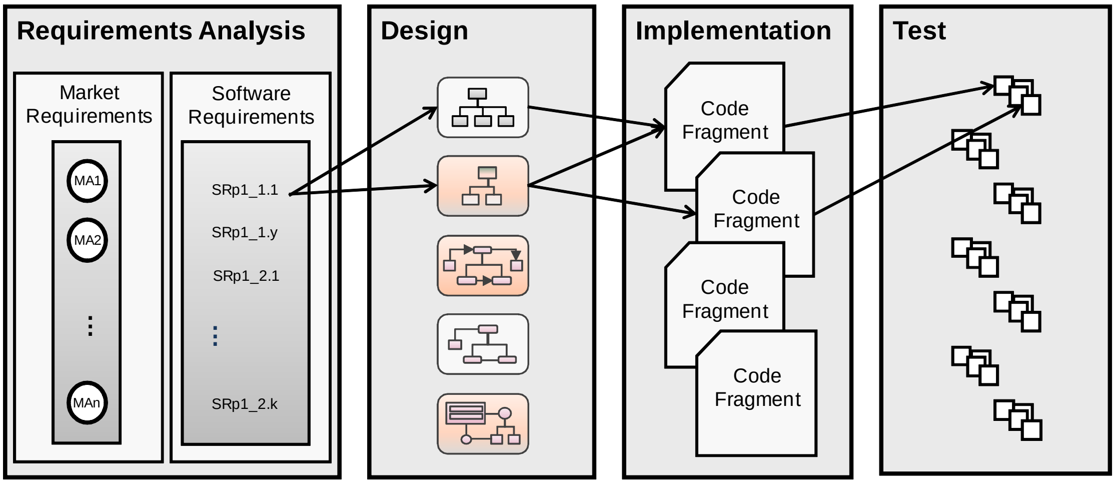


Figure 3.1.: Software Development Process

which artefacts are affected and need to be adapted. For this, the knowledge on all logical dependencies of artefacts is necessary. Trace links provide this kind of information as portrayed in Figure 3.1 through the arrows, tracking the dependencies of a certain software requirement to artefacts of the design, implementation and test phase, thus, spanning the scope of artefacts potentially in need of adaptation. To an MDSD process, the same conditions apply, with regard to the necessity of trace links providing the knowledge on artefact dependencies as motivated through other possible traceability use cases listed under Section 2.2.1.

This work aims at generating trace links automatically for an MDSD process. The problem definition of this domain and of our work is described in the following section.

3.2. Problem Definition

Using the integral model mapping of model transformations, to generate trace links automatically, eases the task of creating and maintaining trace links. In contrast, when done manually, this entails high development costs [ARNRSG06]. Yet, the automatic trace-link generation still holds a wide range of open challenges, upon which we shall elaborate in this section.

We start off to derive different possibilities on trace link generation, being relevant and necessary to MDSD in order to define the scope of our work. In this way, we derive three categories of trace link generation, as summarized in Figure 3.2:

1. **Generation of Trace Links through Transformation**: The transformation engine implicitly generates a model mapping directed through the transformation program at model transformation runtime. The first category uses this integral

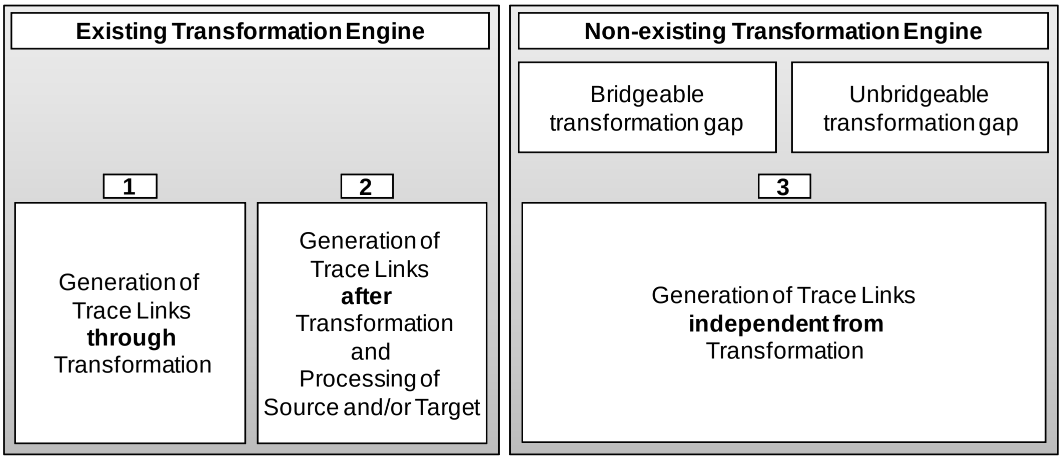

Existing Transformation Engine
Non-existing Transformation Engine
Bridgeable transformation gap
Unbridgeable transformation gap
1
2
3
Generation of Trace Links **through** Transformation
Generation of Trace Links **after** Transformation and Processing of Source and/or Target
Generation of Trace Links **independent from** Transformation

Figure 3.2.: Categories of Trace Link Generation

model mapping to derive trace links in parallel to the execution of a model transformation. Approaches from the classes of implicit and explicit trace-link generation (cf. Section 2.2.2) are examples of the first category.

2. **Generation of Trace Links after Transformation and Processing of Source and/or Target**: Post-processing of the source and target of a transformation is a common practice in MDSD [SV06], for model-to-model as well as for model-to-text transformations. For example, artefacts that cannot be generated automatically and thus have to be added manually, or in general due to the evolution of artefacts. While changing the source and/or target after transformation execution and the resulting trace link generation from the first category, an update of traceability data may be necessary. Thus, the generation of trace links after transformation execution needs to be addressed.

   Re-executing a model transformation to gather traceability data as in the first case may not always be a viable option, when generated artefacts have cross-dependencies to others, or manually added artefacts are unprotected, i.e., may be overwritten.

3. **Generation of Trace Links independent from Transformation**: The first two cases are based on the existence of a model transformation, i.e., the generation of trace links is dependent on the use of a transformation engine. Since this does not generally hold for the domain of MDSD, as described below, the third category covers the generation of trace links, while assuming the non-existence (or inaccessibility) of a transformation engine.

The latter cases relate to either of two subcases (cf. Figure 3.2). We characterise these through the transformation program:

- **Bridgeable transformation gap**: Writing a transformation program is possible (called bridgeable transformation gap in Figure 3.2), yet the transformation engine cannot be leveraged for the sake of trace-link generation. An example is, if the transformation engine is proprietary or generally a third party component and thus, is inaccessible. Further examples are, if a transformation is implemented manually (e.g., in Java), or development is model-based instead of model-driven as often the case at SAP, implying a non-existing transformation engine.
- **Unbridgeable transformation gap**: In this case, it is impossible to write a transformation program, since the difference in the level of abstraction of a given source and target is too great to be able to be bridged through a transformation, while still preserving the semantics, e.g., when mapping features to design models. We refer to this case as unbridgeable transformation gap in Figure 3.2.

Thus, we have three categories of trace link generation, as portrayed in Figure 3.2. Next, we look into the issues and challenges inherent to the above categories of trace link generation as part of our requirements analysis.

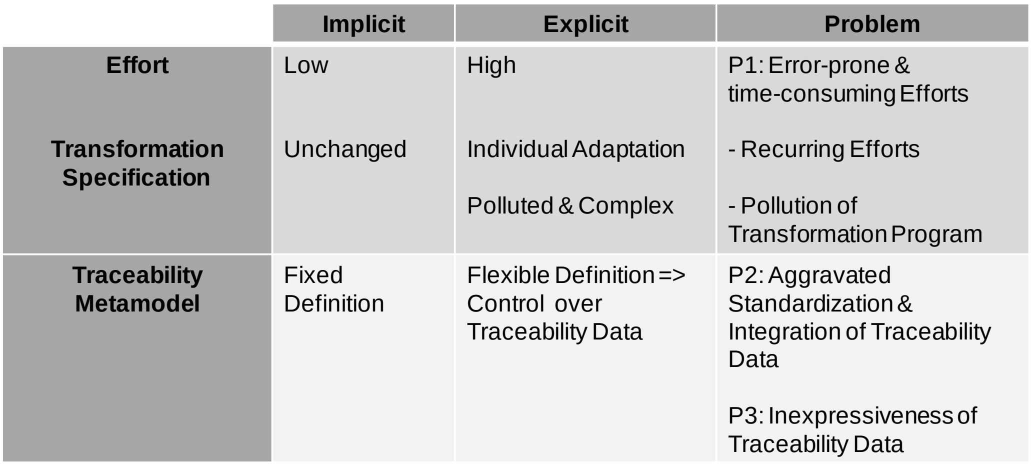
3.3. Requirements Analysis

In this section, we define the range of problems, we wish to tackle in this thesis. Based on these, requirements are derived for our approach. This derivation process is followed with respect to the first category in Section 3.3.1 and the second and third category in Section 3.3.2. The abbreviated notation for requirements and problems is R and P, respectively.

3.3.1. Requirements: Category 1

Based on the dichotomy of generation classes in Section 2.2.2, we extract the main issues and challenges we wish to tackle with our approach regarding the first category. As tabularized in Figure 3.3, we identified three main problems (listed in the fourth column), which effectively stem from the implicit as well as explicit generation class (characterised in the second and third column, respectively):

Problem 1 (**Error-Prone and Time-Consuming** (reoccurring) **Efforts**)**.** *While the implicit trace link generation class provides a default traceability solution, the explicit generation class requires manual effort to achieve a traceability solution. As this task is generally done manually, it is likely to be error prone and time consuming. Moreover, this effort is* reoccurring *for each transformation. Furthermore, the transformation*



	Implicit	Explicit	Problem
Effort	Low	High	P1: Error-prone & time-consuming Efforts
Transformation Specification	Unchanged	Individual Adaptation Polluted & Complex	- Recurring Efforts - Pollution of Transformation Program
Traceability Metamodel	Fixed Definition	Flexible Definition => Control over Traceability Data	P2: Aggravated Standardization & Integration of Traceability Data P3: Inexpressiveness of Traceability Data

Figure 3.3.: Problem Definition

program is not separated from traceability-specific transformation rules, thus potentially leading to program pollution *with the risk of increasing the above-mentioned efforts.*

The following problems relate to the data-integration problem of existing traceability solutions.

Problem 2 (**Aggravated Standardization and Integration of Traceability Data**)**.** *The class of implicit trace-link generation provides an integrated traceability solution, which does not account for the adaption of traceability data. On the contrary, the class of explicit trace-link generation does. This consequential discrepancy results in the problem of not generally having traceability data conforming to a unified traceability metamodel (not even with regard to a generation class itself).*

Since each traceability solution is tightly coupled with a certain transformation language, the reusability of existing traceability solutions is limited to the language used and regarding the explicit trace link generation class to the individual transformation itself. As more than one language may potentially be used in the same model transformation chain, or project and given the above-mentioned poor reusability of existing traceability solutions, the standardization of traceability data is aggravated. Furthermore, this shortcoming results in the problem of poor integration of the above-mentioned heterogenous traceability data.

In summary, the aggravated standardization and integration of traceability data limits the exchangeability of traceability data among MDSD approaches, needed for reasoning over traceability data and more importantly, the conduction of traceability scenarios.

Problem 3 (**Inexpressiveness of Traceability Data**)**.** *Since the kind of traceability data to be collected is dependent on the actual traceability goal, (i.e., traceability scenario) [KP02], a unified traceability metamodel (in accordance with Problem 2) with a fixed*

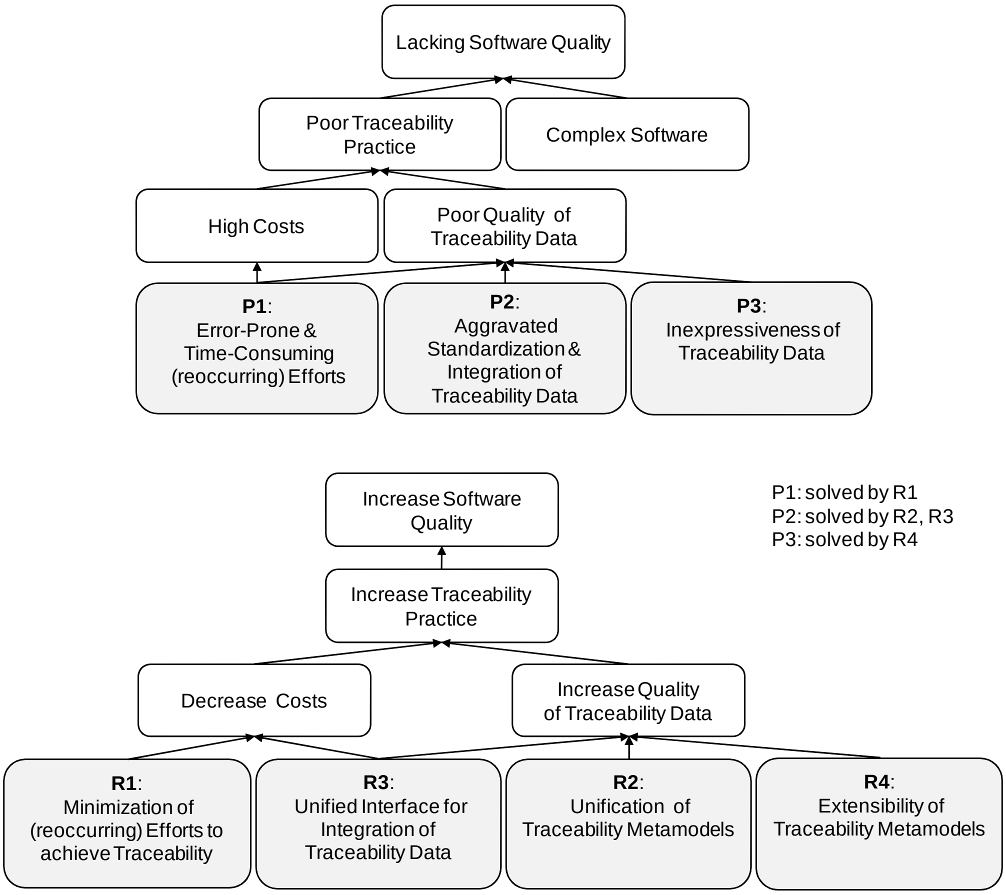

Lacking Software Quality
Poor Traceability Practice
Complex Software
High Costs
Poor Quality of Traceability Data
P1: Error-Prone & Time-Consuming (reoccurring) Efforts
P2: Aggravated Standardization & Integration of Traceability Data
P3: Inexpressiveness of Traceability Data
Increase Software Quality
P1: solved by R1
P2: solved by R2, R3
P3: solved by R4
Increase Traceability Practice
Decrease Costs
Increase Quality of Traceability Data
R1: Minimization of (reoccurring) Efforts to achieve Traceability
R3: Unified Interface for Integration of Traceability Data
R2: Unification of Traceability Metamodels
R4: Extensibility of Traceability Metamodels

Figure 3.4.: Problem and Objective Hierarchy

definition does not necessarily account for an adequate data expressiveness with respect to all possible traceability scenarios. Thus, the problem of traceability-data inexpressiveness needs to be addressed.

Following the ZOPP-approach, we arrange the identified problems in a hierachy of superproblems and subproblems as shown at the top of Figure 3.4. The arrow direction points out the relation between subproblem and superproblem, where the arrow's end points to the subproblem causing the superproblem pointed to by the arrow's head. The goal of this effort is to present the globally defined superproblem of the aforementioned problems. The manual effort required to achieve a traceability solution leads to the problem of *error proneness* and *time consumption*, additionally *reoccurring manual efforts* due to the adaption of individual transformations. This problem advances to the superproblem of *high costs*. Moreover, the error-prone factor results in the superproblem of *poor quality of traceability data* in conjunction with the *inexpressiveness* and *aggravated standardization and integration* of traceability data. Furthermore, the high costs and poor data quality result in *poor traceability practice*, which is climaxed together with the rising *complexity* of software to a lack in *software quality.* By following the ZOPP-approach, we next derive the list of requirements for the first category, where one or more requirements solve a corresponding problem:

- Problem 1 is solved by Requirement 1.
- Problem 2 is solved by Requirement 2 and Requirement 3.
- Problem 3 is solved by Requirement 4.

In accordance with the above problem hierarchy, we present the corresponding objectives (requirements) hierarchy to the bottom of Figure 3.4. The root-cause semantics of the arrow applies analogously to the objectives hierarchy.

Requirement 1 (**Minimization of efforts to achieve traceability**)**.** *Likely error-prone and time-consuming factors of manual efforts to achieve a traceability solution are to be reduced, as in the case of the explicit trace link generation class. Furthermore, traceability is seen as a separate concern and should not lead to the pollution of transformation programs.*

Requirement 2 (**Unification of traceability metamodels**)**.** *Traceability metamodels need to be unified to have a formal definition of traceability data, that is, to standardize traceability data.*

Apart from the unified traceability metamodel from Requirement 2, which is concerned with a unified data definition, we additionally require a unified interface for transformation approaches to be able to exchange data conforming to the unified traceability metamodel.

Requirement 3 (**Unified Interface for Integration of Traceability Data**)**.** *We require a unified integration interface for arbitrary transformation approaches, describing, how the transformation approach is featured with a traceability solution. As a result, the traceability solution needs to conform to the unified traceability metamodel. Moreover, the interface needs to account for the integration of existing traceability solutions.*

Requirement 4 (**Extensibility of traceability metamodels**)**.** *To guarantee for an adequate traceabililty-data expressiveness mandated by traceability scenarios (cf. Section 2.2.1), an extensibility mechanism on the basis of a unified traceability metamodel is required. This extensibility mechanism accounts for case-specific traceability metamodels with regard to the traceability scenarios. Essentially, this mechanism needs to hold for the specification of granularity and scope as well as the trace-link semantics with respect to a certain traceability scenario, as argued in Section 2.2.2.*

3.3.2. Requirements: Category 2 and 3

In the following, we present the requirements analysis due to the second and third category. In both categories the transformation engine is either non-existing, or inaccessible and thus, cannot be leveraged for the sake of trace link generation through the transformation mapping as in the first category. This leads to the problem of lacking traceability data.

Problem 4 (**Lacking Traceability Data**). *Lacking traceability data arises due to non-existing or inaccessible transformation engines.*

In tackling the above problem for the second and third category, we seek a solution of finding correspondences between arbitrary source and target models, yet independent from a transformation engine.

Naturally, this solution will include the handling of a traceability metamodel.

Requirement 5 (**Unification and extensibility of traceability metamodels**). *We require the conformance to traceability data standardization and expressiveness to meet requirement 2 and 4 holistically for our solution on the three categories of trace link generation.*

The following two requirements stem from the problem domain of a traceability solution requiring manual effort.

Requirement 6 (**Automation of achieving Traceability without Transformation Engine**). *In analogy to requirement 1 and for the same reasons to minimize manual efforts, we require an automatic mechanism (at least to a high degree) for the generation of trace links without the use of a transformation engine.*

Requirement 7 (**Language-independence of Traceability Solution**). *For the sake of genericity, we require the traceability solution to be independent from the modelling language used to specify source and target models. With a broad language applicability the effort to achieve a traceability solution is minimized.*

Although the problem hierarchy from the upper part of Figure 3.4 holds for the first categoy's problems, it may be applied to the other categories as well. Essentially, requirement 5 is the objective of the two bottom right problems (P2 and P3) in Figure 3.4 regarding the inexpressiveness and standardization of traceability data, while requirements 6 and 7 are aligned with the problem domain of manual efforts on the bottom left (P1), where automation and language-genericity reduce this effort.

To summarize, we present the corresponding objectives hierarchy of our approach in Figure 3.5 by incorporating the requirements of all categories (omitting duplicates), where the overall goal of our approach is to increase software quality.

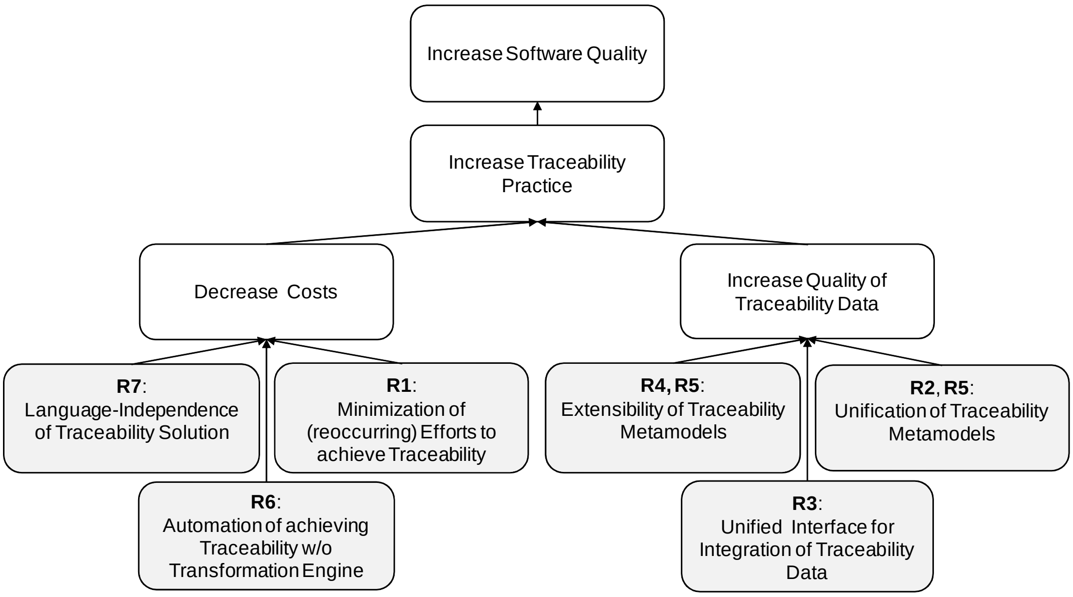

Increase Software Quality
Increase Traceability Practice
Decrease Costs
Increase Quality of Traceability Data
R7: Language-Independence of Traceability Solution
R1: Minimization of (reoccurring) Efforts to achieve Traceability
R4, R5: Extensibility of Traceability Metamodels
R2, **R5**: Unification of Traceability Metamodels
R6: Automation of achieving Traceability w/o Transformation Engine
R3: Unified Interface for Integration of Traceability Data

Figure 3.5.: Objective Hierarchy

3.4. Big Picture

In the following, we sketch the big picture of our approach to the problem definition given in Section 3.2. The key-enabling techniques of our approach are referred to in Figure 3.6 on the lower level. The arrows point out the mapping of the proposed techniques to the problem space defined through the three categories of trace link generation. To tackle the challenges as per problem definition of the first category of trace link generation, we propose a novel way of constructing connectors inherent to the extraction of traceability data from model transformations. In this sense, the connector essentially is seen as an abstraction from arbitrary transformation engines and traceability engines. Thereby, we identify two classes of connectors either working as a *blackbox*, or *invasively* by augmenting the transformation engine for the purpose of trace link generation.

Secondly, for the second and third category, we propose *model matching* techniques from the field of ontology alignment and schema matching [CSH06, RB01] to generate trace links for arbitrary source and target models. The promising idea of this technique is its potential in the automation of trace link generation without executing a model transformation, which is restricted in the second and third category and thus cannot be leveraged for the sake of trace link generation.

Both above-mentioned techniques of trace link generation are incorporated into a framework, called *generic traceability framework*.

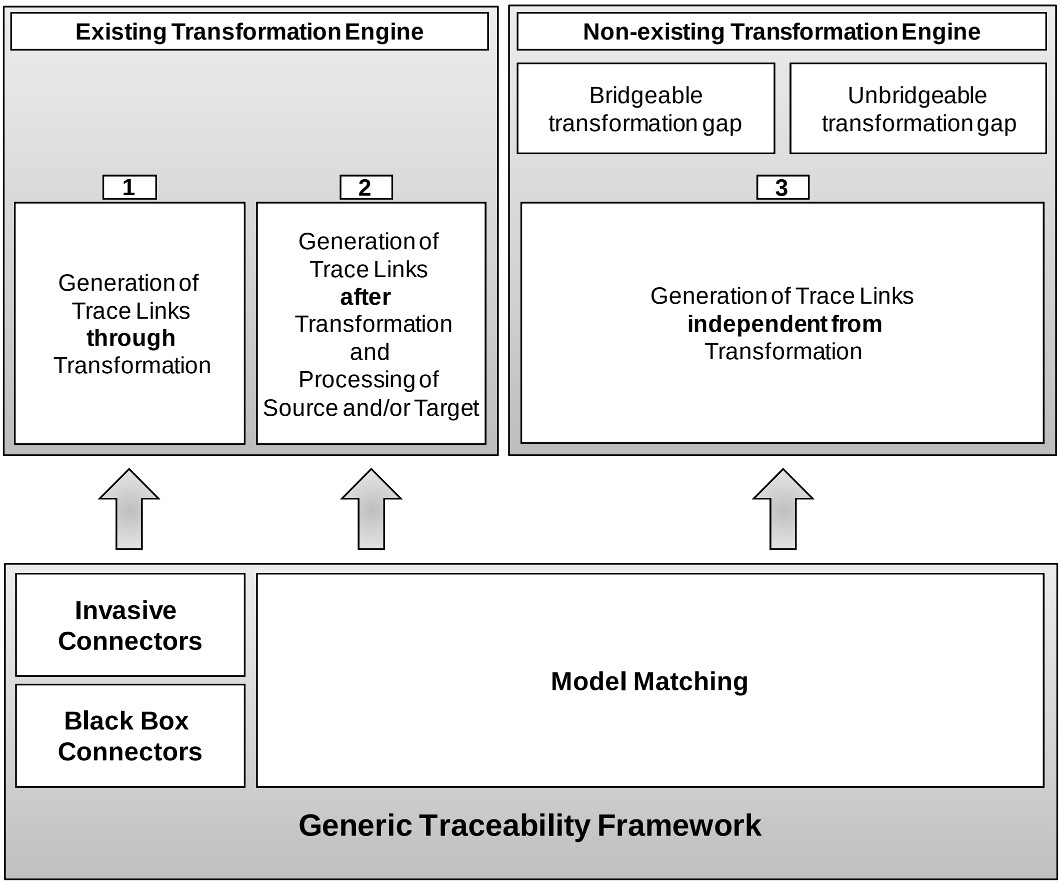

Existing Transformation Engine
Non-existing Transformation Engine
Bridgeable transformation gap
Unbridgeable transformation gap
1
2
3
Generation of Trace Links **through** Transformation
Generation of Trace Links **after** Transformation and Processing of Source and/or Target
Generation of Trace Links **independent from** Transformation
Invasive Connectors
Black Box Connectors
Model Matching
Generic Traceability Framework

Figure 3.6.: The Big Picture

4

Generic Traceability Framework (GTF)

The following chapter gives an overview of our approach, realised as the *Generic Traceability Framework* (GTF). A detailed description of the GTF is provided in the next three chapters, that is, Chapter 5 to Chapter 7.

We begin this chapter with an architecture of the GTF in Section 4.1. Thereafter, we elaborate on the two possibilities of trace link generation of our approach, the connector-based approach on invasive and black-box connectors in Section 4.2 and the approach on model matching in Section 4.3.

4.1. Architecture

The GTF can be instantiated to arrive at a traceability solution accounting for the three categories of trace link generation. In the next section, we give a high-level overview on the architecture of the GTF. In doing so, we reflect the three categories to show, how they are realised.

Regarding category 1, we require the GTF to augment arbitrary model transformation approaches with a traceability mechanism. To rely on existing traceability solutions (cf. Section 2.2.2), our approach does not implement another transformation language. Thus, the aim is to consolidate the benefits of implicit and explicit trace-link generation and to tackle their disadvantages and challenges as motivated in the requirements analysis in Section 3.3.1. Essentially, the GTF is based on a *Generic Traceability Interface* (GTI), which provides the connection point for arbitrary transformation engines, as depicted in Figure 4.1. In this case, the interface supplies the engineer with an API to connect his transformation engine to a traceability engine in terms of a corresponding connector. As a result, the transformation engine is featured with traceability functionality.

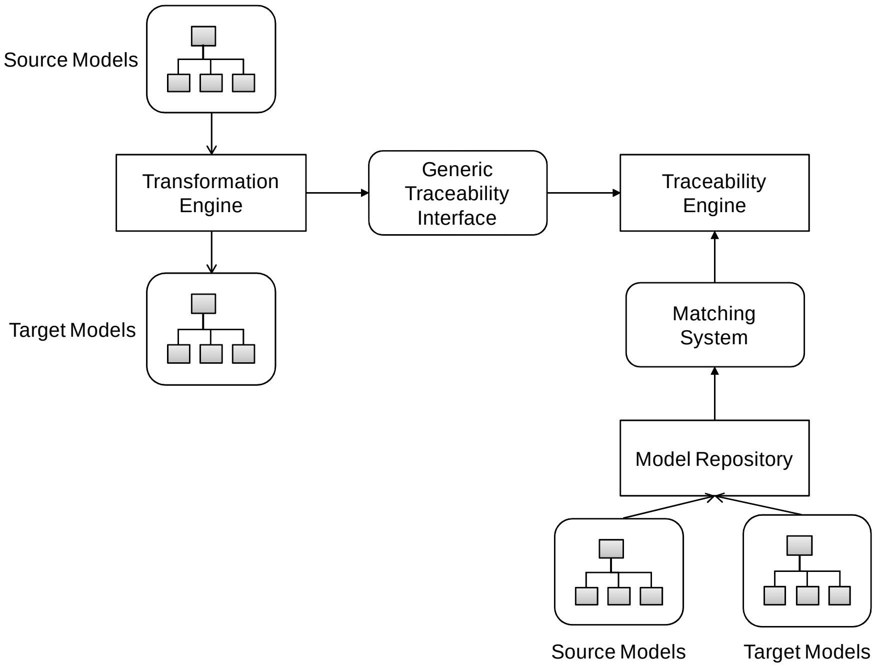


Figure 4.1.: Architecture Overview

Secondly, our architecture entails a matching system as depicted in Figure 4.1 in realisation of the requirements of category 2 and 3. The matching system receives a certain source and target model upon which the matching system works. In doing so, the system finds a potential mapping between source and target model elements on the basis of which trace links may be generated and written to the traceability engine.

Thirdly, our approach is based on a generic traceability metamodel. Essentially, the traceability data exchangeable through the GTI between arbitrary transformation and traceability engines conforms to this traceability metamodel. The same applies to the exchange of traceability data between the matching system and traceability engine. In Figure 4.2 the data exchangeability is indicated by circular arrows, whereas the instance-of relation is reflected through the dotted arrows with a corresponding ”instance-of” label. All categories necessitate a traceability metamodel, that is, require a standardized handling of traceability-data modelling in alignment with Requirement 2 and Requirement 5.

We go into more details of our approach in the following chapters, in accordance with Figure 4.3, depicting a road map on the chapters describing our concept. In Chapter 5, we explain our view on how traceability data is modelled through a generic traceability metamodel. Furthermore, Chapter 6 and Chapter 7 are dedicated to the generation of

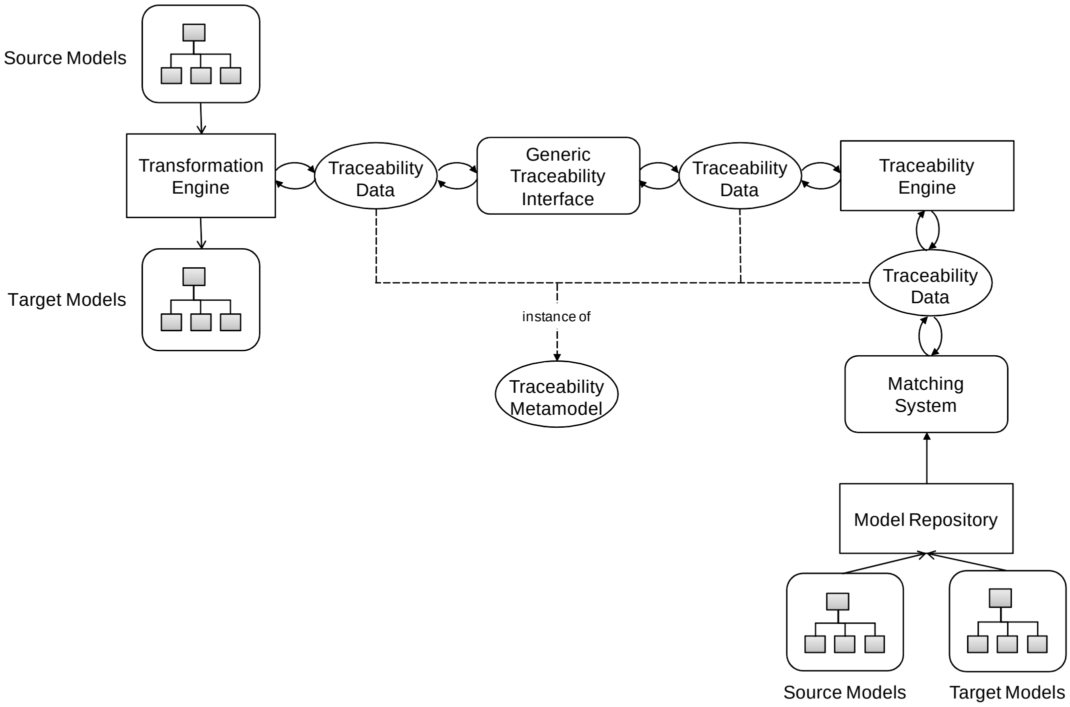


Figure 4.2.: Detailed Architecture Overview

trace links in terms of the connector-based approach in the former case and the model-matching system in the latter.

4.2. Connector-based Extraction of Traceability Data

We propose two classes of connectors for the extraction of traceability data by means of the generic traceability interface. The underlying reason for this classification stems from the dichotomy on generation classes (implicit and explicit) from Section 2.2.2, since we suggest a certain kind of connector for each generation class.

The question arises, how the choice of model-transformation approach influences the kind of connector. In case a developer uses a model-transformation approach from the implicit generation class, he may use the advantage of a traceability solution at the price of having no control over the modelling of traceability data. In this case, we propose a *blackbox* connector as depicted in Figure 4.4 on the left side. The blackbox connector serves the purpose of glue code between the traceability engine and transformation engine, without changing the latter two components, since there is no need to intrinsically change their functionality for the sake of gaining a traceability solution. According to [Aßm03] glue code maps component protocols specifically to each other.

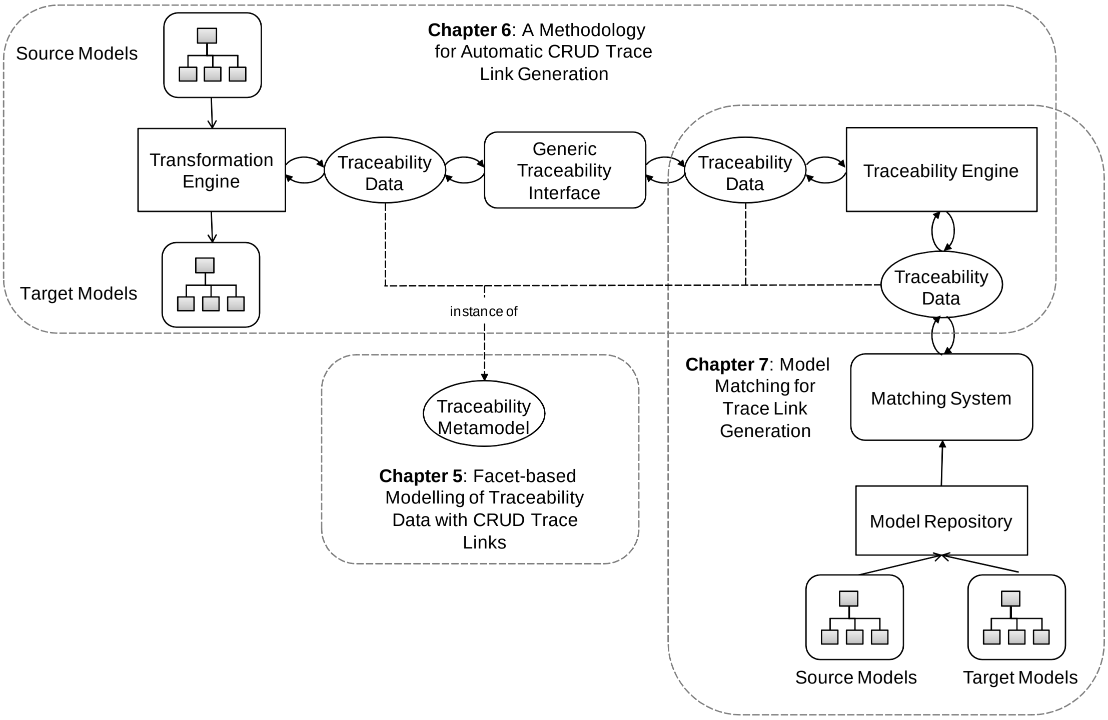

Source Models
Chapter 6: A Methodology for Automatic CRUD Trace Link Generation
Transformation Engine
Traceability Data
Generic Traceability Interface
Traceability Data
Traceability Engine
Target Models
instance of
Traceability Data
Traceability Metamodel
Chapter 5: Facet-based Modelling of Traceability Data with CRUD Trace Links
Chapter 7: Model Matching for Trace Link Generation
Matching System
Model Repository
Source Models
Target Models

Figure 4.3.: Chapter Road Map on the Generic Traceability Framework

For the explicit trace link generation class a different situation pertains, primarily due to the non-existence of a traceability solution, which comes out of the box. Thus, to overcome this disadvantage, we propose an *invasive* connector between traceability engine and transformation engine, as sketched in Figure 4.4 on the right side. This kind of connector has an invasive nature to augment the transformation engine, which is necessary for the purpose of generating traceability data.

We continue describing the details of the above-mentioned connectors in Chap 6, comparing their expressiveness, communalities and variability. In particular, we align our work with the principle of invasive software composition in [Aßm03].

4.3. Model-Matching System

The model-matching system takes two models as input and creates a mapping, that is, correspondences between model elements, as output. This mapping is further analysed with respect to extracting potential trace links. This approach to model matching is based on a parallel matching system (cf. Section 2.3.1).

In the following, we explain the processing steps of our proposed matching system, which involves the following successive steps, as depicted in Figure 4.5:

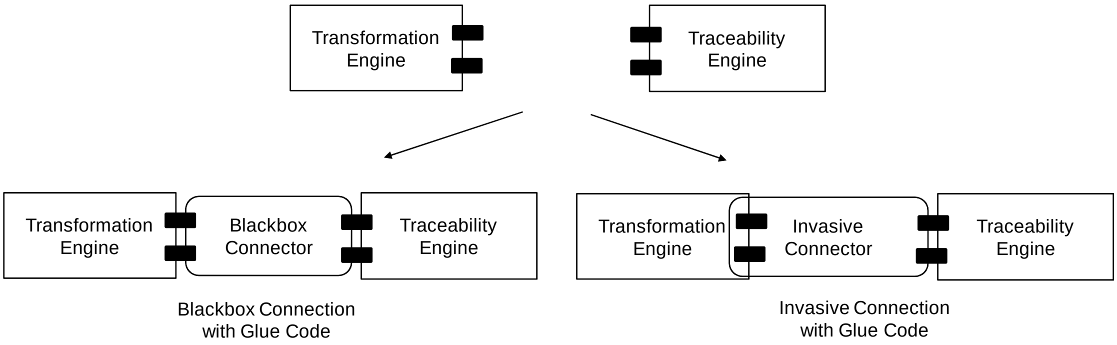


Figure 4.4.: Classes of Connectors for Traceability

1. **Import of Models**: The available source and target models need to be imported into a common data model, to have a common basis for arbitrary matching algorithms.
2. **Matching of Models**: Different matching algorithms are applied to the imported source and target models. A matching algorithm calculates a similarity value for all pairs of source and target elements. The matching system can be configured by selecting a set of matching algorithms involved in the matching process.
3. **Configuration of Similarity Value Cube**: Each matching algorithm provides separate results for a certain source and target model, where the results describe a similarity value for all source and target model element combinations, called similarity value matrix. All of these matrices (one matrix due to a certain matching algorithm) are arranged into a cube, called similarity value cube (SVC), as depicted in Figure 4.5. To derive a mapping between source and target elements out of these results, the similarity value cube needs to be configured, e.g., to form an *aggregation matrix* by calculating the average of similarity values, or *selection matrix* by selecting all elements exceeding a certain threshold. We provide a detailed description of possible configurations in Section 7.1.4.
4. **Extraction of Trace Links**: The resulting mapping is analysed and trace links are extracted according to certain heuristics, or configurations.

The matching system is further described in Chapter 7.

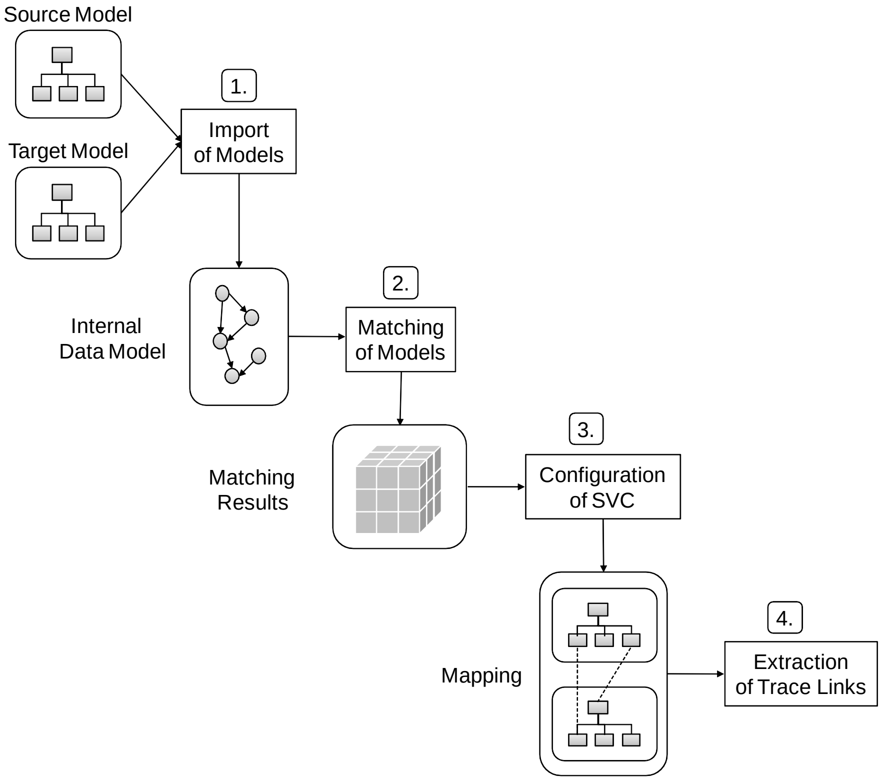


Figure 4.5.: Process Steps of the Traceability Model Matching System

5

Facet-based Modelling of Traceability Data with CRUD Trace Links

This chapter is dedicated to the modelling of traceability data in view of accounting for the *unification* and *extensibility* of traceability metamodels. The unification is targeted at in Section 5.1 through the presentation of a generic traceability metamodel with a minimal set of elementary trace links. The trace link derivation process is described in Section 5.2. Furthermore, we present a facet-based approach for the extensibility of the proposed traceability metamodel in Section 5.3. We conclude this chapter by modelling different sets of traceability data by applying graph theory. In doing so, we claim that traceability scenarios essentially are views on a certain traceability graph.

5.1. Traceability Metamodel

The traceability metamodel of the GTF depicted in Figure 5.1 includes a root element, **TraceModel**, which contains several concepts. These are successively discussed in the following sections.

An **Artefact** represents any traceable product generated in the development process, such as a requirement or class, or a compound artefact, e.g., a method inside a class. Every artefact is uniquely identified by a Universal Unique Identifier (**UUID**). A **TraceLink** is the abstraction for the transition from one artefact to another, such that an instance corresponds to a hyperedge linking numerous source and target artefacts. Working with hyperedges opens the possibility to track, whether a model element was generated in parallel to other model elements through a particular *transformer*

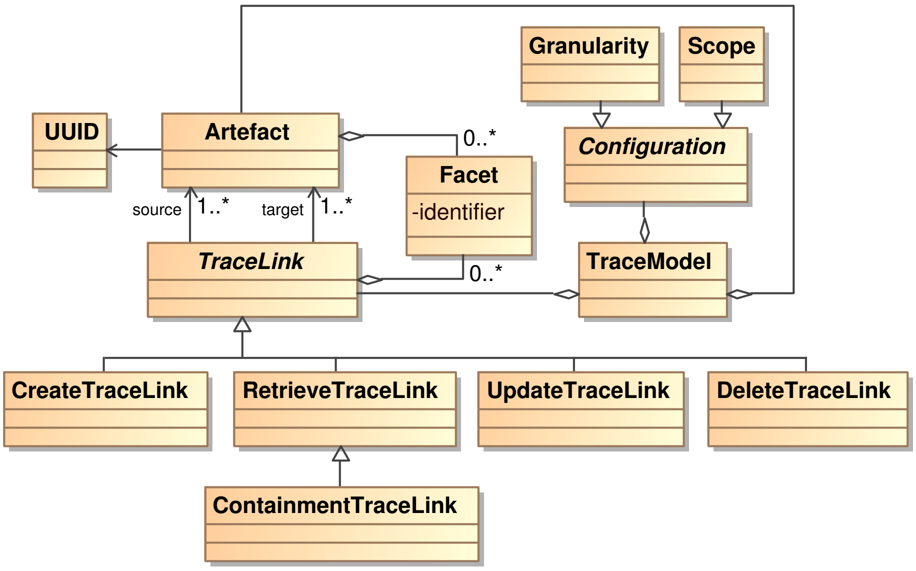


Figure 5.1.: Metamodel for Traceability

(cf. Definition 4). The above-mentioned transition is always directed, therefore a from-to relation between artefacts is defined through a trace link from source to target artefacts.

In the following sections, we will explain the remaining classes of the traceability metamodel.

5.2. Type Set of CRUD Trace Link

In the following section, we will derive a set of trace-link types for the traceability metamodel. Inherent to this derivation process are the questions, how many trace links are necessary for this set and what influences the set's semantic richness? To answer these questions, we essentially propose four elementary link types, called CRUD trace links, based on minimal MDSD-operations as we will show in the following section.

The derivation of link types for the traceability metamodel is based on the following general conditions:

- **Semantic Richness**: The link set has to account for sufficient semantic richness to enable the mapping of arbitrary model transformations to traceability data. Furthermore, since the modelling of traceability data is coupled with the kind of underlying reasoning the user performs on the collected data [RJ01], that is, to a particular traceability scenario, this has consequences for the choice of semantics bestowed on a trace link. The choice of semantics for the trace link set is therefore dependent on the set of traceability scenarios. Thus, we fix the set of traceability scenarios from Section 2.2.1 to have a frame of reference for deriving the trace-link semantics. Concluding, these traceability scenarios are: *System Comprehension*, *Coverage Analysis*, *Change Impact Analysis*, *Orphan Analysis* and *Transformation Debugging*.

- **Minimal Number**: In addition, this set needs to define a type set with a sufficient and necessary number of links with no redundancies, therefore constituting the minimal number of link types needed to account for the mentioned traceability scenarios.
- **Elementary Links**: Furthermore, the goal is to derive elementary links, where no two links from this set are linear combinations of each other. This is to avoid compositions in the type set.

The second and third conditions aim at lowering the semantic richness to the maximum possible extent. Minimising the semantic richness this way, has an advantage with respect to eliminating unnecessary link types such that the traceability user is not burdened with these potentially leading to inconsistent usage. In summary, we follow a scenario-driven derivation process of link types with the maxim to define a minimal set of elementary links.

In realising, the derivation process, we focus on the kind of relationships between source and target model elements of a transformation, e.g., a relationship of creation, or deletion. Key to the derivation process is to capture these relationships through the semantics of the proposed set of trace links. In order to get a complete view on the different kinds of relationships, we turn to the classification of source and target relationships of model-transformation approaches proposed by [CH06]. This classification holds for the following different classes as summarized in Figure 5.2: Approaches either create a new target model and/or allow for transformations to operate on an existing target model. The latter case may be split up into *update* transformations, where the existing target model is updated (other than the source model, which remains unchained), or *in-place* transformations, where the source and target model are the same model. Furthermore, update as well as in-place transformations work *destructively* or (exclusively) by *extension only*, that is, model elements may be updated or extended.

Based on the above classification, we will define a set of trace-link types for each class, describing the semantics of the possible relationships between source and target model elements. In addition, we base our definition on the well-known set of CRUD-actions [Cod70], which seem promising as frame of reference to describe the semantics of all possible changes on models [GK09], since a model element may be created, retrieved, updated or deleted.

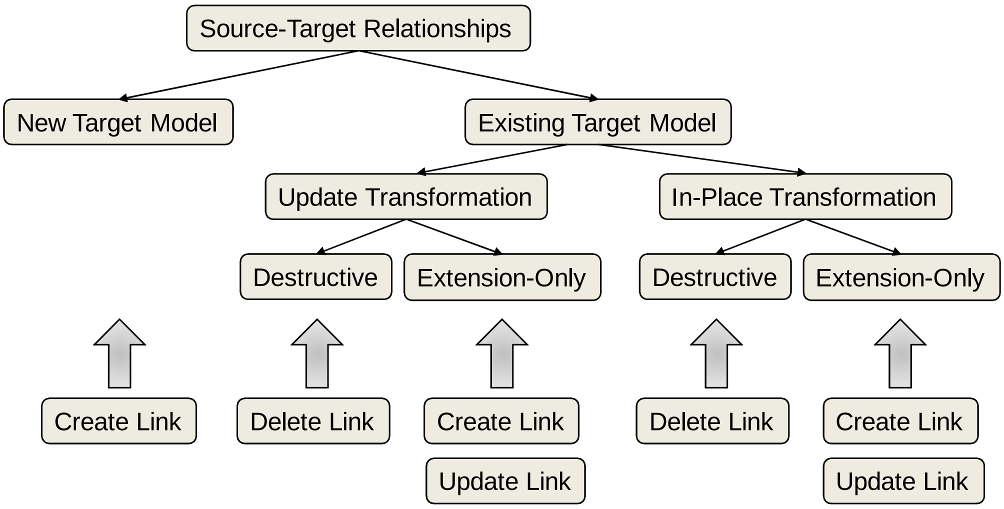


Figure 5.2.: Source-Target Relationships

In doing so, we identify three types of trace links on the basis of the above-mentioned classification, as depicted in Figure 5.2:

- For a newly created target model, we define a **create link** between respective source-model elements and all newly created target-model elements.
- For an extension-only update as well as in-place transformation, we define a **create link** between respective source-model elements and newly created target-model elements, while the extension constraint allows for the creation of new model elements. Otherwise, we define an **update link** between respective source-model elements and updated target-model elements to record that model elements are modified through an update operation.
- For a destructive update as well as in-place transformation, we define a **delete link** between respective source-model elements and deleted target-model elements to record the deletion of model elements, as supported in some endogenous transformation approaches, e.g., the refinement mode in ATL.

In addition, we identify two more links: For a query operation on a model returning a subset of model elements from the source model, we define a **retrieve link** between each source model element of this subset and returned target model element, in analogy to the retrieve operation from the set of CRUD actions. Finally, in addition to the CRUD link types, we propose a **containment link** due to containment relationships of model elements.

Therefore, the traceability metamodel in Figure 5.1 is defined through five types of trace links in total: **CreateTraceLink**, **RetrieveTraceLink**, **UpdateTraceLink**, **DeleteTraceLink** and **ContainmentTraceLink**. The containment relationship is defined as a special case of a model query und thus subclasses the RetrieveTraceLink.

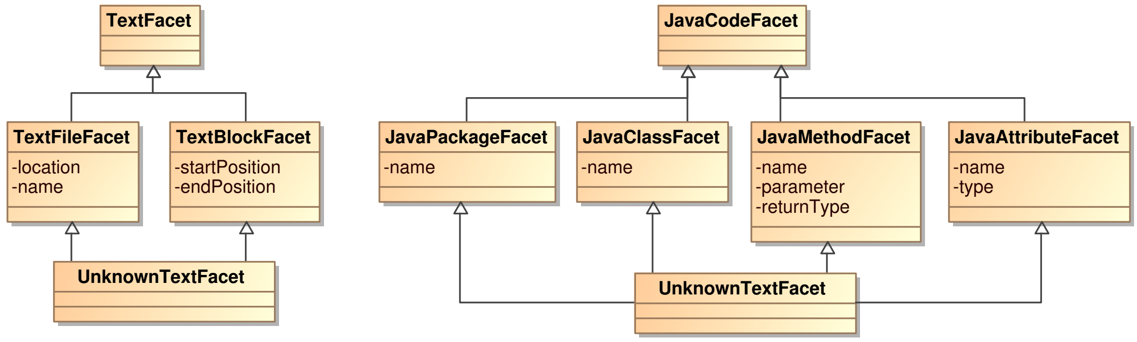


Figure 5.3.: Facets for the Traceability of Source Code

5.3. Facets for Modelling of Traceability Data

Furthermore, to assign types to artefacts and trace links, we use the concept of facets as defined in Section 2.5 through Definition 35. This means that the traceability metamodel assigns a set of facets (**Facet**) to every artefact and trace link.

Possible facets for modelling traceability data are given below. These might capture: the *life cycle* of artefacts, taking on the faceted values, such as, requirement, design, code, or test artefact; the *location*, where artefacts were created; the *stakeholder* of artefacts with values like project manager, or developer. We provide another example facet in Figure 5.3, namely a source-code facet. This facet has two subfacets called **TextFacet** and **JavaCodeFacet**, as explained in the following. The TextFacet identifies code artefacts, either being a text file or a text block within such a file. For these types, we introduce two faceted values **TextFileFacet** and **TextBlockFacet**, respectively. The attributes of the TextFileFacet can be set to identify the *location* and *name* of a file. In terms of the TextBlockFacet, the *startPosition* and *endPosition* of a block inside a file can be set. For the purpose of identifying source code the usage of start and end position may not be sufficient. Therefore, a more advanced approach is provided by the **JavaCodeFacet** for tracing programming code written in Java. Hereby, we distinguish between faceted values referring to packages, classes, methods and attributes. Hence, it is possible to identify Java-specific artefacts, e.g., a Java method with a specific name, as opposed to general artefacts, such as strings, bounded by the *startPosition* and *endPosition* of the **TextBlockFacet**.

The motivation for choosing facets is twofold. Since facets factorize inheritance hierarchies and thus, simplify them, we use this advantage for the sake of simplifying artefact- and link type hierarchies. Without the use of facets, a full model would multiply all classes, leading to the product of $n_1 n_2 ... n_f$ classes, where n_i denotes the number of instances holding for a facet, $i \in N$ and N is the set of natural numbers with f denoting the number of facets. On the contrary, modelling with facets amounts to using $n_1 + n_2 + ... + n_f$ classes.

Secondly, facets account for an extensibility mechanism for the type system of the traceability metamodel. Since facets can be varied independently, the extensibility of the traceability metamodel is achieved, aligned with Requirement 4 of the requirements analysis (p. 49). More precisely, the definition of case-specific traceability metamodels (with regard to the traceability scenarios) through facet-based extensibility implies the following adaptations: a) Selecting the required facets for a given traceability scenario and b) **Configuration** of **Granularity** and **Scope**.

The Configuration of the granularity level of traceability data implies the specification of those artefacts and trace links that need to be traced with regard to a particular traceability scenario. Thus, in the context of facets, this implies, choosing facets and faceted values from the set of facets defined in a). For instance, regarding the previously mentioned code facets, the traceability of artefacts referring to text blocks might be too fine-grained, whereas the granularity of artefacts referring to files might be sufficient. In this case, artefacts with a **TextBlockFacet** would not be traced.

The Configuration of the traceability scope implies to constrain the traceability data in the sense of having a specific attribute-value combination. While the configuration of granularity solely checks for the existence of facets, the configuration of scope additionally examines a facet-specific property. For instance, regarding the **TextFileFacet** it might be necessary to trace only **TextFiles** of a certain *name*. In this case, the configuration of scope requires the TextFileFacet's attribute, *name*, to be set accordingly.

5.4. Expressiveness of Traceability Data

Traceability scenarios essentially are views on the traceability graph. The chain of reasoning for this statement is as follows. Since the traceability scenario dictates the traceability model and consequentially the traceability-data expressiveness, it is of vital importance to have an adequate expressiveness in order to conduct traceability scenarios successfully. Principally, it is possible to use a separate traceability graph for each scenario, yet for reasons of maintenance, we choose to work with one traceability graph in total. This graph in turn has to account for the expressiveness mandated by all traceability scenarios chosen by the user. Therefore, the scenarios are views on the traceability graph.

In the following, we define classes of traceability graphs based on the interpretation of *transformers* (cf. Definition 4) with respect to the traceability metamodel from Section 5.1. Recalling from Definition 7 in Section 2.2, traceability is the tracking of corresponding source and target artefacts of transformers in the transformation program. According to the traceability metamodel, the basic unit of traceability data relates a certain source and target artefact through a trace link. This raises the question how, to interpret the transformers regarding the above-mentioned triple. There are two different ways of modelling a traceability graph, depending on whether the transformer is perceived as a *trace link* with certain attributes, e.g., name and identifier, or as an *artefact* as such.

In Figure 5.4 an example graph of the former way of modelling is depicted as an instance of the traceability metamodel in Figure 5.1. Regarding the semantics of the graphical notation, we refer to Section 2.4 on typed attributed graphs. A source and target artefact both with a data node *artefact name* and graph node *identifier* are connected through a trace link of the CRUD type set. This trace link carries a graph node, *identifier* and a data node, *operator name*.

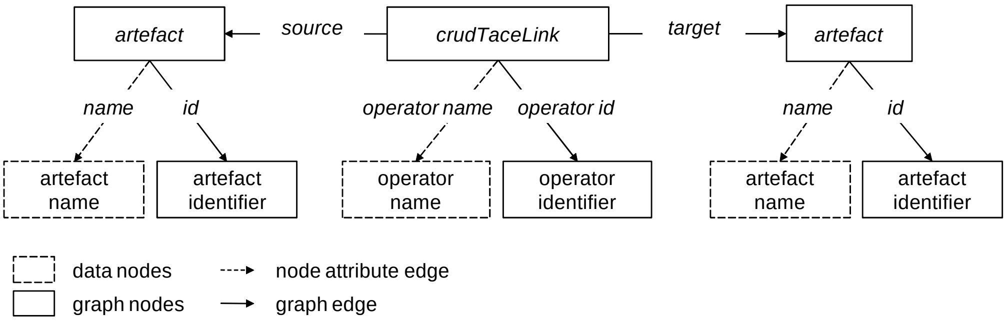


Figure 5.4.: Traceability Graph with Transformer represented as Trace Link

In contrast, Figure 5.5 shows an example of the other type of traceability-graph modelling on the basis of defining a transformer as an artefact. In this case, two different trace links are needed, one, to link the source artefact to the transformer and another one between the transformer and the target artefact. Therefore, two corresponding trace links are represented as graph nodes with exemplary labels, that is, *transformer input* and *transformer output*. The naming of the labels is chosen for illustrative purposes only.

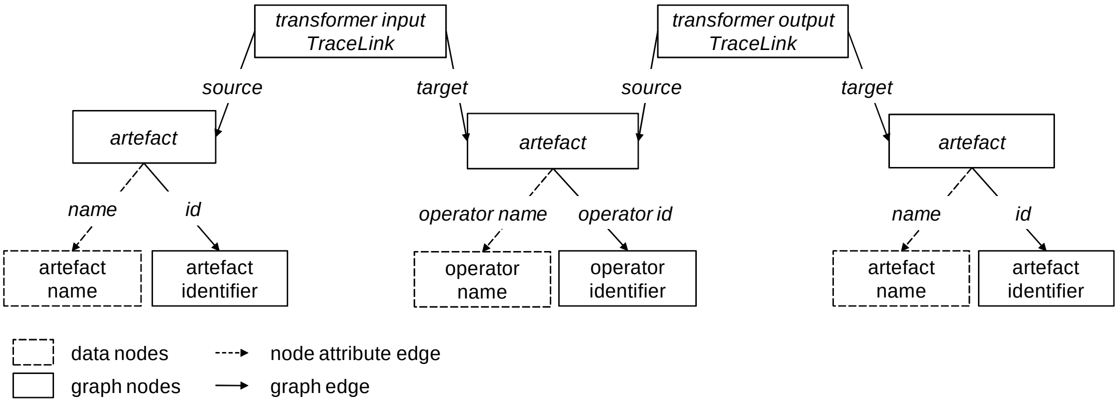


Figure 5.5.: Traceability Graph with Transformer represented as Artefact

The above classification gives insight into two alternative ways of modelling traceability data. As we will show in Chapter 6, this has consequences for the design of connectors. Furthermore, we describe, how the set of CRUD trace links are used with respect to different transformers, that is, we assign the set of CRUD trace links to transformers.

5.5. Contributions

This chapter contributes the following:

C1: **Facet-based Modelling of Traceability Data with CRUD Trace Links**

In this chapter, we contribute a generic traceability metamodel with a minimal set of 4 elementary trace links founded on the set of CRUD actions from database operations. The derivation process of these link types is based on the classification of source and target relationships from [CH06], to capture the semantics of all possible relationships between source and target elements of model transformations. In doing so, corresponding link types are derived for each class by using the set of CRUD actions.

Furthermore, the traceability metamodel includes an extensibility mechanism based on facets to account for a sufficient expressiveness of traceability data. The specification of user-defined artefacts and link types in traceability models is possible through the definition of corresponding types as facets. Since facets factorize inheritance hierarchies and thus, simplify them, we use this advantage for the sake of simplifying artefact and link type hierarchies. Furthermore, since facets can be varied independently and recombined, the extensibility of traceability metamodels is achieved. The set of CRUD trace links builds the foundation on defining link types in a standardized way, yet may be extended through the use of facets as described above.

6

A Methodology for Automatic CRUD Trace-Link Generation

In the following chapter, we propose a methodology for the generation of trace links in terms of a design pattern, called *Generic Traceability Interface* (GTI), which essentially is a contract the developer has to comply with in order to gain a traceability solution for his model transformation approach. Implementing the GTI necessitates three development steps, as we will show in Section 6.1. Depending on the class of trace-link generation (explicit or implicit), the methodology proposes two different kinds of connectors as part of the GTI, namely *blackbox* and *invasive* connectors in Section 6.2. Both connectors serve the purpose of augmenting model transformations for the sake of generating traceability data.

6.1. Generic Traceability Interface (GTI)

The *Generic Traceability Interface* (GTI) needs to allow the cooperation of arbitrary transformation and traceability engines in order to collect traceability data. Consequently, in defining the GTI, we need to abstract from both the transformation engine performing model operations and the traceability engine collecting traceability data. Figure 6.1 depicts the GTI. Interfaces regarding the transformation engine are summarized in the subpackage **TransformationEngine**, analogously, traceability-engine concerns are grouped in the **TraceabilityEngine** subpackage. Common structures and behaviour are contained in the **Core** package. In the next sections, we will discuss every interface in detail and derive the steps a developer has to take, to implement the GTI. In summary, three steps are necessary by implementing the following classes:

1. **UUID** for the artefact identification mechanism and **Facets** for defining custom facets
2. **Repository** for storing traceability data and **Configuration** for the specification of **Scope** and **Granularity** (as defined in Chapter 5, Section 5.3)
3. **Connector** for extracting traceability data from the transformation engine and **FacetFactory** for creating facets

Step 1: UUID and Facets

The **Core** package provides essential interfaces used by both the transformation and traceability engine. Essentially, the Core package realises the left part of the traceability metamodel (Artefact, TraceLink, UUID, Facet) as depicted in Figure 5.1. **TraceLinks** are characterised by a type, referring to the 5 TraceLink subclasses in the traceability metamodel. The relationship is modelled as an association to an enumeration. We disregard an alternative subclassing realisation, due to the fact that the types of trace links are fixed to a number of five, as specified in the traceability metamodel. Furthermore, TraceLinks aggregate a set of source and target artefacts, while establishing a directed relationship among them. An **Artefact** is required to encapsulate an unambiguous reference, denoted as Universal Unique Identifier (**UUID**). A UUID is split up into a Unique Resource Identifier (URI) and a fragment identifier. The URI is used to uniquely identify resources. This is an open-ended concept, since an URI may refer to a file in the file system, a web page on the internet or any other kind of data. A fragment identifier, on the other hand, uniquely identifies objects within a resource. The usage of UUIDs solely depends on the implementation of the GTI, yet it is important to define a mapping between the identification format used by the transformation and traceability engine and the UUID interface implementation.

Since facets (defining the artefact type) encapsulate any conceivable view on artefacts to allow for a flexible artefact type definition, the interface **IFacet** is kept as generic as possible, yet we require a facet to have a name and value.

Thus, the first development step entails to define the types of facets to be used, as implementations of IFacet. Secondly, an implementation of UUID is needed.

Step 2: Repository and Configuration

In the **TraceabilityEngine** subpackage, all interfaces relevant to the traceability engine are summarized. The central interface is the **TraceEngineManager**, providing access to registered repositories and configurations. Repositories implement the **IRepository** interface, which constitutes the connection point for arbitrary traceability engines to the generic interface. IRepositories own a name for providing an identifier. Additionally, they own the methods *connect()* and *disconnect()*, which hold for possible database

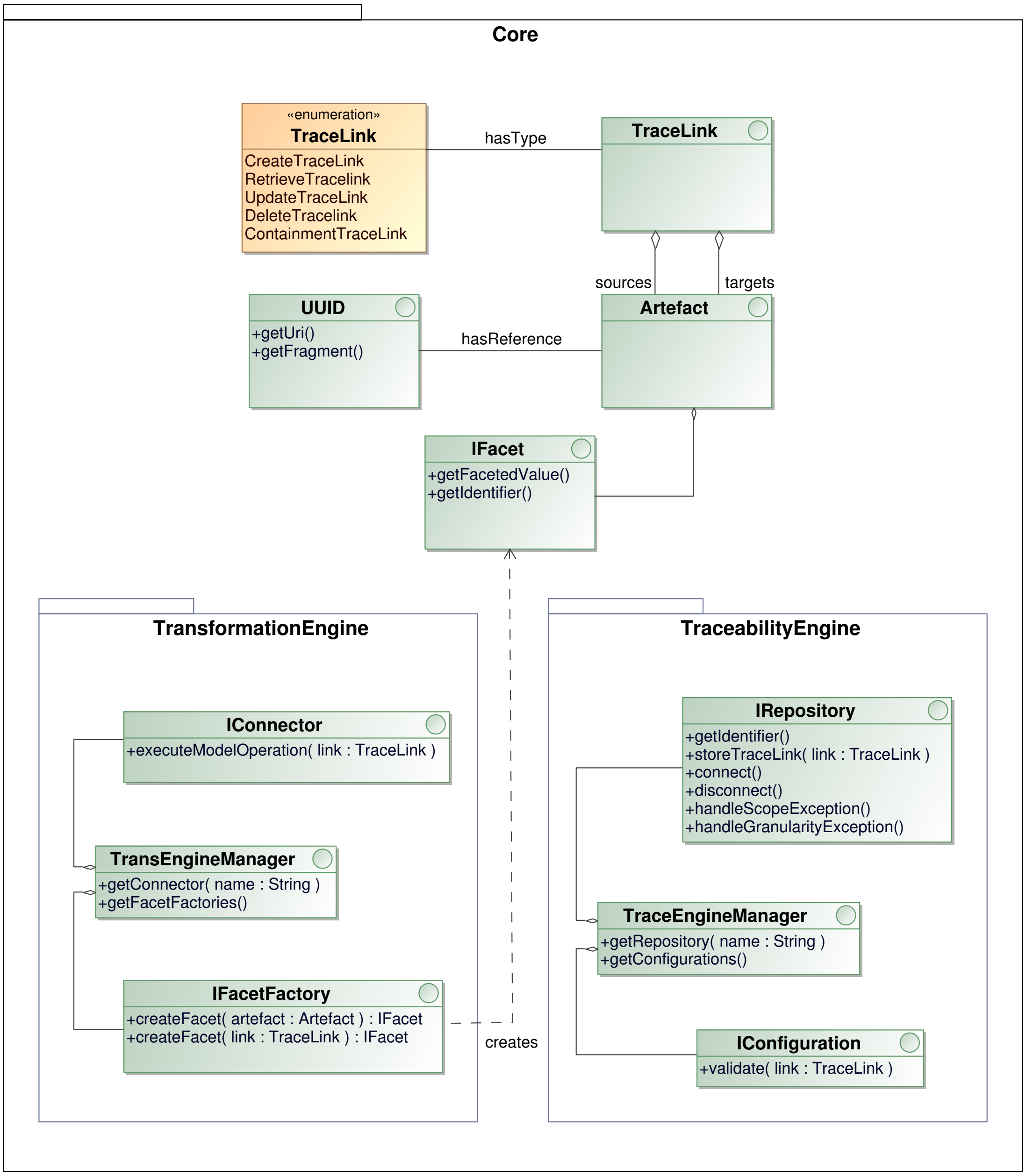


Figure 6.1.: Generic Traceability Interface

connections for persistency. The most fundamental method is *storeTraceLink()*, which persists a trace link, its source and target artefacts and all contained facets.

Furthermore, the exception handler methods are used for the configuration of traceability data, which essentially is realised by **IConfiguration**. IConfiguration has a validation method for validating traceability data. The method, thus takes a trace link as input and checks, whether the trace link and referred source and target artefacts are within the correct scope and granularity level (cf. Section 5.3).

Concluding, a developer has to implement IRepository and IConfiguration, as described above.

Step 3: Connector and Facet Factory

The **TransformationEngine** subpackage contains all interfaces relevant to the transformation engine and provides the connection point for arbitrary transformation engines. The central interface is the **TransEngineManager**, which provides access to registered connectors and facet factories. Connectors implement the **IConnector** interface, which constitutes the connection point for arbitrary transformation engines to the generic interface. The connector's task is to collect traceability data in parallel to the execution of a model transformation. We propose two phases for completing this task: The **Extraction Phase** and **Typing Phase** of traceability data. The extraction phase is to derive traceability data from the integral model mapping of a model transformation and is defined as follows:

Definition 36 (Extraction Phase)**.** *The extraction phase determines the traceability graph structure through the instantiation of* **Artefact** *and* **TraceLink** *of the traceability metamodel.*

Therefore, the extraction phase concerns the left region without the use of **Facet** of the traceability metamodel.

The typing phase uses the result of the extraction phase and is defined as follows:

Definition 37 (Typing Phase)**.** *The typing phase enhances the traceability graph resulting from the extraction phase by determining the types of its artefacts and trace links through the instantiated **Facets** and settings from the configuration of **Granularity** and **Scope** of the traceability metamodel.*

Essentially, the typing phase revolves around the *executeModelOperation()* method, which is the core method triggering all necessary steps from receiving trace links from the extraction phase to their persistence.

In Section 6.2, we will elaborate on the detailed working of connectors, the extraction as well as typing phase and present a novel way of combining and automating the traceability-data extraction with aspect-oriented programming.

Finally, the TransEngineManager includes **IFacetFactories**. Facet factories are facilities creating facet objects out of traceability data and/or transformation-engine specific knowledge by adding faceted values to artefacts and trace links. For instance, the creation of the so-called **TransformationEngineFacet** for tracing a specific version of transformation engine is completely independent of the traceability data. This is due to the fact that the current version of a transformation engine is available within its framework. On the contrary, the **JavaCodeFacet** factory (cf. Figure 5.3), solely applying to Java files, is dependent on the traceability data. Additionally, there are facet factories, which are decoupled from both factors, transformation engine and traceability data, e.g., a **TransformationTimeFacet** for tracing the time of creation for a certain artefact with respect to a Gregorian time stamp (date and time), or fiscal time stamp (fiscal year and week) depending on a given Java Virtual Machine providing the correct time stamp.

All in all, regarding the third step, it is necessary to implement a connector and the necessary facet factories.

Collaboration Behaviour of Interfaces

In the following, we show the collaboration of the GTI interfaces in order to collect traceability data. Thereby, we describe the tracing process, starting with the creation of trace links out of a model transformation and ending with their persistence in a traceability repository. The individual process steps are outlined in Figure 6.2 as sequence diagram.

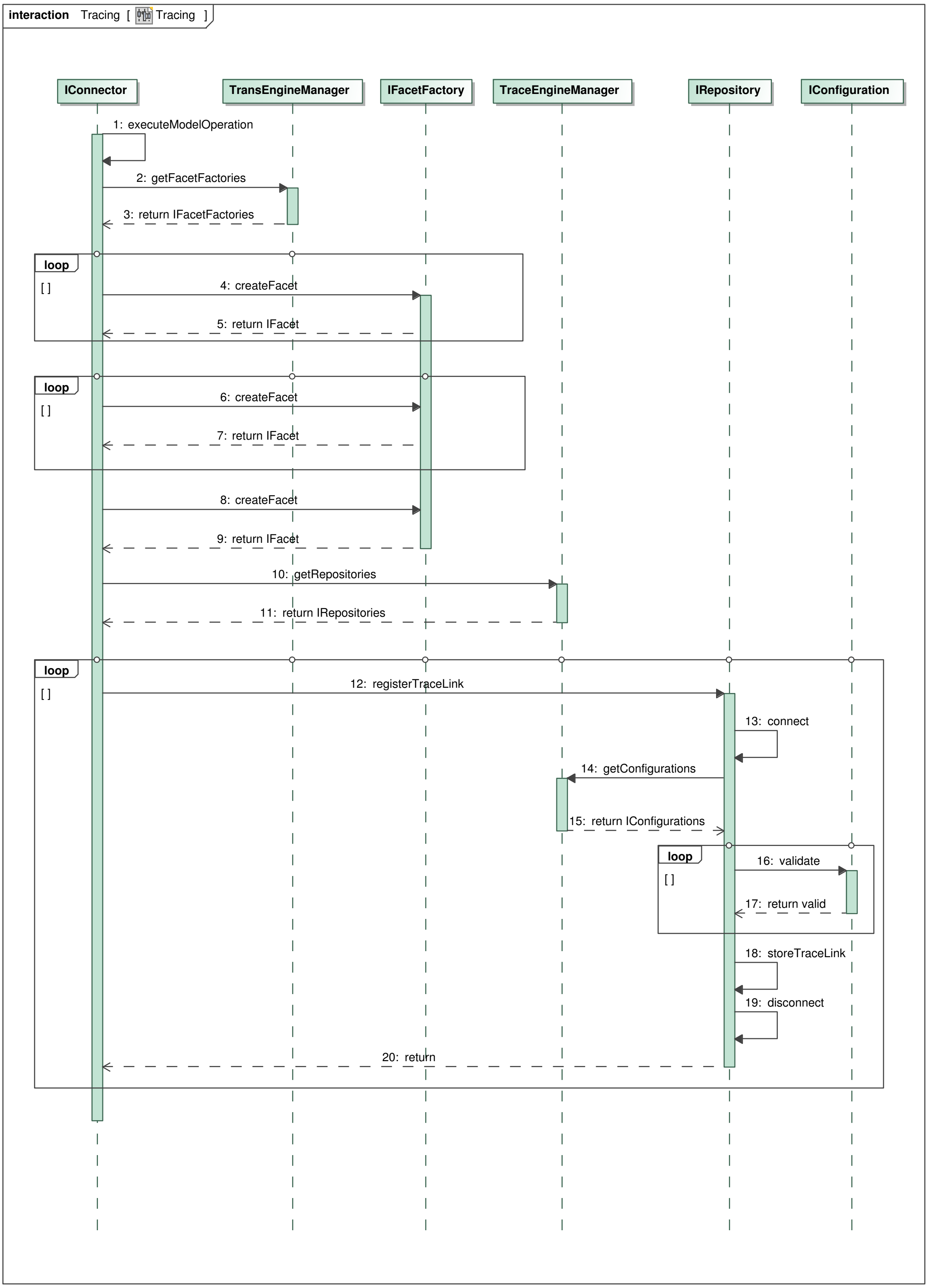


Figure 6.2.: The Tracing Process regarding the GTI

First of all, the *executeModelOperation()* method is invoked by the IConnector, which triggers the tracing process, whenever a model transformation is executed. At this time, we abstract from the issue how to extract traceability data and provide the details in Section 6.2. Thus, we start explaining the tracing process after the extraction phase, that is, trace links referencing source and target artefacts have been created.

The following process steps relate to the typing phase. After creating a trace link and its corresponding source and target artefacts, facets have to be added. Therefore, the **TransEngineManager** is required to provide all registered **IFacetFactories**. Facets are then added to the traceability data by iterating over all source artefacts (Process Step 4–5), target artefacts (Process Step 6–7) and finally, the currently processed trace link itself (Process Step 8–9). In each iteration, the *createFacet()* method of the IFacetFactory is invoked. In case the factory can be applied to the traceability data, the currently processed facet is returned and stored in the respective artefact or trace link.

Afterwards, the registration of trace links to corresponding repositories is carried out. In this way, the **TraceEngineManager** is called to invoke the *getRepositories()* method. In turn, a connection to the repository is established through IRepository. Thereafter, traceability data is validated (Process Step 16–17), and hence, all registered configurations are requested from the TraceEngineManager. A trace link is validated by every **IConfiguration** to check, whether the trace link and respective source and target artefacts are within the correct scope and granularity level (cf. Section 5.3).

Finally, if the validation does not fail, the trace link is stored in the traceability repository and the connection is closed. In summary, the typing phase concerns the storage of facets to artefacts and trace links (Process Steps 4–9) and the validation of granularity and scope (Process Step 16–17).

6.2. Augmentation of Model Transformations with Generation of Trace Links

In the following two sections, we come back to the proposal of two classes of connectors, namely invasive connectors and blackbox connectors. For the explicit generation class, we propose the former connector with an invasive nature to augment the transformation engine. This allows for custom modelling of traceability data. On the other hand, the blackbox connectors are proposed for the implicit generation class. A blackbox connector serves the purpose of glue code between the traceability engine and transformation engine, without changing the latter two components.

We continue with a detailed description on the design of invasive connectors in Section 6.3 and blackbox connectors in Section 6.4.

6.3. Invasive Connectors

As mentioned before, the implementation of a connector entails using the *executeModelOperation()* method, taking a trace link, referencing source and target artefacts as input and proceeding with the tracing process outlined in Figure 6.2. Essentially, these processes concern the *typing* of traceability data. What has not been dealt with is, how traceability data is extracted, that is, what happens before the *executeModelOperation()* method is invoked. Yet, the issue of *extraction* is at the core of an invasive connector.

In the following section, we shed light on the invasive connector's functionality concerning the two phases: a) **extraction** and b) **typing** (cf. Definition 36 and 37) of traceability data. Recalling from Chapter 5, the choice of traceability scenario determines the kind of traceability model used, which in turn defines the traceability data, as pointed out in Figure 6.3. We will show that this contract applies to the *extraction* as well as *typing* phase.

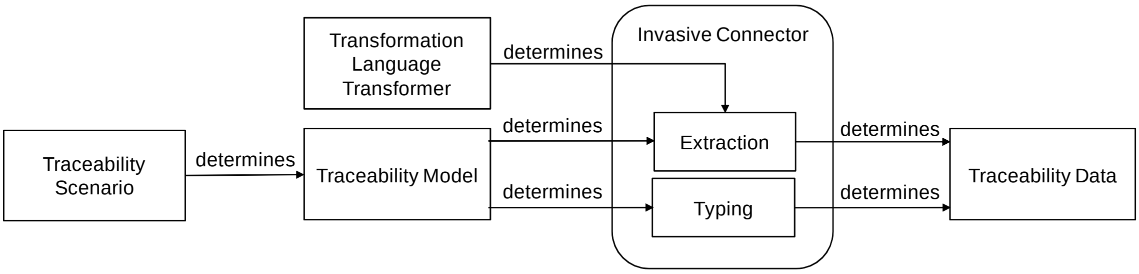


Figure 6.3.: Scenario- and Transformer-driven Invasive Connector

Furthermore, we investigate the influence of *transformers* (cf. Definition 4), that is, transformation rules and operators, on the extraction phase. In order to describe the influence of the transformation rules on the extraction phase, we take on a more finegained view on transformation rules, that is, their contained operators (cf. Definition 3). In doing so, we will show that the extraction depends on the transformation rules and certain kind of operators included in the rule (i.e., the process is transformer-driven) as well as on the traceability scenarios (i.e., the process is scenario-driven). The typing phase in turn is only scenario-driven.

6.3.1. Running Example

In order to explain the extraction phase, we provide an example transformation. As representative from the explicit generation class, we chose the model-to-text transformation language, Xpand [Foug] with an example transforming entities to object-oriented class specifications. The source model, as depicted in Figure 6.4, contains an **Entity**, called *Person* with respective **Features** *name*, *age* and *address*.

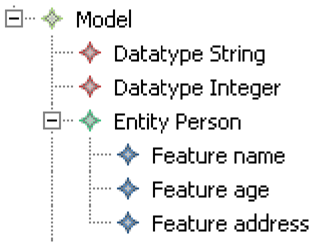


Figure 6.4.: Source Model representing the Person Entity

This source model is transformed through the transformation program (Xpand template) from Listing 6.1. The resulting target source-code is represented in Listing 6.2 showing a **class**, called *Person*, containing three **Fields**, namely *name*, *age* and *address* as well as according getter and setter methods.

```
<<IMPORT metamodel>>

<<EXTENSION template::GeneratorExtensions>>

<<DEFINE main FOR Model>>
<<EXPAND javaClass FOREACH entities()>>
<<ENDDEFINE>>

<<DEFINE javaClass FOR Entity>>
  <<FILE name+".java">>
    public class <<name>> {
      <<FOREACH features AS f>>
        private <<f.type.name>> <<f.name>>;

        public void <<f.setter()>>(<<f.type.name>> <<f.name>>) {
          this.<<f.name>> = <<f.name>>;
        }

        public <<f.type.name>> <<f.getter()>>() {
          return <<f.name>>;
        }
      <<ENDFOREACH>>
    }
  <<ENDFILE>>
<<ENDDEFINE>>
```

Listing 6.1: Xpand Template

```
public class Person {

  private String name;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  private Integer age;

  public void setAge(Integer age) {
    this.age = age;
  }

  public Integer getAge() {
    return age;
  }

  private Address address;

  public void setAddress(Address address) {
    this.address = address;
  }

  public Address getAddress() {
    return address;
  }

}
```

Listing 6.2: Target Source-Code representing the Person Class

6.3.2. Extraction Phase

We start off with the extraction phase and provide a guideline on designing this phase for the connector. Essentially, there are two steps to follow:

1. **Structuring of Traceability Graphs**: To determine the structure of the traceability graph, we look at,
    a) the constraints the traceability metamodel imposes on the extraction,
    b) the influence of traceability scenarios on the extraction, i.e., scenario-driven extraction
    c) the influence of transformation rules and operators on the extraction, i.e., transformer-driven extraction
    d) the necessity of containment relationships
2. **Automation of Extraction**: To automate the augmentation of model transformations, we apply aspect-oriented programming.

In the following, we go into the details of the above-mentioned road map by proceeding with the same outline:

1. Structuring of Traceability Graphs:

1. a) Traceability Metamodel Constraints

Traceability in MDSD is the runtime footprint of transformation execution (cf. Definition 6). Based on this definition, we interpret the runtime footprint of a transformation as the tracking of corresponding source and target artefact(s) of *transformers*, that is, rules and operators, in the transformation program (cf. Definition 7). Thus, the extraction needs to capture this runtime footprint and extract the implicitly given logical dependencies of source and target artefacts per rule and operator relevant to traceability. On an abstract level, the extraction essentially implements a mapping of the model transformation to traceability data.

The constraints that the traceability metamodel imposes on the extraction essentially concerns the region left of the dotted line in Figure 6.5. According to the traceability metamodel, one or more source and target artefacts are related through a trace link. Thus, this principle holds for the modelling of traceability data during extraction and is the format we will adhere to. However, as differentiated and motivated in Section 5.4, there are two different ways of modelling traceability graphs, depending on whether the above-mentioned rule or operator is interpreted as a *trace link*, or as an *artefact* (cf. Figure 5.4 and 5.5). For the rest of this section, we will explain our approach based

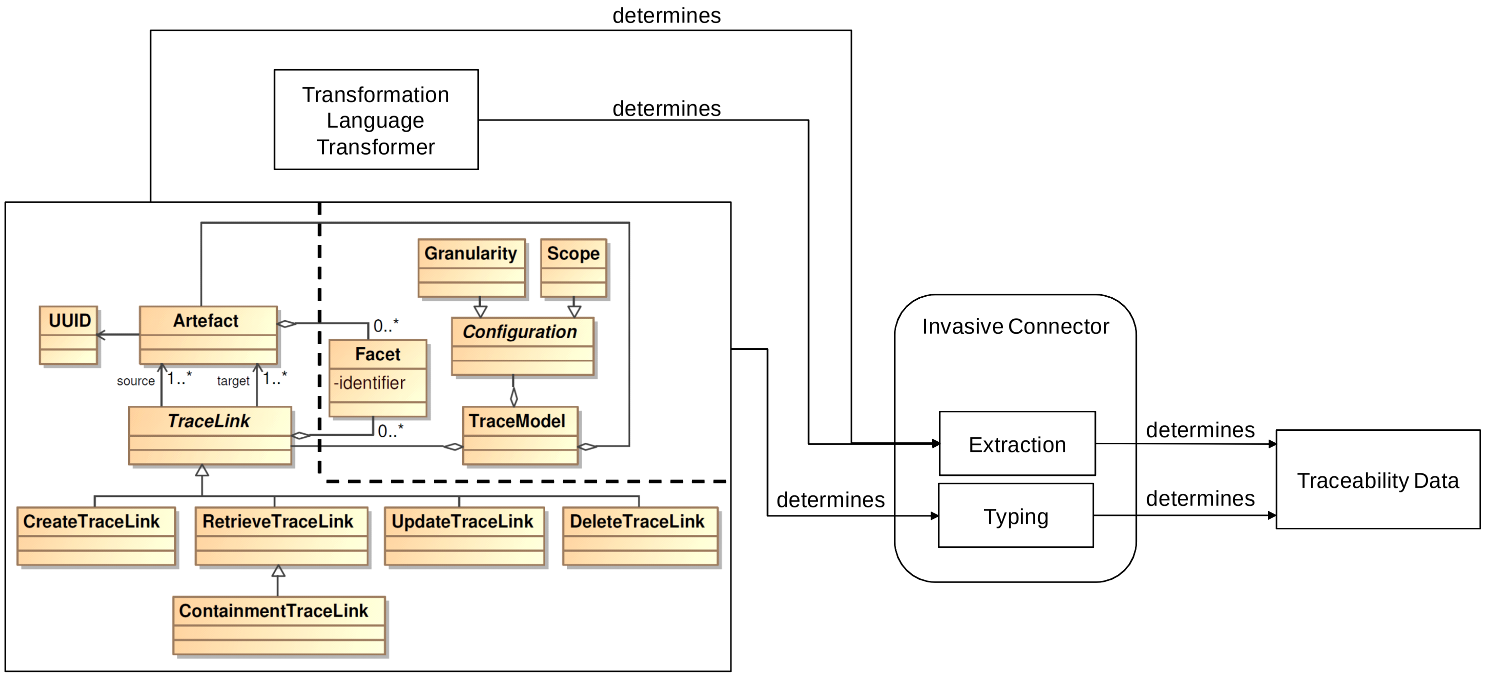


Figure 6.5.: Constraints of the Traceability Metamodel on the Connector

on the former option. However, all proposals on modelling can be transferred to the latter option in applying the same CRUD link types to the pair of trace links modelled in the graph of Figure 5.5.

1. b) Scenario-driven Extraction

The kind of traceability data to be collected depends on the selection of traceability scenarios. For example, for a debugging use case it is essential to trace certain attributes (e.g., an identifier) of a given rule and/or operator to reflect the debugging of the transformation program, while it might not be the case for a simple system analysis for inspecting the dependencies of model elements. Provided the application of the invasive connector-based approach, this kind of data can only be traced during the runtime of transformation and thus, needs to the traced during the extraction phase.

Since we have fixed the traceability scenarios to *System Comprehension*, *Coverage Analysis*, *Change Impact Analysis*, *Orphan Analysis* and *Transformation Debugging*, we instantiate a single traceability graph with an expressiveness valid for the above-mentioned scenarios without loss of generality of our approach, since the individual scenarios are views on the traceability graph as argued in Section 5.4. In doing so, we propose to trace the identifier and name of rules and operators next to their associated trace links as described under 1.a). Furthermore, the trace links are typed over the set of CRUD link types. How these are interrelated, follows from the transformer-driven extraction presented next.

1. c) Transformer-driven Extraction

We begin this section by looking closer at the definition of rules and their operators regarding the difference of model-to-model transformations and model-to-text transformations. The former uses *rule-based* transformations as specified in the QVT standard, while the latter is based on *hard-coded* transformations typically given in template languages [Kle08]. In Table 6.1, examples for corresponding rules and operators are given for the above classification. We will show that this classification has an impact on the way traceability data is modelled. In doing so, we present a *course-grained* and *fine-grained* level on modelling, respectively for rules and operators (cf. Table 6.1, 4th column). Furthermore, we provide a classification on operators relevant to traceability-data modelling.

Transformer	**Model-to-Model**	**Model-to-Text**	**Traceability**
Operator	OCL query	our operator classification	fine-grained
Rule	QVT rule [Obj11]	MOFScript rule [Fouf]	coarse-grained

Table 6.1.: Rules and Operators regarding the Granularity of Traceability

In turns out that this classification is applicable to model-to-model and model-to-text transformations.

I Model-to-Text transformations: In this section, first the operator classification necessary for model-to-text transformations is derived. Thereafter, we follow up on the traceability of rules.

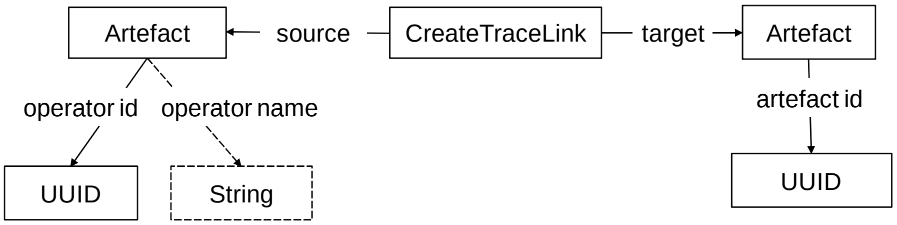
We identified certain reoccurring patterns applying to the extraction of traceability data. These depend on the type of operator used. In order to explain this principle of operator-driven extraction, we analyse the different operators in template processing and derive for each such operator, how the extraction has to take place, that is, a) which specific CRUD-trace link type(s) to use and b) which enhancements to the traceability data are necessary to account for an adequate expressiveness to conduct the traceability scenarios successfully.

For the pattern derivation, we investigated different template-based approaches and identified 6 different kinds of operators as summarized in Table 6.2. This classification is aligned with the work of [Har11] on template processing of XML documents, including a classification of operators on slot markup languages. Recalling from Chapter 5, the classification on source-target relationships and the derivation process of the set of CRUD trace links (cf. Figure 5.2), model-to-text transformations are exogenous transformations, thus creating a new target. Thus, all operators will be implemented with a **CreateTraceLink**, except for one case using a **RetrieveTraceLink**. In the following, we describe each identified patterns and give an example based on the running example of Section 6.3.1. To identify artefacts in templates as well as source code, we will use the line numbering as references in corresponding Listings. Furthermore, the proposed approach on modelling of traceability data is specified in terms of TAGs (cf. Section 2.4). To differentiate between the levels of model abstraction, the graphical notation on the level of the traceability metamodel uses capital words, while its instantiation uses cursive letters and non-capital words:

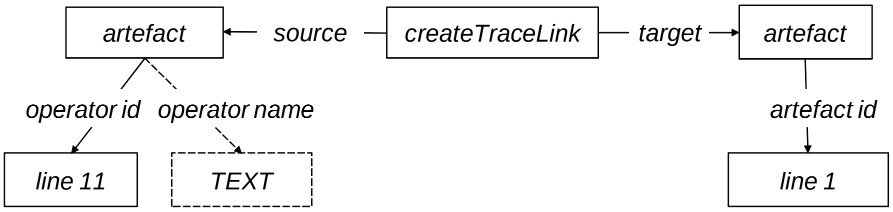
Operator	**Source Artefact**	**Target Artefact**	**Link Type**
i. Static Text	template snippet	text block	CREATE
ii. Model Access Operators	model element	text block	CREATE
iii. Data Manipulation Operators	model element(s)	text block	CREATE
iv. Control Flow Operators	model element	text block	CREATE
v. Modularization Operators	model element	text block/file	CREATE
vi. Query Operators	model element	template snippet	RETRIEVE

Table 6.2.: Derivation of Traceability Data out of Operators

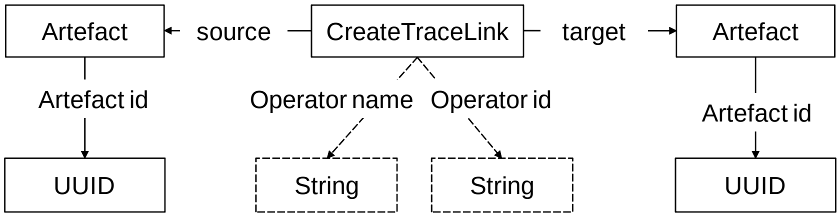
i. **Static Text Operators** generate static text without modification to the target. Since the static text operator receives its input from the template, we choose this input of static text as source artefact, which is linked to the static text in the target by using the trace link type **Create**. Additionally, we propose to track the operator *name* and *identifier*, leading to the following traceability ATG:



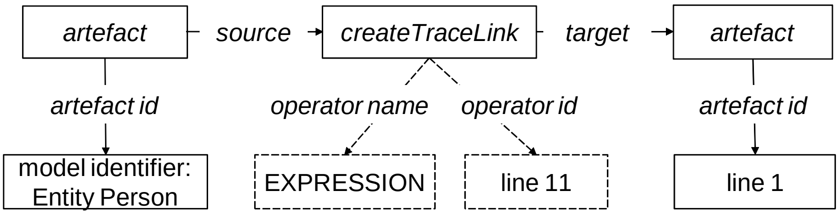
In Xpand, TEXT-statements result in textual information from the template being written to the output file, where a TEXT statement is any text, which is not surrounded by guillemets, e.g., *public class* in Listing 6.1, Line 11. According to the above concept, the following traceability graph results, as instance of the above attributed typed graph. The *source artefact* refers to the textblock *public class* in Line 11 of the template and the *target artefact* to its corresponding textblock in the generated source code in Line 1:



ii. **Model Access Operators** retrieve data from the model to either directly generate it to the target, or to do some additional processing with it. To model the traceability data structure, we propose to link the source-model element from which the data was retrieved to the generated target artefact by using a **CreateTraceLink**:



The EXPRESSION-statement of Xpand evaluates properties of the source model elements according to its metamodel, for example, the operator «name» in Listing 6.1, Line 11 retrieves its value *Person* from the name of the model element *Entity Person*. In order to capture this traceability relationship, we define a CreateLink between the source-model element and the text block in the target:



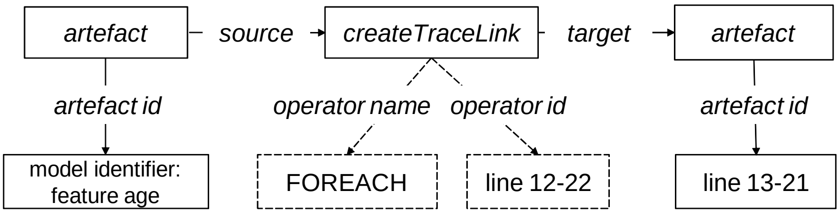
iii. **Data Manipulation Operators** execute different operations (e.g., string concatenation, addition of integer values or set operations) to finally create a string that is generated to the target. Data manipulation operators are treated as model access operators and use the same type of trace link, namely a **CreateTraceLink**, where the source artefact(s) refer(s) to the operator input and the target artefact to the result of the operator as part of the generated target.

iv. **Control Flow Operators** enable iterations (loops) and conditional execution (if statement) of certain template parts. In order to extract traceability data, we map the source, the control flow target holds for, to the target generated through one iteration of the control flow operation in accordance with the same ATG as for Model Access Operators.

In Xpand, FOREACH-statements expand the body of the FOREACH block for each element of the target collection that results from the expression:

```
<<FOREACH expression AS variableName
 [ITERATOR iterName] [SEPARATOR expression]>>
 a sequence of statements using variableName
 to access the current element of the iteration
<<ENDFOREACH>>
```

Thus, we define a **CreateTraceLink** between the model element the expression refers to and the target collection that results from the expression, as denoted by the following graph:

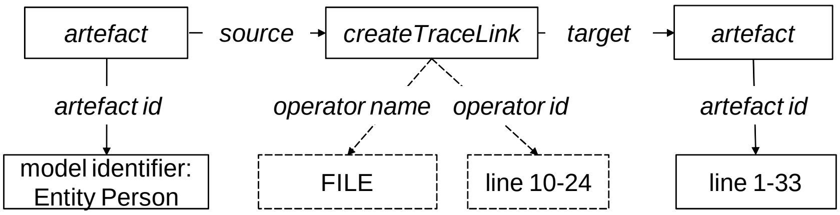


v. **Modularization Operators** allow structuring of complex templates into subtemplates similar to procedures in programming languages and into separate files using an include mechanism like in modular programming. Examples of operators in Xpand, for a) generating files, b) defining template units and c) expanding templates are given below.

a) The Xpand FILE-statement invokes the generation of an output file from its body statement to the specified target, where the "expression" denotes the file name in the statement syntax:

```
<<FILE expression [outletName]>>
a sequence of statements
<<ENDFILE>>
```

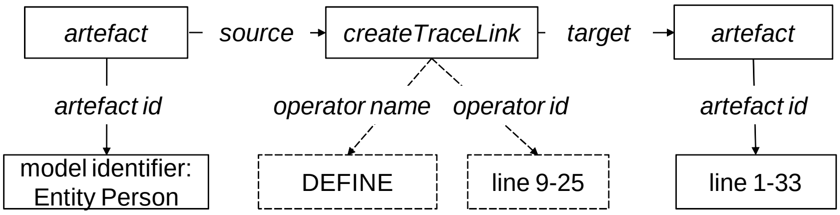
Therefore, we derive a **CreateTraceLink**, which connects the currently processed model element as source artefact and the actual file in the file system as the target artefact. We show this as example by generating the Java class *Person* from the model element *Entity Person* as shown in the following graph:



b) As example, we take the Xpand DEFINE statement (also called block). This DEFINE block is the smallest identifiable unit in a template file, where the tag consists of a name, an optional comma separated parameter list as well as the name of the metamodel class for which the template is defined.

```
<<DEFINE templateName(formalParameterList) FOR MetaClass>>
a sequence of statements
<<ENDDEFINE>>
```

The methodology proposes to define a **CreateTraceLink** from the metamodel (class) instance to the generated target defined through the beginning and ending of the define statement. For example, regarding the DEFINE statement in Line 9–25, a create link is defined from the model element *Entity Person* to the Java class *Person* as shown in the following graph:



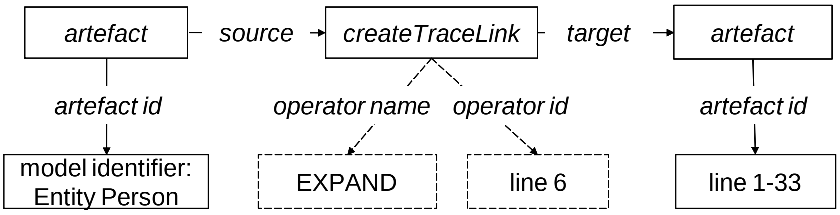
c) The EXPAND statement is an example of template expansion and acts as a subroutine call. While expanding a DEFINE block (in a separate variable context), the EXPAND statement inserts its output at the current location and continues with the next statement.

```
<<EXPAND definitionName [(parameterList)]
[FOR expression | FOREACH expression [SEPARATOR expression] ]>>
```

Thus, we derive a **CreateTraceLink** from the currently processed expression to the generated target defined through the *definitionName*. Given the following statement from the template (Line 6) of the illustrative example,

```
<<EXPAND javaClass FOREACH entities()>>
```

then, a CreateTraceLink is defined from the *Entity Person* (as currently processed Entity) to the generated target defined through the template module *javaClass*, as depicted below. In this context, the *javaClass* corresponds to the Java class *Person*:



vi. **Query Operators** A model query effectively navigates through a model and generates a collection of model elements that satisfy one or more properties, for example, a model query generating a collection of class elements that have exactly one feature. The operator logic accessing the source model may have different forms [CH06]. For example, the operator could be a declarative query as in OCL (e.g., Acceleo [Foua]) or XPath [Lan], or in another case witten in Java accessing the API provided by the internal representation of the source model.

For query operators a **RetrieveTraceLink** is defined from the set of returned source model-elements to the template snippet defining the query.

We now turn to the traceability of rules, that is, the coarse-grained view on modelling. Regarding model-to-text transformation languages, the definition of a transformation rule is not always represented syntactically in the transformation program, for example, as in the case of Xpand. Cleary distinguishable are the rules, for example in MOF-Script. In the latter case, the rule input and output are traced, by linking the processed source model element(s) (rule input) to the generated text (rule output) with a **CreateTraceLink**. The trace link carries the rule *name* and *identifier* as properties in analogy to the above cases.

II Model-to-Model Transformations: Rule-based transformations of model-to-model transformations work differently than hard-coded rules of model-to-text transformations with respect to the output of the rule directed through the rule's target domain. In model-to-text transformations the rule output does not correspond to a model element, but is degenerated to a string. In fact, the rule output is a concatenation of strings due to the operators. However, in model-to-model transformations, the output corresponds to one or more model elements directed through the target domain. Thus, we need to track the input and corresponding output of a certain rule. Regarding our proposal on modelling, this reflects the coarse-grained traceability of rules. Yet, on this level, the properties of model elements that might have been set through the rule are not traced. This follows on the level of fine-grained traceability based on operators. Furthermore, regarding the following proposal for model-to-model transformations, we distinguish between mapping and update transformations.

Update transformations Regarding update transformations, model elements may be modified, deleted or added. Accordingly, the methodology accounts for update, delete and create links, respectively, as described in the classification on source and target relationships from Chapter 5, Section 5.2. Following this straightforward approach, there are, however, some points to note, for avoiding dangling edges in the traceability graph. We present example transformations in ATL to demonstrate our approach, essentially proposing the following two patterns on *extension-only* and *destructive* update transformations.

i. **Extension-only Rules** Extension-only rules allow for updating existing model elements and/or creating new model elements, for example, the following ATL transformation that changes the visibility of attributes and adds corresponding getters:

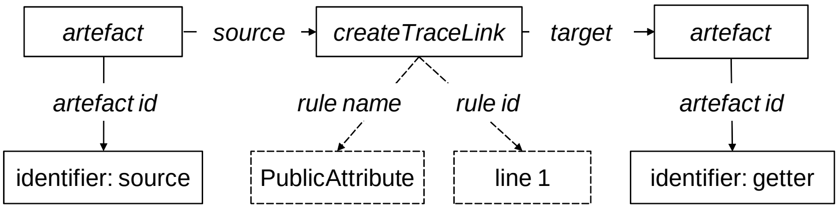
```
rule PublicAttribute {
  from
      source : ClassDiagram!Attribute (
          not s.isPrivate and
          not s.owner.op->exists(o | o.name = 'get' +
          s.name.toUpperCase() and o.returnType = s.type)
      )
  to
      target : ClassDiagram!Attribute (
          name <- s.name,
          owner <- s.owner,
          isPrivate <- true,
          type <- s.type
      ),
      getter : ClassDiagram!Operation (
          name <- 'get' + s.name.toUpperCase(),
          owner <- s.owner,
          isPrivate <- false,
          returnType <- s.type
      )
}
```

Listing 6.3: Extension-only Update Transformation in ATL

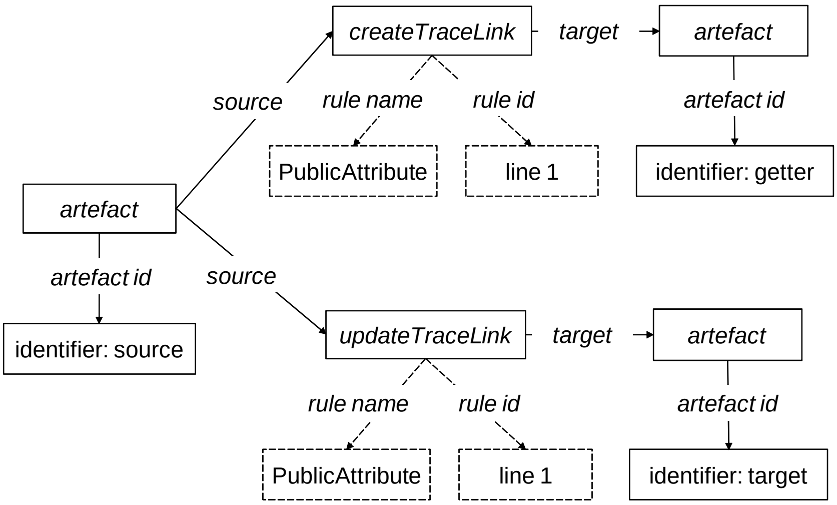
Regarding the modelling of traceability data, we distinguish between two cases:

- Element creation - where the generated element is not represented through a metamodel element in the rule input, e.g., in Listing 6.3, the element *getter* is not represented in the from clause.
- Element update - where the generated element is represented through the same metamodel element in the rule input as well as output, e.g., in Listing 6.3, the elements *target* and *source* represent the same metamodel element *Attribute*.

Since there is no corresponding source artefact in the first case, our approach uses the given rule input as pseudo-source artefact to avoid dangling edges. Furthermore, a **CreateTraceLink** from the given rule input as source artefact is linked to the newly generated element. In the above example, this corresponds to the following traceability graph:



In case of updating an element an **UpdateTraceLink** is defined from the to-be updated element and the generated target element. In addition to the given graph above, the following traceability graph holds:



ii. **Destructive Rules** For a destructive update as well as in-place transformation, we define a **DeleteTraceLink** between respective source elements and updated target elements to record that model elements are modified through a delete operation, as supported in some endogenous transformation approaches, e.g., the refinement mode in ATL. Traceability data is persisted in terms of trace links referencing source and target artefacts. In case of deleted model elements, the outdated references (of deleted elements) are persisted to track the state of deletion even though the references are technically invalid. Further processing on the invalid references is necessary for the maintenance of traceability data, yet this is out of scope of this work next to other possibilities on recording deleted elements.

```
rule PrivateAttribute {
  from
      s : ClassDiagram!Attribute (s.isPrivate and
          s.owner.op->exists(o|o.name = 'get' +
          s.name.toUpperCase() and o.returnType = s.type)
          )
  to
      t : ClassDiagram ! Attribute (
          isPrivate <- false
      )
}
rule DeleteOperation {
  from
      s : ClassDiagram!Operation (s.owner.attr
          ->exists(a|a.name = s.name.toUpperCase().substring(3, s.name->size())
              and a.isPrivate))
  to
      drop
}
```

Mapping transformations For mapping transformations, only **CreateTraceLinks** are used, since all target elements are newly created.

In summary, Table 6.3 gives an overview on the patterns used to define trace links for rules in model-to-model transformations:

Rule		**Link Type**
Update	i. Extension-Only	CREATE/UPDATE
	ii. Destructive	DELETE
Mapping		CREATE

Table 6.3.: Derivation of Traceability Data out of Rules

Regarding the fine-grained level of traceability based on operators, we need a classification as in the case of model-to-text transformations. Model-to-model transformation approaches mostly use an embedded query language, such as OCL or a dialect of OCL [BCW12]. Hence, model-to-model transformations use the same classification on operators as presented above for operator class i.-iii. and vi. (cf. Table 6.2), since they may be represented in OCL. *Static Text* may be presented through the basic types of

OCL, *Model Access Operators* as OCL queries and *Data Manipulation Operators*, for example through OCL concatenation. *Query Operators* use OCL queries. Furthermore, the classes iv. and v. are aligned with the QVT standard. Examples for *Control Flow Operators* are the *when* and *where* or *forEach* clause in QVT as well as the *foreach*[1] clause in ATL. Regarding *Modularization Operators* (v.), a simple separation of a call to a rule and its definition is possible in QVT. In ATL, rules can be grouped in several files and be composed by the superimposition mechanism. Thus, in summary, the same classification on operators for model-to-text transformations may be used for model-to-model transformation and with this the same proposal on traceability data modelling.

Iterative Execution and Hyperedges Both proposals for modelling a traceability graph (cf. Section 5.4) need to be enhanced for operators as well as rules that are executed iteratively. According to our approach, iteratively executed operators (and rules) are modelled through the use of hyperedges in the traceability graph, for example, control flow operators in connection with model access operators. Recalling from our running example, the expression «f.name» (e.g., in Listing 6.1, Line 13) is executed for every feature, namely *name*, *age* and *address*, regarding the generation of the *Person* class. Due to the iterative nature of the control flow operator, all source as well as target artefacts due to all iterations of the «f.name» operator are linked through a **CreateTraceLink**, which is modelled as a hyperedge, as indicated in the graph of Figure 6.6.

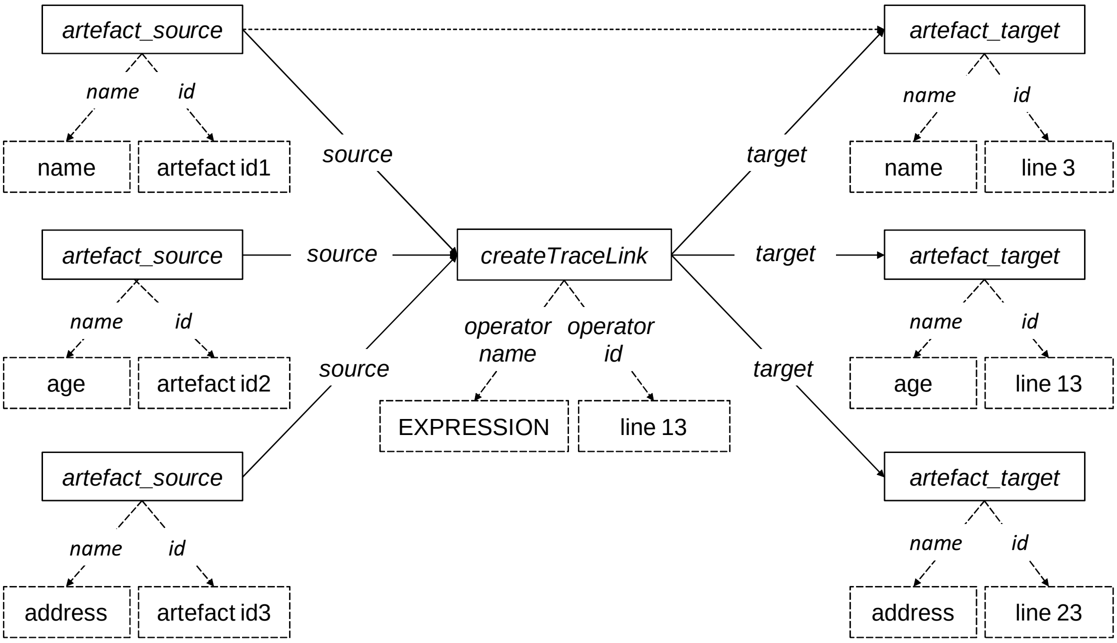


Figure 6.6.: Modelling of Hyperedges and Iterative Execution

---

[1] In newer ATL versions the execution flow is controlled by using lazy rules or imperative code.

Our definition of traceability requests to track corresponding source and target artefacts due to a particular transformer. Yet, this also implies the need for assigning source and target artefacts due to *one* iteration of the control flow operator. Hyperedges do not include this information per default, for example, in the graph below, it is not clear which of the three target artefacts was generated from a specific feature. Without this information, a user may interpret that a given source artefact, e.g., the feature *name* is transformed into the three target artefacts, field *name*, field *age* and field *address*, which is incorrect information, for a change impact analysis, for example. One possibility to solve this issue, is to introduce a mapping from each feature to its generated output due to one iteration, as depicted in the graph below by the dotted arrow, exemplified for the feature *name*. Furthermore, this mapping may be added to the properties of the trace link itself. The above-mentioned approach applies to rules the same way as for operators.

We now turn back to our roadmap and continue with the modelling of containment relationships.

1. d) Containment Relationships

Apart from the trace links leading to the traceability graph structure due to the nature of transformers, a second type of trace link is proposed. This type results from the implicitly given containment relationships of the source and target artefacts. According to the traceability metamodel, we propose to use the type **ContainmentTraceLink** for all containment relationships. The ContainmentLink is defined as a subtype of the RetrieveLink, since a containment relationship may be interpreted as a model query returning a subset of model elements or text sequences (cf. Section 5.2 on the derivation of CRUD links).

The primary motivation for adding trace links to the trace graph due to containment relationships, is for reasons of maintenance and *system comprehension* (traceability scenario). Looking at the illustrative example, not having a trace link between the *Entity Person* and its contained *feature name* and deleting the *Entity Person* results in a lack of information for reasoning about updates on the traceability graph. Consequently, all subelements of the Entity (i.e., features and their dependencies) would remain in the traceability graph. In analogy, this applies to the target artefacts, for example for generated static text strings.

Regarding the collected traceability data for the traceability scenario, system comprehension, the above modelling not only allows for understanding, which source and target artefacts are related due to a certain transformer, yet, also the hierarchy of containment relationships inherent to these artefacts.

2. Automation of Extraction

Furthermore, we base our approach on the following train of thought leading to the invasive nature of the connector. The implementation of the GTI requires the adaption of a transformation engine for the explicit generation class. Therefore, we adopt the concept of aspect-oriented programming [EFB01] to augment the transformation engine with traceability-specific code to instrument the engine for a traceability solution. The use of *aspects* has the advantage of easier maintenance for this traceability-specific code. Furthermore, its integration is achieved automatically.

In applying aspect-oriented programming to the extraction phase, we encapsulate the mapping of transformation execution to traceability data (in terms of traceability-specific code) into *aspects*. The mapping results by way of the identified patterns on the modelling of traceability data. In this way, the *advice* of an aspect contains this mapping.

Generally applying to model-transformation approaches, at runtime, the transformation engine instantiates the abstract syntax tree (AST) of the transformation language, including the rules and operators. The execution of each model transformation necessitates for each rule and its containing operators, the invocation of specific internal methods (amongst others) directing the AST instantiation. Thus, we propose to use these internal methods for the *pointcut* definition to implement the mapping of model transformations to traceability data by analysing the AST at runtime according to our proposed patterns.

To be able to ”weave” this mapping, it is necessary to identify the coding in the transformation engine that is specific to the execution of rules as well as operators to derive a corresponding pointcut definition. This process depends on three factors. Firstly, the way the engine is implemented influences the pointcut definition, for example, if the above-mentioned internal methods are scattered, or, if there is a single point of entry to the AST instantiation. Secondly, the language used for the base code, where weaving takes place, needs to be compatible with the aspect language. Finally, the expressiveness of the pointcut language needs to be sufficient in order for weaving to take place in the correct parts of the base code.

In the following, we present an example integration based on AspectJ [EFB01] and Xpand. The Listing in 6.4 presents an *aspect* exemplary for the FILE statement of Xpand.

```
 1  @Aspect
 2  public class OawConnector extends AbstractTransEngineConnector {
 3    @After("execution(* * evaluate(..)) && this(fs) && args(ctx)")
 4    public void executeModelOperationInternal(FileStatement fs,
          XpandExecutionContext ctx) {
 5
 6      // Model Artefact
 7      Artefact sourceArtefact = new Artefact(...);
 8
 9      // File Artefact
10      Artefact targetArtefact = new Artefact(...);
11
12      // Trace Link
13      TraceLink link = new TraceLink(sourceArtefact, targetArtefact,
            CreateTraceLink);
14      this.executeModelOperation(link);
15    }
16
17  }
```

Listing 6.4: Pointcut Definition for FILE Statement

In Xpand, the execution of each model transformation necessitates for each Xpand Statment (operator), the invocation of a particular internal method (amongst others) directing the AST instantiation. In Line 3, the corresponding method *evaluate()* is used as pointcut definition. After the execution of this method, triggered by the FILE statement, the traceability-specific code is woven, which is captured through the *executeModelOperationInternal()* method (Line 4). This leads to the creation of corresponding source (of type Model) and target (of type File) artefacts based on the framework-internal XpandExectutionContext (Line 4). Finally, a **CreateTraceLink** between both artefacts is established (Line 13), followed by the invocation of the *executeModelOperation()* method. Recalling from Section 6.1, the latter call is required by the connector to trigger the typing phase.

6.3.3. Typing Phase

As mentioned in the beginning of this section, the typing phase depends on the traceability scenarios and starts with the execution of the *executeModelOperation()* method. Essentially, the typing phase involves the typing of the traceability graph after its structure is fixed through the extraction phase. Regarding the typing, we look at the constraints the traceability metamodel imposes on the typing phase. As depicted in Figure 6.5, the typing of the traceability graph, that is, of the artefact and link types (apart from the already defined CRUD-link types due to the extraction phase) is achieved through the region of the traceability metamodel right of the dotted line. Depending on the defined **Facets**, **FacetFactories** and the configuration of **Scope** and **Granularity**, the resulting traceability graph is typed as described in Chapter 5.

6.4. Blackbox Connectors

We recall that the implicit generation class already provides an integrated traceability solution, that is, traceability data is generated in parallel to the transformation execution. In order to make use of this advantage, the implementation of a blackbox connector is based on this traceability solution. In doing so, our approach proposes a model transformation from the traceability model generated from the transformation engine of the implicit class to an *instance of the GTF traceability metamodel*. With this, the connector implementation does not include an extraction phase (as opposed to the invasive connector), which is actually solved by the transformation engine. Thus, the blackbox connector only implements the typing phase, involving the typing of the traceability graph, as we will show in the following. For an illustrative example, we refer to Section 8.1, where we present a blackbox connector for QVT, being a representative of the implicit generation class.

6.4.1. Extraction Phase

With the extraction, as depicted in Figure 6.7, the transformation engine delivers the traceability graph structure in terms of its generated traceability model. The blackbox connector transforms this model into an instance of the *GTI traceability metamodel* through a mapping transformation, while preserving the traceability graph expressiveness. Apart from this predefined structure, containment relations need to be added to the connector output as argued for the invasive connector, if not already present. Possible sources for containment relations may be the traceability model or source and target models.

Regarding the typing of CRUD trace links, the same way of modelling applies as for the invasive connector. However, the applicability of each case depends on the expressiveness of the implicit traceability model, potentially missing information. In case the transformation creates new elements (mapping transformations), we propose to use create links. In case of an *update* or *in-place* transformation, update links and create links are used for extension-only rules (cf. update transformations of invasive connector). However, this requires that the implicit traceability model reflects the difference between an update and a create link. Generally, the traceability data needs to be expressive enough to gather the information as proposed for the invasive connector regarding update and in-place transformations. With respect to destructive rules, if the information of deletion is not persisted in the implicit traceability model, it is impossible to reproduce this information from the traceability model.

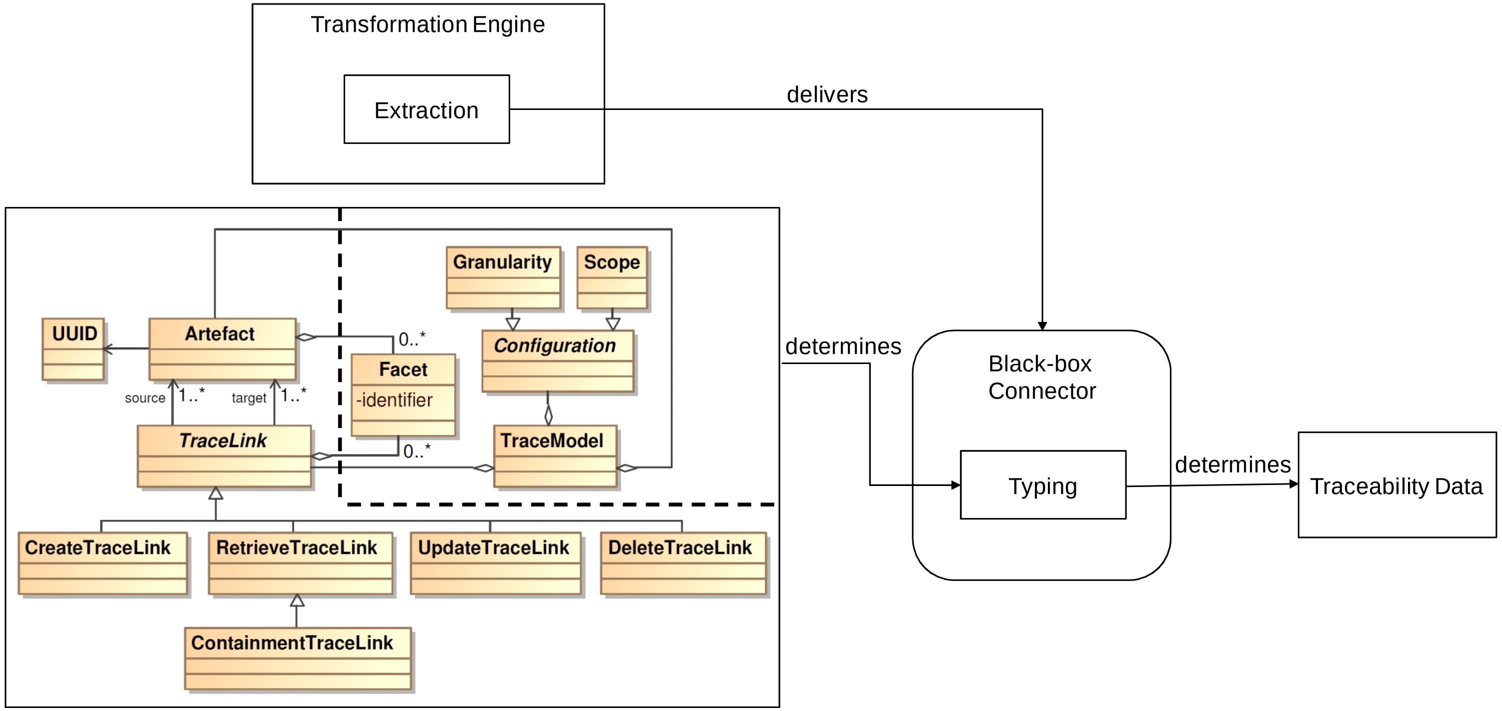


Figure 6.7.: Constraints of the Traceability Metamodel on the Blackbox Connector

6.4.2. Typing Phase

As in the case of the invasive connector, the typing phase concerns the typing of the traceability graph, i.e. the process is scenario-driven. In order to do this, the user has to define **Facets** and **FacetFacories** by using the above-mentioned model transformation. Hence, the constraints the traceability metamodel imposes on the blackbox connector concern the region of the traceability metamodel right of the dotted line in Figure 6.7. Consequently, the control over the expressiveness of traceability data with regard to artefact typing is limited to the traced entities of the implicit generation approach. It is only possible to define artefact types on the basis of existing artefact types in the traceability model due to the implicit generation approach (implicit traceability model). For example, in a model-to-text transformation, if the granularity level of traceability is kept on file level, it is not possible to capture a more fine-grained level, e.g., through a facet for text blocks within a file (cf. Figure 5.3). Thus, the control over traceability data is restricted to the granularity level of the integrated traceability solution, yet still allows for the flexibility to choose the artefact types on the basis of the given granularity level. This is achieved through the configuration setting of **Scope** and **Granularity**. The above-mentioned is valid for model-to-text as well as model-to-model transformations.

6.5. Contributions

In this chapter, we contribute the following:

C2: **Design Pattern on Augmentation of Model Transformations with Trace Link Generation**

We propose a methodology entailing three steps to augment arbitrary model transformations (model-to-model as well as model-to-text transformations) with a specific traceability mechanism. The methodology is based on a design pattern that can be realised through the generic traceability interface for arbitrary transformation engines.

Regarding the explicit and implicit generation class, we present two possible augmentation methods to achieve a traceability mechanism, respectively: a) Augmentation of the transformation engine based on aspect-oriented programming and b) Augmentation of the traceability-data output through the use of a model transformation. In terms of the generic traceability interface, these augmentation method require the implementation of two different kinds of connectors, that is, *blackbox* and *invasive* connectors for implicit and explicit generation, respectively.

Thus, the different kind of connectors realise two different kinds of traceability approaches:

- A priori traceability: Augmentation of the transformation engine before transformation execution

- A posteriori traceability: Augmentation of the traceability-data output after transformation execution

Once the augmentation method has been applied in either of the two cases, the transformation approach is featured with a traceability mechanism that generates traceability data conforming to the proposed traceability metamodel.

Furthermore, the methodology proposes a guideline for modelling traceability data based on a classification of operators applicable to model-to-model as well as model-to-text transformations.

7

Model Matching for Trace-Link Generation

The following chapter demonstrates how parallel model matching can be instrumented for the generation of trace links. This matching approach applies to the second and third category of trace link generation as per problem definition in Chapter 3, Section 3.2. In this case, the generation takes place *after* and *independently from* the execution of a model transformation.

We first explain the conceptual work of the model matching system in Section 7.1, followed by an overview on its realisation in Section 7.2.

7.1. Model-Matching System for Traceability

In this section, we describe our proposed model-matching system for traceability, essentially, taking two models as input and creating a mapping, that is, correspondences between model elements, as output. This mapping is further analysed with respect to the extraction of potential trace links.

Next, we explain the processing steps of our proposed matching system, which involves the following successive steps, as depicted in Figure 7.1:

1. **Import of Models**: The available source and target models need to be imported into a common data model, to have a common basis for arbitrary matching algorithms.
2. **Matching of Models**: Different matching algorithms are applied to the imported source and target models. A matching algorithm calculates a similarity value for

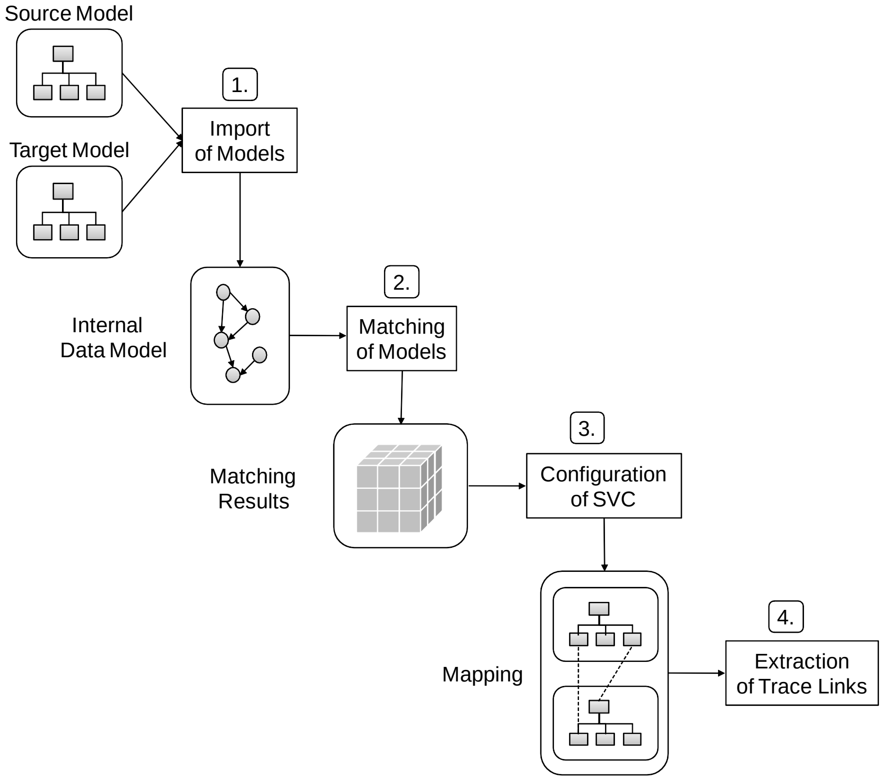


Figure 7.1.: Process Steps of the Model-Matching System for Traceability

all pairs of source and target elements. The matching system can be configured by selecting a set of matching algorithms involved in the matching process.

3. **Configuration of Similarity Value Cube**: Each matching algorithm provides separate results for a certain source and target model, where the results describe a similarity value for all source and target model element combinations, called similarity-value matrix. All of these matrices (one matrix due to a certain matching algorithm) are arranged into a cube, called similarity-value cube (SVC), as depicted in Figure 7.1. To derive a mapping between source and target elements out of these results, the similarity value cube needs to be configured, e.g., to form an *aggregation matrix* by calculating the average of similarity values, or *selection matrix* by selecting all elements exceeding a certain threshold. We provide a detailed description of possible configurations in Section 7.1.4.
4. **Extraction of Trace Links**: The resulting mapping is analysed and trace links are extraced according to certain heuristics, or configurations.

7.1.1. Step 1: Import of Models

The first process step requires models to be imported into a common data model for the sake of genericity. The idea behind this choice is to have a standard data model, upon which the matching algorithms operate. Alternatively, one could adapt the matching algorithms individually to be able to work on each source and target model potentially being from different languages. Yet, this would result in a higher implementation effort. Thus, we choose the first option and implement an importer for each modelling language.

To be able to base our work on the field of graph theory to make use of graph-based matching algorithms, we require the internal data model of our matching system to have a graph structure. Thus, for the model import, we need a graph formalism on the basis of which arbitrary models can be expressed uniformly as graphs, yet with an adequate expressiveness. Secondly, we require this formalism to express models in relation to their corresponding metamodels, since we make use of the *instance-of* relationship in our matching approach as explained in more detail in Section 7.1.3.

The formalism of *Typed Attributed Graphs* (TAG) (cf. Section 2.4, Definition 28) fulfils the above-mentioned requirements [EEPT06], concerning the universality and expressiveness of typed attributed graphs to represent models including *instance-of* relationships. Therefore, TAGs serve as the foundation of our graph construction.

The key idea is to model attributed graphs with node and edge attribution. On the basis of TAGs, we consider models as Attributed Graphs (AG) (cf. Definition 25) and metamodels as Attributed Type Graphs (ATG) (cf. Definition 26). Recalling, an AG is an E-graph combined with an algebra over a data signature, whereas an ATG is a special kind of AG used for typing of AGs. Thus, the ATG defines the set of all possible types used for typing an AG. Formally defined, an AG is typed over an ATG by an attributed graph morphism $t : AG \rightarrow ATG$. We provide an example in Section 7.1.2. For the graphical representation of models in terms of TAGs, we refer to Section 2.4.3.

Since we require the import of models as well as their referring metamodels, yet claim one internally used data model (as we will substantiate in the following train of thought), both models as well as metamodels need to be transformed into one internal data model. Since models and metamodels lie on different levels of abstraction, this has consequences for the import. In general, there are numerous possibilities for this transformation, as depicted in Figure 7.2. Per definition of a model transformation, instances of a metamodel are transformed into instances of another metamodel, where the metamodels may be different or the same. Thus, a model transformation is defined at the metamodel level.

If we follow possibility a) (called upper import) and specify the model transformation responsible for the import at the metalevel M_{n+2}, meaning that we specify the transformation program to transform instances of the metametamodel level to instances of the ATG (internal data metamodel), we solve the import for arbitrary metamodels for one modelling language. To be able to import model instances as well, which are on a

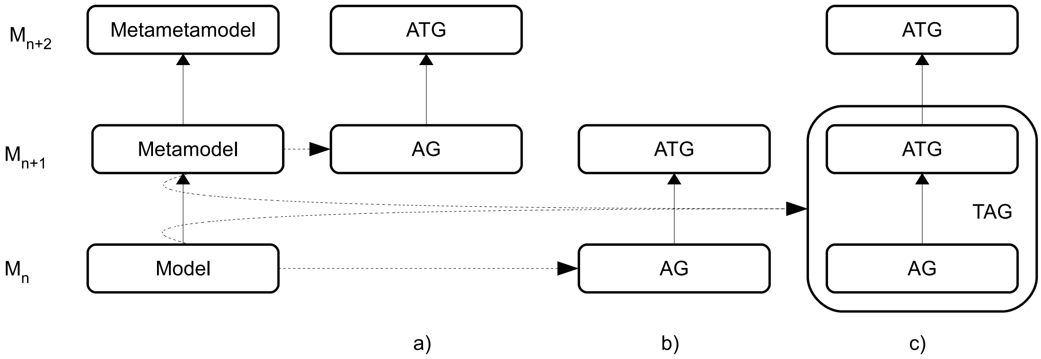


Figure 7.2.: a) Upper Importer, b) Lower Importer, c) Bridging Importer

lower level of abstraction as their corresponding metamodels, a model transformation at metalevel M_{n+1} is necessary, as shown in case b) (called lower import). Together with the upper as well as lower import, our goal of importing models and their referring metamodels is achieved, yet the implementation effort is still higher as the possibility portrayed under c). We propose to use one importer (bridging importer) per modelling language applicable to both the model as well as metamodel level. Essentially, we bridge the gap in the level of abstraction by regarding *metamodels* as instances of metametamodels and *models* as instances of instances of metametamodels. In this way a model and its metamodel are transformed into a TAG.

7.1.2. Running Example

To underline our matching process, we introduce a matching scenario, which we use across the following sections. The example is based on a model transformation from certain entities to object-oriented class specifications, the same as in Section 6.3.1. In Figure 7.3, the source and target model are represented as AGs (lower level), while their respective metamodel is depicted as ATGs (upper level). Each AG is typed over a particular ATG by a corresponding attributed graph morphism $t : AG \rightarrow ATG$. A formal definition is given in Appendix A.

We adopt the graphical notation as introduced in Section 2.4.2 for E-graphs. An E-graph has two different kinds of nodes, representing graph and data nodes, and three kinds of edges, the usual graph edges and special edges used for the node and edge attribution. The solid nodes and arrows refer to the graph nodes V_G and edges E_G, respectively. The dashed nodes are the (used) data nodes V_D and dashed arrows represent node and edge attribute edges E_{NA} and E_{EA}. In Figure 7.3 a), the ATG-Source Metamodel includes **Entities**, which contain **Features**. Both, **Entities** and **Features**, are characterised by a *name* of type **String**. The AG-Source Model (an instance of ATG-Source Metamodel) in c) includes an ***entity*** called *Person* owning two ***features***, carrying the names, *name* and *age*. The target metamodel in b) specifies **Classes** that consists of **Fields** and

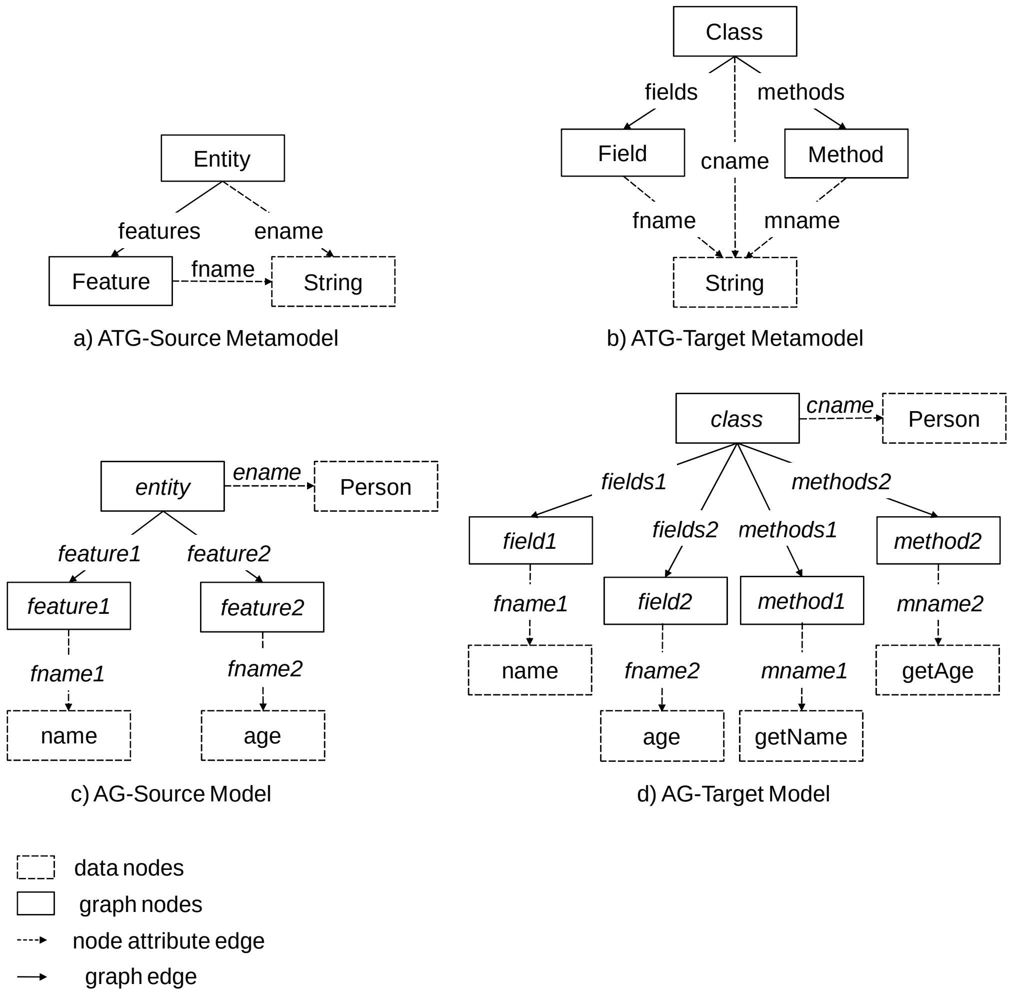


Figure 7.3.: Running Example for Model Matching

Methods. Again, all model elements own a *name* of type **String**. The *AG*-Target Model in d) describes a *Person* ***class*** containing two ***fields***, called *name* and *age* as well as according getter methods, namely *getName* and *getAge*.

7.1.3. Step 2: Matching of Models

In the following, we explain the matching process and different matching algorithms used. Model matching is the process of finding a mapping between two AGs based on a given similarity measure. In other words, the Cartesian product of the set of source as well as target *graph nodes* is generated. For each source-target combination, a similarity is calculated according to a pre-defined similarity measure. Finally, the resulting similarities are analysed to derive matches. Our approach deals with numerous ways of finding feasible similarity measures regarding the generation of trace links from the resulting mapping. For this purpose, we identified three measures to calculate similarity values:

1. **Attribute Similarity Measure**: Similar *data nodes* from source and target graphs, indicate shared characteristics, referred to as *attributes*, and thus, a potential similarity between the *graph nodes* that the *data nodes* are connected to.
2. **Connection Similarity Measure**: The similarity between a set of source and target children nodes acts as a similarity measure. The measure is based on the rationale that similar *children* graph nodes, have similar *parent* graph nodes. Thus, the connectivity of a graph node to its children graph nodes is used to propagate the similarity from child to parent node.
3. **Instance-of Similarity Measure**: We base the matching process on model level on the results of metamodel matching by making use of the *instance-of* relation. Thus, we investigate the outcome of propagating the similarity of metamodel elements to their conforming model elements.

In the next subsections, we propose a configurable base-matching algorithm making use of the above-mentioned similarity measures. Depending on the chosen similarity measure, the algorithm's functionality is defined. In Section 7.1.3, we introduce the attribute as well as connection similarity measure to configure matching algorithms for model (as well as metamodel) matching, followed by an example in Section 7.1.3. Thereafter in Section 7.1.3, the instance-of similarity measure is used for model matching on the basis of metamodel matching results, which we call *metamodel-driven model matching*. An example of the latter is given in Section 7.1.3.

Model Matching

The model-matching process with regard to the first two, above-mentioned similarity measures is outlined in Algorithm 1 and 2, as explained in the following. For the sake of clarity, we choose two algorithms for separating the matching process generally applying to graphs (Algorithm 1) from the similarity-value calculation of two graph nodes (Algorithm 2).

The core idea of Algorithm 1 is to match an attributed source and target graph and to return a mapping between corresponding source and target graph nodes. In more detail, the algorithm works as follows: The procedure MATCHGRAPHS (Line 1-8) matches two attributed graphs AG^1 and AG^2 on the basis of the similarity measure denoted with the variable $similarityMeasure$ from the set $\{\text{ATTRIBUTES}, \text{CONNECTIONS}\}$. In either way, the Cartesian product of source and target graph nodes is calculated (Line 2-3), thereby assigning to each Cartesian pair (s_i, s_j) a similarity value through the $matchNodes$ function (Line 4 resp. 9-16) and depending on the similarity measure chosen in Line 1. Accordingly, all similarity values are arranged in a similarity value matrix denoted by $SIM_{|V_G^1||V_G^2|}$ (Line 4), such that the following holds: A certain similarity value of (s_i, s_j) is assigned to a cell, denoted by sim_{ij}, from $SIM_{|V_G^1||V_G^2|}$.

The RETRIEVEMATCHES function (Line 7) receives $SIM_{|V_G^1||V_G^2|}$ as input and renders a mapping between source and target graph nodes. There are numerous ways to this mapping calculation, depending on the configuration of the SVC. For the details of this calculation, we refer to Section 7.1.4 and Appendix B.3.

In case the similarity values are calculated according to the ATTRIBUTES similarity measure, the MATCHATTRIBUTES function (Line 11) is called. Accordingly, the MATCHCONNECTEDNODES function (Line 13) is called for the CONNECTIONS similarity measure. The detailed working of these functions is described in Algorithm 2.

Algorithm 1. Matching of two attributed graphs

```
Require: $AG^1 = (G^1, D^1)$ and $AG^2 = (G^2, D^2)$
Require: $G^1 = (V_G^1, V_D^1, E_G^1, E_{NA}^1, E_{EA}^1, (source_j^1, target_j^1)_{j \in \{G,NA,EA\}})$
Require: $G^2 = (V_G^2, V_D^2, E_G^2, E_{NA}^2, E_{EA}^2, (source_j^2, target_j^2)_{j \in \{G,NA,EA\}})$
Require: $D^1 = (S_D^1, OP_D^1)$ and $D^2 = (S_D^2, OP_D^2)$
Require: $similarityValues = \{r | r \in \mathbb{R} \text{ and } 0 \leq r \leq 1\} \cup \{\text{UNKNOWN}\}$
Ensure: $similarityMeasure \in \{\text{ATTRIBUTES}, \text{CONNECTIONS}\}$

 1: procedure MATCHGRAPHS($AG^1, AG^2, similarityMeasure$)
 2:     for all $s_i \in V_G^1$, $i = \{1, \ldots, |V_G^1|\}$ do
 3:         for all $t_j \in V_G^2$, $j = \{1, \ldots, |V_G^2|\}$ do
 4:             $SIM_{|V_G^1||V_G^2|} \ni sim_{ij} \leftarrow matchNodes(s_i, t_j, similarityMeasure)$
 5:         end for
 6:     end for
 7:     $matches \leftarrow retrieveMatches(SIM_{|V_G^1||V_G^2|})$
 8: end procedure

Ensure: $s_i \in V_G^1$
Ensure: $t_j \in V_G^2$

 9: function MATCHNODES($s_i, t_j, similarityMeasure$)
10:     if $similarityMeasure = \text{ATTRIBUTES}$ then
11:         return $matchAttributes(s_i, t_j)$
12:     else if $similarityMeasure = \text{CONNECTIONS}$ then
13:         return $matchConnectedNodes(s_i, t_j)$
14:     end if
15:     return UNKNOWN
16: end function

end
```

Through Algorithm 2, the similarity of two graph nodes is calculated as follows. In case Algorithm 1 uses the similarity measure ATTRIBUTES, the similarity value assigned to a certain Cartesian pair (s_i, t_j) is calculated through the MATCHATTRIBUTES function (Line 4–13). In particular, the set of data nodes of s_i as well as t_j is rendered (Line 5–6), denoted by $Nodes_{source}^{ATT}$ resp. $Nodes_{target}^{ATT}$, followed by the calculation of the Cartesian product of these sets (Line 7–8). For each such Cartesian pair (of data nodes), the degree of similarity is calculated through the COMPUTESIMILARITY function (Line 12 resp. 1-3) and placed into a similarity matrix $SIM_{Max(k)Max(l)}^{ATT}$ (Line 9). Afterwards, the resulting similarity matrix is reduced to a single similarity value according to a certain selection strategy by applying a set similarity function called *computeSetSimilarity* (Line 12). This value is returned as similarity value of the two given graph nodes (s_i, t_j). We assume the existence of the computeSetSimilarity function to calculate a single similarity value from the matrix of similarity values due to the source and target data nodes of a certain graph node pair (s_i, t_j). In general, there are several possibilities to calculate such a value. We provide an example on page 105, where the average value is calculated. For a description of possible configuration strategies, we refer to Appendix B.3.

Alternatively, the similarity of two graph nodes is calculated on the basis of the similarity measure CONNECTIONS and thus, the MATCHCONNECTEDNODES function (Line 14–23) is called. The function takes as input a Cartesian pair (s_i, t_j) from Algorithm 1, renders the set of all children graph nodes from s_i (Line 15) resp. from t_j (Line 16), denoted by $Nodes_{source}^{CON}$ resp. $Nodes_{target}^{CON}$ and calculates the Cartesian product of these sets (Line 17–18). For each such Cartesian pair (of graph nodes) a similarity value is calculated on the basis of the function MATCHATTRIBUTES and placed into a similarity matrix $SIM_{Max(p)Max(q)}^{CON}$ (Line 19). Finally, the *computeSetSimilarity* reduces this matrix to a single similarity value, which is assigned to the two given initial graph nodes (s_i, t_j).

Remark The above two algorithms are applicable for a given source and target ATG as well, since an ATG can be interpreted as an AG (cf. Definition 26).

Algorithm 2. Similarity of two graph nodes

```
Require: $compare : D_s^1 \times D_s^2 \rightarrow similarityValues$
Ensure: $dataNode_{source} \in D_s^1 \subseteq V_D^1$
Ensure: $dataNode_{target} \in D_s^2 \subseteq V_D^2$

function computeSimilarity($dataNode_{source}, dataNode_{target}$)
    return $compare(dataNode_{source}, dataNode_{target})$
end function

Ensure: $s_i \in V_G^1$
Ensure: $t_j \in V_G^2$

function matchAttributes($s_i, t_j$)
    $Nodes_{source}^{ATT} \leftarrow \{target_{NA}^1(e) | e \in E_{NA}^1$ and $source_{NA}^1(e) = s_i\}$
    $Nodes_{target}^{ATT} \leftarrow \{target_{NA}^2(e) | e \in E_{NA}^2$ and $source_{NA}^2(e) = t_j\}$
    for all $s_{i_k} \in Nodes_{source}^{ATT}$ , $k = \{1, \ldots, |V_D^1|\}$ do
        for all $t_{j_l} \in Nodes_{target}^{ATT}$, $l = \{1, \ldots, |V_D^2|\}$ do
            $SIM_{Max(k)Max(l)}^{ATT} \ni sim_{i_k j_l} \leftarrow computeSimilarity(s_{i_k}, t_{j_l})$
        end for
    end for
    return $computeSetSimilarity(SIM_{Max(k)Max(l)}^{ATT})$
end function

function matchConnectedNodes($s_i, t_j$)
    $Nodes_{source}^{CON} \leftarrow \{target_G^1(e) | e \in E_G^1$ and $source_G^2(e) = s_i\}$
    $Nodes_{target}^{CON} \leftarrow \{target_G^2(e) | e \in E_G^2$ and $source_G^2(e) = t_j\}$
    for all $s_{i_p} \in Nodes_{source}^{CON}$, $p = \{1, \ldots, |V_G^1| - 1\}$ do
        for all $t_{j_q} \in Nodes_{target}^{CON}$, $q = \{1, \ldots, |V_G^2| - 1\}$ do
            $SIM_{Max(p)Max(q)}^{CON} \ni sim_{i_p j_q} \leftarrow matchAttributes(s_{i_p}, t_{j_q})$
        end for
    end for
    return $computeSetSimilarity(SIM_{Max(p)Max(q)}^{CON})$
end function

end
```

Example: Model Matching

In the following, we demonstrate the above model matching process on the basis of the similarity measure ATTRIBUTES as well as CONNECTIONS in accordance with the running example from Section 7.1.2. Essentially, we match the two AGs depicted in Figure 7.4.

According to the procedure (Algorithm 1, Line 1), the Cartesian product of the set of all source *graph nodes* (***entity***, ***feature1***, ***feature2***) and the set of all target *graph nodes* (***class***, ***field1***, ***field2***, ***method1*** and ***method2***) is calculated (Line 2-3), yielding a similarity matrix of 15 cells.

If the similarity values are derived on the basis of the similarity measure ATTRIBUTES, the MATCHATTRIBUTES function (Algorithm 2, Line 4) is invoked by the MATCHNODES function (Algorithm 1 Line 4). Thus, the *data nodes* of each Cartesian pair are retrieved (Algorithm 2, Line 5-6) and their similarity values are calculated according to the *computeSimilarity* function (Algorithm 2, Line 9). For the sake of simplicity, we assume this function to calculate string similarity values on the basis of the following definition:

Given the data sorts $D_s^1 = D_s^2 = String$ (where a sort is defined as a certain label type used for attribution, i.e., String, cf. Section 2.4.2) with $v \in D_s^1$ and $w \in D_s^2$, we define the function $compare : D_s^1 \times D_s^2 \rightarrow similarityValues$ such that the following holds:

$$compare(v, w) = \begin{cases} 1, & \text{if } v = w \\ 0,5, & \text{if } v \subseteq w \\ 0, & \text{if } v \neq w \end{cases}$$

For example, for the Cartesian pair (***entity***, ***class***), the data node *Person* is retrieved for the source graph nodes ***entity*** as well as for the target graph node ***class*** (cf. Figure 7.4). Since their labels (i.e., strings) are identical, a similarity value of 1 is assigned to the data node pair (*Person*, *Person*). Since the resulting similarity-value matrix SIM_{11}^{ATT} (Algorithm 2 Line 12) contains only one cell with the similarity value 1, the *computeSetSimilarity* functions assigns the same value to the node pair $(entity, class)$, as depicted in Table 7.1. For the Cartesian pair (***feature1***, ***class***) a similarity value of 0 is calculated, since the labels of the data nodes *name* and *Person* are unequal (cf. Figure 7.4). In analogy, the similarity values of the other Cartesian pairs are calculated, as depicted in Table 7.1. The resulting mapping of the graph nodes due to the similarity measure ATTRIBUTES is depicted by the dashed lines in Figure 7.4.

	class	field1	field2	method1	method2
entity	1	0	0	0	0
feature1	0	1	0	0,5	0
feature2	0	0	1	0	0,5

Table 7.1.: Similarity Values of Graph Nodes

In case the similarity values are calculated on the basis of the similarity measure CONNECTIONS, the MATCHCONNECTEDNODES function (Algorithm 2, Line 14) is invoked through the MATCHNODES function (Algorithm 1, Line 4). This entails the retrieval of all children graph nodes per Cartesian pair of graph nodes. Their similarity

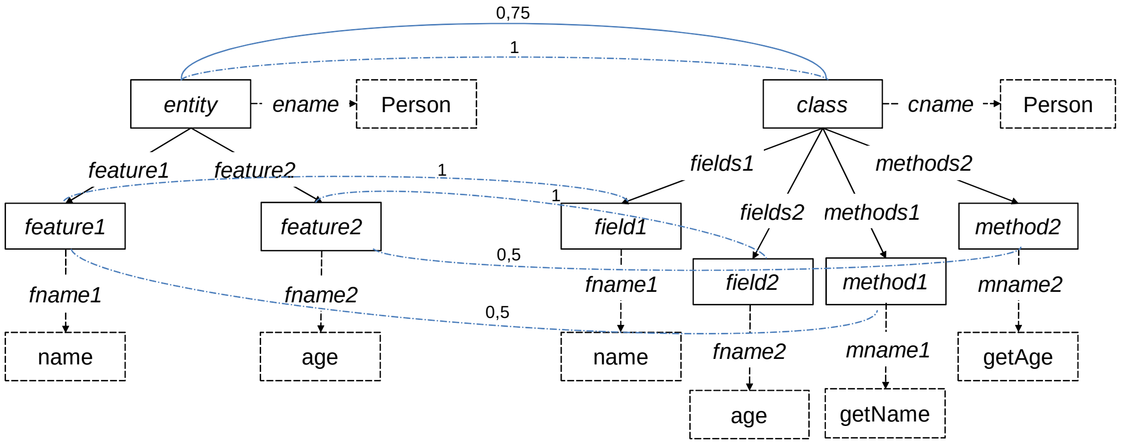


Figure 7.4.: Model-Matching Mappings for AG-Source Model and AG-Target Model

values are computed according to the matchAttribute function (Algorithm 2, Line 19). For example, the graph nodes ***feature1***, ***feature2*** resp. ***field1***, ***field2***, ***method1*** and ***method2*** are retrieved for the source graph node ***entity*** resp. target graph node ***class***. Traversing the Cartesian product of the retrieved sets, we assign to each Cartesian pair a similarity value on the basis of the similarity measure ATTRIBUTES as shown in Table 7.1. The resulting similarity value matrix (Algorithm 2, Line 19) essentially is the 2×4 matrix from Table 7.1, except for the first column and row. For this example, we assume for the *computeSetSimilarity* function (Line 22) to reduce this matrix to a similarity value by taking the average of all matching results from corresponding source-target children nodes. Thus, a similarity value of $0,75$ is calculated and propagated to the node pair (***entity***, ***class***). Afterwards, the similarity of all remaining Cartesian pairs of graph nodes are calculated analogously. Since ***feature1*** and ***feature2*** do not contain any related children graph nodes, no further similarity values are calculated. Thus, as a resulting mapping, only one mapping is retrieved by the *retrieveMatches* function in Algorithm 1 (Line 7), as depicted in Figure 7.4 by the full line.

Metamodel-driven Model Matching

In the following section, we introduce an algorithm to realise the idea, how mappings due to metamodel matching may be used for improving model matching results. Essentially, this improvement can be achieved by verifying or rejecting found matches based on the metamodel mapping. This idea is called metamodel-driven model matching and implements the third similarity measure, called INSTANCEOF. For this purpose, we extend the algorithms in Section 7.1.3 by the similarity measure INSTANCEOF. Additionally, in order to use of the *instance-of* relationship between metamodels and models, which is a necessary step in the process of metamodel-driven model matching, we base the algorithms on TAGs. In analogy to Section 7.1.3, we again introduce two algorithms for the sake of clarity. While Algorithm 3 focuses on the matching process of graphs, Algorithm 4 deals with the matching process of graph nodes.

The procedure matchTypedGraphs in Algoritm 3 works according to the procedure in Algorithm 1, calling the matchGraph function to work on source and target AGs, which in turn calls the matchNodes function, (Algoritm 3, Line 4). However, the latter function is extended by the use of the similarity measure INSTANCEOF (Line 9-10). In case this parameter is used, the matchNodeTypes function is evoked, which is defined in Algorithm 4. The matchNodeTypes function takes a certain Cartesian pair (s_i, t_j) of graph nodes from Algorithm 3 as input, thus working on respective source and target AGs. For each such pair (s_i, t_j) the corresponding graph nodes $Node_{source}^{ATG\_Type}, Node_{target}^{ATG\_Type}$ of source and target ATGs are returned by the attributed graph morphism t (Line 2-3). For each pair $Node_{source}^{ATG\_Type}, Node_{target}^{ATG\_Type}$ a similarity value is calculated through the matchNodes function (Algorithm 4, Line 4), the same way as in Algorithm 1. As a consequence, the similarity value of $(Node_{source}^{ATG\_Type}, Node_{target}^{ATG\_Type}$ is propagated to (s_i, t_j) (Algorithm 1, Line 4). We restrict the use of the similarity measure INSTANCEOF to avoid potential cycles.

Algorithm 3. Matching of two typed attributed graphs

```
Require: $AG^1 = (G^1, D^1)$ and $AG^2 = (G^2, D^2)$
Require: $G^1 = (V_G^1, V_D^1, E_G^1, E_{NA}^1, E_{EA}^1, (source_j^1, target_j^1)_{j \in \{G,NA,EA\}})$
Require: $G^2 = (V_G^2, V_D^2, E_G^2, E_{NA}^2, E_{EA}^2, (source_j^2, target_j^2)_{j \in \{G,NA,EA\}})$

Require: $ATG^1 = (TG^1, Z^1)$ with $Z^1$ as final algebra of $D^1$
Require: $ATG^2 = (TG^2, Z^2)$ with $Z^2$ as final algebra of $D^2$
Require: $TG^1 = (\mathcal{G}^1, Z^1)$ and $TG^2 = (\mathcal{G}^2, Z^2)$
Require: $\mathcal{G}^1 = (\mathcal{V}_G^1, \mathcal{V}_D^1, \mathcal{E}_G^1, \mathcal{E}_{NA}^1, \mathcal{E}_{EA}^1, (Source_j^1, Target_j^1)_{j \in \{G,NA,EA\}})$
Require: $\mathcal{G}^2 = (\mathcal{V}_G^2, \mathcal{V}_D^2, \mathcal{E}_G^2, \mathcal{E}_{NA}^2, \mathcal{E}_{EA}^2, (Source_j^2, Target_j^2)_{j \in \{G,NA,EA\}})$

Require: $TAG^1 = (AG^1, t^1)$ over $ATG^1$ with $t^1 : AG^1 \rightarrow ATG^1$
Require: $TAG^2 = (AG^2, t^2)$ over $ATG^2$ with $t^2 : AG^2 \rightarrow ATG^2$

Require: $similarityValues = \{r | r \in \mathbb{R} \text{ and } r \geq 0 \text{ and } r \leq 1\} \cup \{\text{UNKNOWN}\}$
Ensure: $similarityMeasure \in \{\text{ATTRIBUTES}, \text{CONNECTIONS}, \text{INSTANCEOF}\}$

procedure matchTypedGraphs($TAG^1, TAG^2, similarityMeasure$)
    $matchGraphs(AG^1, AG^2, similarityMeasure)$
end procedure

Ensure: $(s_i, t_j) \in (V_G^1 \times V_G^2) \cup (\mathcal{V}_G^1 \times \mathcal{V}_G^2)$

function matchNodes($(s_i, t_j), similarityMeasure$)
    if $similarityMeasure = \text{ATTRIBUTES}$ then
        return $matchAttributes(s_i, t_j)$
    else if $similarityMeasure = \text{CONNECTIONS}$ then
        return $matchConnectedNodes(s_i, t_j)$
    else if $similarityMeasure = \text{INSTANCEOF}$ then
        return $matchNodeTypes(s_i, t_j)$
    end if
    return UNKNOWN
end function

end
```

Algorithm 4. Similarity of two typed graph nodes

```
Ensure: $s_i \in V_G^1$ and $Node_{source}^{ATG\_Type} \in \mathcal{V}_G^1$
Ensure: $t_j \in V_G^2$ and $Node_{target}^{ATG\_Type} \in \mathcal{V}_G^2$
Ensure: $typeSimilarityMeasure \in similarityMeasure \setminus \{\text{INSTANCEOF}\}$

1: function matchNodeTypes($s_i, t_j$)
2:     $Node_{source}^{ATG\_Type} \leftarrow t^1(s_i)$
3:     $Node_{target}^{ATG\_Type} \leftarrow t^2(t_j)$
4:     return $matchNodes(Node_{source}^{ATG\_Type}, Node_{target}^{ATG\_Type}, typeSimilarityMeasure)$
5: end function
end
```

Example: Metamodel-based Model Matching

We illustrate the above two algorithms according to our running example in Section 7.1.2 with corresponding source and target AGs and their referring ATGs. In doing so, we assume a given metamodel mapping according to Algorithm 1 working on source and target ATGs, since we have illustrated its execution already in the previous example (see p. 105). The final metamodel mapping is depicted in the upper layer of Figure 7.5 with a mapping from **Entity** to **Class** as well as from **Feature** to **Field** and **Method**. Regarding this mapping, we assume the similarity values to be equal to 1. Secondly, we set the similarity measure to INSTANCEOF in Line 1 of Algorithm 3 to demonstrate the newly introduced similarity measure.

First, the Cartesian product of the set of all AG-source graph nodes (***entity***, ***feature1***, ***feature2***) and the set of all AG-target graph nodes (***class***, ***field1***, ***field2***, ***method1*** and ***method2***) is calculated, yielding a similarity matrix of 15 cells. For each Cartesian pair, the matchNodes function (Algorithm 1, Line 4) from the matchGraph procedure (Algorithm 1, Line 1) is invoked, returning a similarity value on the basis of the matchNodeType function (Algorithm 3, Line 10). For example, for the graph nodes ***entity*** and ***class***, the attributed graph morphism t (Algorithm 4, Line 2-3) yields the graph nodes **Entity** and **Class** (being the metamodel types), respectively. Since a mapping exists between **Entity** and **Class** (as an outcome of the matchNodes function (Line 4), the similarity value, 1, is propagated to the pair (***entity***, ***class***). Regarding the graph nodes ***entity*** and ***field1***, the following similarity is calculated. Since their according types **Entity** and **Field** do not match, the similarity of source and target graph node is set to 0. Analogously, the similarities of the other Cartesian pairs is calculated and assigned to the final similarity matrix (Algorithm 1, Line 4) as depicted in Table 7.2. As a result, matches are computed (Line 7).

Due to the propagation of similarities of metamodel elements to corresponding model elements, numerous false mappings are computed, such as ***feature1*** to ***field2*** and

	class	field1	field2	method1	method2
entity	1	0	0	0	0
feature1	0	1	1	1	1
feature2	0	1	1	1	1

Table 7.2.: Similarity Values of AG-Graph Nodes

method2 as well as ***feature2*** to ***field1*** and ***method1***. This motivates the fact to combine matching results due to differently configured matching algorithms to filter out false mappings and to ideally render a correct and complete mapping as depicted in the lower level of Figure 7.5. In the following section, we will show a specific combination of matchers and configuration of the SVC that provides the matching results as portrayed in Figure 7.5.

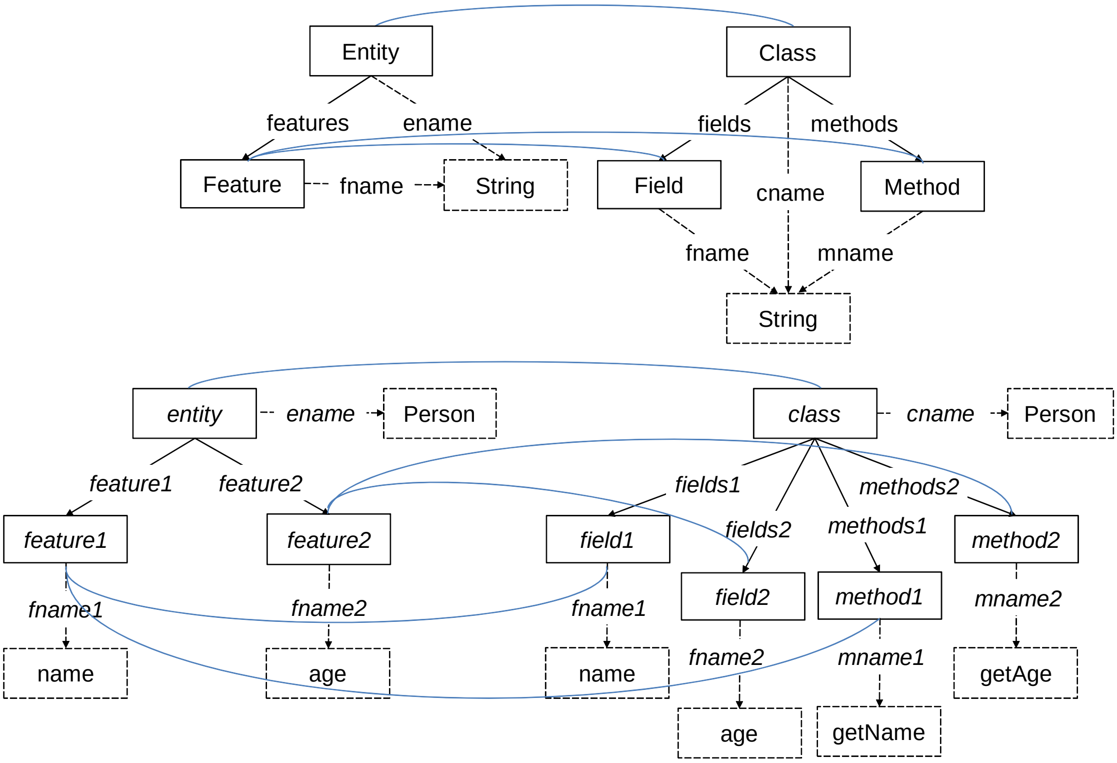


Figure 7.5.: Mappings of Metamodel-driven Model Matching

7.1.4. Step 3: Configuration of Similarity Value Cube

As motivated in the previous example (see p. 110), there is a need to configure the SVC. This is achieved in terms of the strategies and parameters listed in Appendix B.3.

In the following, we demonstrate the *aggregation* and *selection* strategy regarding the running example. Let us assume, the results of the attribute (cf. Table 7.3) and instance-of similarity measure (cf. Table 7.4) from the previous examples. By applying the aggregation strategy, *Average*, the above-mentioned matrices are aggregated to the aggregation matrix in Table 7.5. Thereby, the average value is calculated for each source and target node. Next, we use the selection strategy, *Threshold*, by setting the threshold to $0,75$. Hence, the resulting selection matrix is obtained as presented in Table 7.6. This matrix represents the configured SVC and corresponds to a complete and correct mapping as required and presented in the lower level of Figure 7.5.

	class	field1	field2	method1	method2
entity	1	0	0	0	0
feature1	0	1	0	0,5	0
feature2	0	0	1	0	0,5

Table 7.3.: Results of Attribute Similarity Measure

	class	field1	field2	method1	method2
entity	1	0	0	0	0
feature1	0	1	1	1	1
feature2	0	1	1	1	1

Table 7.4.: Results of Instance-of Similarity Measure

	class	field1	field2	method1	method2
entity	1	0	0	0	0
feature1	0	1	0,5	0,75	0,5
feature2	0	0,5	1	0,5	0,75

Table 7.5.: Aggregation Matrix

	class	field1	field2	method1	method2
entity	1	0	0	0	0
feature1	0	1	0	0,75	0
feature2	0	0	1	0	0,75

Table 7.6.: Selection Matrix

7.1.5. Variations of Metamodel-driven Model Matching

This section describes the dimensions of metamodel-driven model matching. We investigate how metamodel matching may improve the results of model matching by propagating the similarity of metamodel elements to their conforming model elements. In this way, we examine three possible ways of influence, differing in a) the *quality* of provided metamodel mappings and b) the point in *time*, when applied to the model matching process.

Model element mappings due to model matching may be filtered, that is, validated, if a correct metamodel mapping exists, or rejected otherwise. This requires a correct and complete metamodel mapping to be beneficial for the quality of the mapping.

Besides the quality of the metamodel mappings, the point in time can be varied, when metamodel element mappings are used in the matching process. We analyse three possibilities: a) before matching, referred to as *blocking*, b) during matching, referred to as *instance-of* (as demonstrated in Algorithm 3 for the similarity value INSTANCEOF) and c) after matching, referred to as *filtering*.

In the following, we elaborate on the principle understanding of the above-mentioned three possibilities:

- **Blocking**: The key idea of blocking is to reduce the combinations of source and target elements of the Cartesian product, being a necessary step in the matching process. According to [KR09] blocking is needed for large inputs to reduce the search space for model matching from the Cartesian product to a small subset of the most likely matching pairs. Blocking techniques typically use a key to partition the model elements to be matched into blocks. In the case of applying metamodel matching, this key can be derived based on metamodel element matches. In doing so, all source elements conforming to a certain source metamodel element and all target elements conforming to a certain target metamodel element are assigned to a block.

  Thus, by the process of blocking only model elements assigned to a block and with a corresponding metamodel mapping are incorporated into the matching process and found mappings are constraint to model elements whose metamodel elements match (provided a correct and complete metamodel mapping). As a consequence, complexity is reduced. This can be seen in our running example, where the complexity is reduced from 15 calculations to 5.

- **Instance-of Matching**: During the matching process, metamodel matching can be applied in two ways: a) as an independent matcher, or b) as a filter mechanism applied on the results of existing matchers. The first approach is to implement a matching algorithm on the basis of the *instance-of* similarity measure as described in Section 7.1.3. The similarity of a given source and target model element is derived from the similarity of the according source and target metamodel elements.

Assuming a correct and complete metamodel mapping, the resulting similarity would be either 0 (no similarity) or 1 (equivalence).

The second approach to instance-of matching, filters the results of existing matchers. In particular, existing matchers calculate the similarity between model elements as usual, yet, afterwards the result can be invalidated due to the non-existence of a metamodel match.

Both approaches influence the computed similarities during the matching process. In contrast to the blocking mechanism, the instance-of matchers has an impact on both the similarity of certain source and target model elements and potentially on other source-target combinations. The latter is due to the aggregation and selection strategies applied on the SVC.

- **Filtering**: Filtering refers to the process of applying metamodel matching results to the resulting mappings obtained from model matching. In this way, the resulting model mapping may be validated with respect to a metamodel mapping. In particular, all found model matches are checked for an appropriate metamodel match. If an according match is found the match is kept, otherwise it is removed from the mapping. As a consequence, the resulting model mapping exclusively contains matches which have an according metamodel match. In summary, this mechanism has the same effect as blocking, yet does not reduce complexity.

For an evaluation of the above-mentioned approaches with respect to matching quality, we refer to Section 9.2.

7.1.6. Step 4: Extraction of Trace Links

The result of the matching process is a mapping from source to target elements. During the process of extraction, traceability data is derived from this mapping. Numerous possibilities on how this derivation works are proposed in the following. Common to all derivation possibilities is the use of additional information. Given that, the derivation entails an analysis of the additional information, which may be an automatic, or a manual process. Finally, the extracted traceability data is populated to the traceability repository.

Automatic Analysis

The automatic analysis may be based on *existing traceability data* to increase data consistency, or *stochastic data* to deliver filtered matches. Both analyses are to improve the quality of found matches.

- Existing traceability data: The extraction may rely on existing data from a given repository for traceability data. The repository is used to perform consistency checks on the basis of existing traceability data and the newly proposed state of data due to the extraction process. In doing so, potential duplicates and dependency cycles can be calculated and avoided in the traceability data by filtering the to-be extracted matches.

- Stochastic data: The extraction of traceability data is based on the distribution function of found matches (result of matching process) according to their similarity value. The distribution of certain similarity values is analyzed to derive traceability data, for example, by applying a threshold to filter out matches with a similarity value below the threshold, or more generally, by allowing matches from a certain quantile. In both cases, the input mapping is filtered and the remaining matches are populated to the traceability repository.

Manual Analysis

Beyond the automatic processes, the extraction may be based on user-specific data. This form of analysis is a manual process as opposed to the formerly-mentioned. In effect, the user is provided with a suggestion system over the resulting matches to analyse and evaluate these, for example, by incorporating domain knowledge. Afterwards, the user-selected matches are extracted to the traceability repository. The matching process potentially delivers incorrect and incomplete matching results (cf. Section 8.2) due to insufficient information in models for matching, or missing context information [Voi11]. Therefore, the suggestion system gives way to another optimization to improve the quality of matches and thus, the traceability data.

In summary, the above-mentioned possibilities may also be combined to a semi-automatic process. However, further investigation on their feasibility is out of scope of this work. Regarding their implementation, each possibility (or combination) essentially is a realisation of the **IConnector** in terms of the *GTI* (cf. Section 6.1). The *extraction phase* concerns the derivation of trace links out of a given mapping due the above-mentioned possibilities, followed by the *typing phase*, which takes these trace links—referencing source and target artefacts—as input for the *executeModelOperation()* function. As a result, trace links from matched source and target models are populated to registered traceability engines of the GTI. Therefore, this connector is a blackbox connector. The matching system can be interpreted as an engine delivering traceability data and thus, realising the extraction phase, whereas the blackbox connector implements the typing phase as depicted in Figure 7.6.

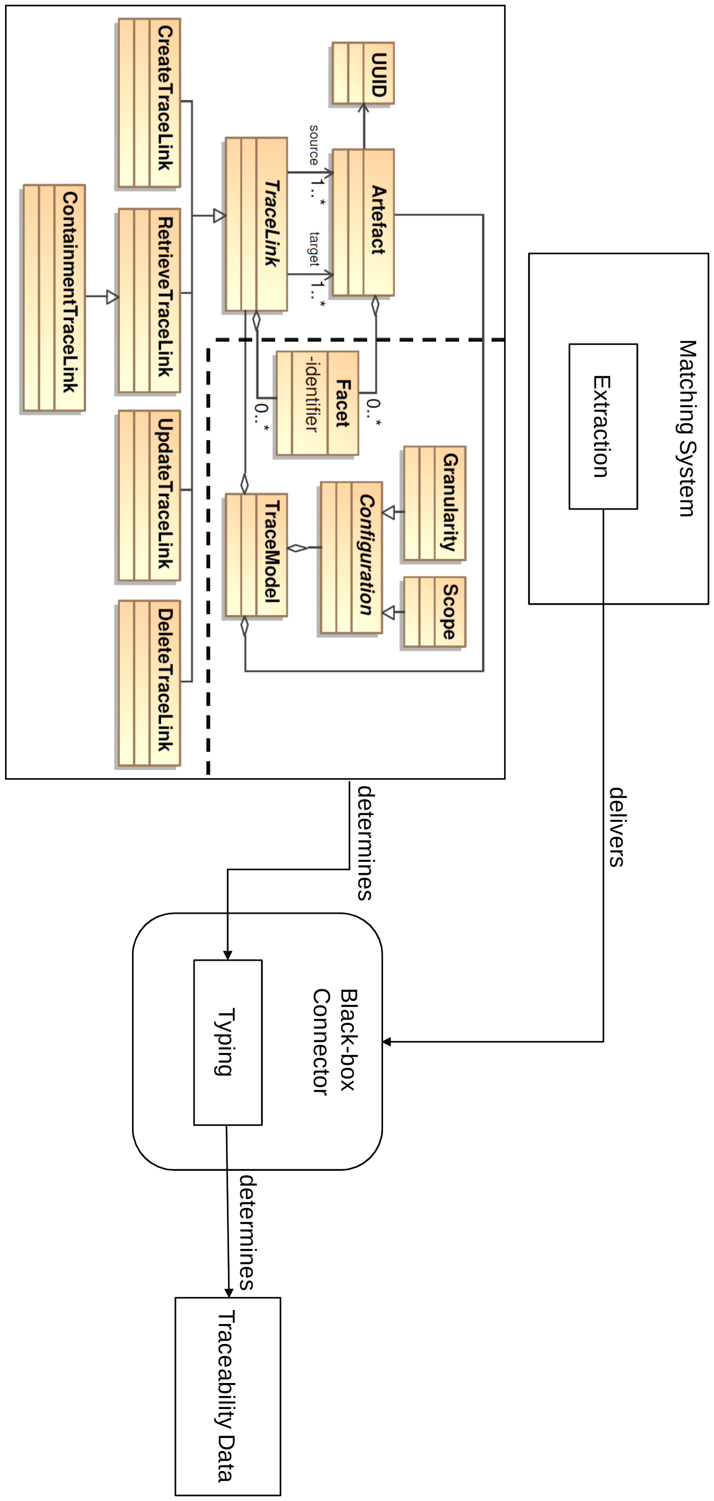


Figure 7.6.: Blackbox Connector for Matching System

7.2. Implementation of the Model-Matching System

In this section, we provide an overview on the implementation of the model matching system as introduced in Section 7.1. The realization of this matching component is based on a metamodel matching framework, called *Matchbox*. This framework is build upon the *SAP Auto Mapping Core*, an implementation inspired by the schema matching framework, COMA++ [DR07]. We refer to Appendix B for a detailed description on the used matchers and the approach on parallel matching of Matchbox. The reason for choosing Matchbox is its language genericity and broad scope of optimized metamodel matchers, fully aligned with our conceptual work on the three similarity measures for model matching. Based on Matchbox, we introduced 8 matchers for model matching in realization of these three similarity measures: *Name* and *name path* matcher in terms of the **Attribute Similarity Measure**; *children*, *parent*, *leaf*, *sibling*, *graph edit distance* and *pattern* matcher being structural matchers and variations of the **Connection Similarity Measure**. Furthermore, we implemented an *instance-of* matcher based on the **Instance-of Similarity Measure**, which works in analogy to the *data type* matcher. Apart from instance-of matching, we investigated *blocking* and *filtering* techniques as part of metamodel-driven model matching.

The main adaptions to Matchbox concern the introduction of model matchers. Model instances differ in structure and content compared to metamodels and thus requires a different approach to leveraging model-specific characteristics for matching. These main characteristics are:

- Instances provide *values* of attributes.
- Instances provide an instance-of relation.
- Instances may contain other instances, e.g., attributes, which characterise the containing instance.

Matchbox provides several existing matchers, which were built for the purpose of metamodel matching (cf. Appendix B). Yet, these do not make use of the above-mentioned characteristics. As a consequence, an instance-specific version of each metamodel matchers is implemented. In the following sections, we shed light on the model import and new versions of matchers with respect to the above-mentioned characteristics.

7.2.1. Model Import

The importers of Matchbox hold for Ecore, XSD and OWL. Each importer transforms models of the according language into the internal data model of Matchbox, called *Genie* (cf. Appendix B). Regarding our work, these importers need to account for metamodels as well as for model instances.

Since the internal data model of Matchbox is able to express models and metamodels as TAGs and for the sake of generality, we abstract from technical names in the following discussion. For a technical description, we refer to [Kas11]. In order to make use of model-specific characteristics, the import follows the description of Section 2.4.3 for model instances and their corresponding metamodels, that is, for metamodels, we base the import on flattened containment relationships, while for model instances on containment relationships.

In addition, we summarise the most important aspects of the model import with respect to:

- ambiguousness of labels for graph nodes
- mandatory attribute values
- compatibility of tree structure

Regarding metamodel matching, numerous matchers, as described in Appendix B.2, are based on the *name* and *name path* matcher, which essentially are string-based matching algorithms. The algorithms are founded on the assumption that graph nodes of source and target with the similar labels refer to similar model elements corresponding to the graph nodes. Thus, the metamodel-element name is an important source of linguistic information for metamodel matching. By way of import of metamodels, the label of a graph node is set to the metamodel element's name. However, for model instances, the element name does not guarantee uniqueness, since numerous instances of a metamodel element carry the same name (or null, if not set). This has implications for the attribute matcher as explained in Section 7.2.2 below.

In Section 2.4.3, we have described, how the labels of graph nodes and data nodes are set. The import of model instances (representing classes, attributes, references etc.) results in a graph node, where its label is set to the metamodel element's name and a corresponding data node with its label set to the value of the instance. In terms of the running example from Figure 7.3, an attribute instance, for example, the feature with value *name* (denoted as *feature1*) is represented as a graph node, where the label is set to the metamodel element's name *Feature*[1], while the corresponding data node's label is set to *name*. The allocation of this value property is mandatory for attributes and reference for corresponding representations in the graph model.

In order to be compatible with previous versions of Matchbox, in particular, with respect to the existing matchers, parent, children and leaf matcher, working more efficiently on a tree, the internal data model of Matchbox features a primary tree structure. Essentially, this upward compatibility requires the transformation of a graph into a tree, as discussed in Section 2.4.3. Regarding model instances, the primary tree structure is based on containment relationships, while for metamodels, the flattened containment mode is chosen.

---

[1] We have chosen the label *feature1* over *Feature* for the sake of better readability to be able to differentiate between different features, which would carry the same label.

7.2.2. Applied Model Matchers

Matchbox provides several existing matchers which were built for the purpose of metamodel matching. As mentioned in the beginning of this section, the metamodel matchers (cf. Appendix B) do not make use of the above-mentioned model-specific characteristics and thus need to be extended. As a consequence, an instance-specific version of each metamodel matcher is implemented. In the following, we list the adapted matchers and state their characteristics.

Attribute Similarity Measure

The *name* and *name path* matcher for metamodel matching calculate string similarities based on the names of metamodel elements. As mentioned under Section 7.2.1 regarding the import of model instances, the labels of graph nodes are set to their corresponding metamodel element names. Consequentially, numerous instances of a metamodel element carry the same name (or null, if not set) and are not unique and thus, the name matcher would calculate false similarities. Therefore, the names of elements alone do not suffice for name matching on instance-level. To tackle, this shortcoming, we additionally analyse the *values* of attributes. Not every instance's value is set. Yet, instances may contain attribute instances with set values. In fact, a very important characteristic of attributes is that, not only the name of an attribute is set, but also its value. We use this information and define the attribute similarity measure as such: Two model elements are similar, if their containing attributes have a similar *name* and *value*. The resulting matcher is called *attribute* matcher and replaces the *name* and *name path* matcher used on metamodel-level. In contrast to the latter matchers, the attribute matcher uses a value matcher, which is an adaption of the name matcher. Essentially, the value matcher calculates the string similarities on the values of instances. Since a *value* (cf. Appendix B, data model of Matchbox) is a list of strings, rather than a single string (as in the case of the *entityName*), the algorithm uses an additional aggregation step.

Connection Similarity Measure

Regarding the connection matcher, we take following relationships (i.e., containment and reference) of instances into account. Inheritance relationships are not needed at the level of models. The connection matcher essentially derives the similarity of two instances based on the similarity of its related instances. To derive the similarity of related instances, the connection matcher uses the attribute matcher. In Matchbox, the *children*, *parent*, *sibling* and *leaf* matcher, all employ the *name* and *name path* matcher (cf. Appendix B). These are adapted by replacing the *name* and *name path* matcher with the attribute matcher.

The graph matchers of Matchbox, namely *graph edit distance* and *pattern* matcher, use an additional internal data model, whose creation requires instance-specific adaptions. In the following, we highlight these adaptions.

The *graph edit distance* matcher (GED) calculates the similarity of two given model elements based on their connection in a common subgraph. The GED matcher of Matchbox is an approximate subgraph isomorphism algorithm based on Neuhaus and Bunke [NB04]. Since the calculation of the maximal common subgraph is NP-complete [GXTL10], the GED of Matchbox calculates a lower bounded graph edit distance, instead of the minimal graph edit distance for two given input graphs. The input graphs are restricted to be planar. In doing so, the problem of complexity can be solved in almost quadratic time. If the graphs are non-planar, these are transformed to planar graphs by carrying out a minimal set of graph modifications. As an adjustment to metamodel matching, the GED requires a seed mapping (set of correct matches), which is used as an initial mapping for the similarity calculation in order to increase the quality of matching results. For a detailed description of the algorithm, we refer to [Hei10, Voi11].

The *pattern* matcher calculates the similarity of two given model elements based on reoccurring patterns (information) in the source and target model. The underlying matching algorithm is based on the premise that model elements, which are part of the same pattern in different models are assumed to be similar. Patterns indicate redundant information or the usage of established design structures. Both scenarios are accounted for in terms of the pattern matcher. A comprehensive description follows in [Voi11, Muc10]. First, the algorithm needs to identify patterns (pattern mining), after subsequently matching these patterns in order to identify possible element mappings.

Adaptions to the graph matchers relate to the model import, particularly to adaptions of the internal graph model used by the graph matchers of Matchbox. In the first place, this graph model needs to be enhanced with respect to model instances, not being required for metamodel matching and thus not present. For metamodel matching, metamodel elements are transformed into graph nodes with a corresponding label. Both, GED and pattern matcher, use these label names for matching. Yet as argued above, according to the current implementation of the model import, this name does not characterise model instances uniquely. Therefore, the setting of the label is based on the *value* of instances. As a result, the labels of graph nodes are more expressive for matching on instance level.

Furthermore, in order to provide a seed mapping for the GED matcher, a specific flag as attribute of the imported model elements is set to true for corresponding root model elements of source and target model. The GED matcher uses a mapping between both root elements as seed mapping.

Instance-of Similarity Measure

The realisation of the instance-of similarity measure requires the implementation of a new matcher, the *instance-of* matcher. Since the internal data model of Matchbox reflects the instance-of relationship between model and metamodel elements, no further adaptions to the data model are necessary in order to propagate similarity values of metamodel elements to their corresponding instances. The *instance-of* matcher uses 4 other matchers in order to compute the similarity of metamodel elements, that is, *name* matcher, *name path* matcher, *parent* matcher and *leaf* matcher. This choice and number of matchers is founded on the work of [DR02, Muc10]. Their evaluations show matcher combinations of $3-4$ matchers rendering best matching quality as opposed to combinations with a higher or lower number. The adoption of graph matchers was omitted due to runtime concerns.

7.3. Contributions

This chapter contributes the following:

C3: Parallel Model Matching for Trace-Link Generation

The main contribution of this work is a traceability solution based on parallel model matching for arbitrary source and target models. This solution is proposed in addressing the problem of *lacking traceability data due to non-existing or inaccessible transformation engines* (blackbox systems). Aligned with the approach on blackbox and invasive connectors, this is an *a posteriori* traceability approach, since the blackbox system is extended by the matching system and its resulting mapping is augmented afterwards through a blackbox connector.

Furthermore, this contribution is based on a novel, language-agnostic concept, defining three similarity measures for model matching to generate trace links:

1. **Attribute Similarity Measure**: Similar *data nodes* from source and target graphs, indicate shared characteristics, referred to as *attributes*, and thus, a potential similarity between the *graph nodes* that the *data nodes* are connected to.
2. **Connection Similarity Measure**: The similarity between a set of source and target children nodes acts as a similarity measure. The measure is based on the rationale that similar *children* graph nodes, have similar *parent* graph nodes. Thus, the connectivity of a graph node to its children graph nodes is used to propagate the similarity from child to parent node.
3. **Instance-of Similarity Measure**: We base the matching process on model level on the results of metamodel matching by making use of the *instance-of* relation. Thus, we investigate the outcome of propagating the similarity of metamodel elements to their conforming model elements.

In realisation of this concept, 8 matching algorithms are implemented on the basis of the three similarity measures: *Name* and *name path* matcher in terms of the Attribute Similarity Measure; *Children*, *parent*, *leaf*, *sibling*, *graph edit distance* and *pattern* matcher being structural matchers and variations of the Connection Similarity Measure and finally, an *instance-of* matcher based on the Instance-of Similarity Measure.

As common to parallel matching systems, the matching results due to different matching algorithms are combined. In doing so, we base our work on the metamodel matching system from Voigt et al. [Voi11] to apply specialised graph-based matching using planar graphs with the benefit of improving matching results. Since metamodel matching does not incorporate attribute values of model instances, the metamodel matching algorithms from [Voi11] are extended to instantiate the proposed similarity measures.

8

Evaluation

The evaluation of this work concerns two parts. First, we evaluate the connector-based approach in terms of the generic traceability interface in Section 8.1. Secondly, we evaluate our approach regarding the model matching system for traceability in Section 8.2. In this way, we evaluate the two main components of the generic traceability framework.

The evaluation for both components was performed on a laptop running Java 1.6.0.22 64-bit on 4 Intel i5 cores with 2.4 GHz each (OS Windows 7, 64 bit). The main memory comprises 4 GB with 2 GB assigned to Java.

8.1. Methodology for Automatic CRUD Trace-Link Generation

In this section, we evaluate the connector-based approach on invasive and blackbox connectors regarding the first category. The evaluation strategy is based on building representative connectors and evaluating their generated traceability data against reference data. The strategy is further described in Section 8.1.1, followed by the evaluation setup in Section 8.1.2. Next, we present the examined connectors in Section 8.1.3. Furthermore, we evaluate the conduction of traceability scenarios on the generated data due to the connectors in Section 8.1.4. Finally, we conclude this section with a summary of our results in Section 8.1.5.

8.1.1. Evaluation Strategy

The goal of the evaluation for the first category is to show that the methodology for connector development can account for traceability solutions based on invasive and blackbox connectors. Therefore, the resulting traceability data is evaluated against a reference mapping, called gold mapping, which is used as a frame of reference to analyse the quality of the connector-generated data.

Secondly, to acquire representative results, the following prerequisites are adhered to:

1. evaluate diverse language scope to prove language-independency
2. evaluate with respect to the dichotomy of trace link generation classes, to show the feasibility of both connector approaches (invasive and blackbox connector)
3. evaluate scenarios with model-to-model and model-to-text transformations, to demonstrate the difference in granularity in terms of traceability-data modelling (cf. rule-based versus operator-based tracing).

Thirdly, the connector-based solutions need to deliver traceability data with a sufficient expressiveness to conduct traceability scenarios successfully. The traceability scenarios taken into account for this evaluation are: *System Comprehension*, *Coverage Analysis*, *Change Impact Analysis*, *Orphan Analysis* and *Transformation Debugging*. Since traceability scenarios are views on the traceability graph as argued in Section 5.4, we need to show that the resulting traceability graph has an adequate expressiveness to execute the above traceability scenarios successfully.

8.1.2. Evaluation Setup

In the following two sections, we describe the evaluation scenarios and the derivation of the gold mapping.

Evaluation Scenarios

To cover a broad language scope (1.) with representatives from both generation classes (2.), the following language spectrum was chosen as summarized in Table 8.1. The first row refers to the illustrative example introduced previously in Section 6.3.1. This example is based on a Xpand transformation, *Person2JavaClass*, rendering Java classes from a given source model representing a person. For this scenario an invasive connector is evaluated. We extend this scenario with a range of Xpand model-to-text transformations from an SAP business application called the *Sales Scenario*. For a detailed description, we refer to Appendix C.

With respect to the Sales Scenario, the evaluation is demonstrated for the feature *Quotation Management* (QM) exemplary for one domain-specific language (DSL) to avoid repetition, since it suffices to implement the connector for all Xpand statements relevant to traceability, that is, according to the operator classification in Section 6.3. This operator-complete reference implementation works analogously for all DSLs and features of the Sales Scenario. Therefore, we chose the *View DSL* as its transformations include a representative of each operator class. This DSL is used to model the graphical user interface for the Sales Scenario including the Quotation Management capabilities. The source and target artefacts of the selected transformation as well as the transformation program are listed in Appendix D.1. The transformation program consists of 135 lines of code and generates two Java classes with 32 and 48 lines of programming code.

Furthermore, we evaluate blackbox connectors for QVT and ATL. The former connector is evaluated against a QVT transformation, *QVTO UML2RDB*, receiving an UML source model and issuing an instance of a relational database model as specified in Appendix E.1. The ATL connector is tested for a range of 41 model-to-model transformations from the ATL Zoo [Fouc].

Therefore, the evaluation scenarios include model-to-model and model-to-text transformations (3.). In total, the evaluation entails three different connectors (one per language). Accordingly, we provide a description of each connector in Section 8.1.3.

Generation Class	Language	Connector Type	Scenarios
Explicit	Xpand	Invasive	Person2JavaClass
Explicit	Xpand	Invasive	Sales Scenario
Implicit	QVT	Blackbox	QVTO UML2RDB
Implicit	ATL	Blackbox	ATL Zoo

Table 8.1.: Language Representatives

Evaluation Gold Mapping

The resulting traceability data of the connectors is evaluated against a certain gold mapping, which is used as a frame of reference to analyse the quality of the connector-generated data. In this sense, the gold mapping represents what the connector is supposed to deliver. The derivation process of the gold mapping is manual. For the explicit generation class (invasive connectors), this gold mapping is derived by analysing the transformation program and identifying its operators and rules. For each operator (or rule, if applicable), corresponding trace links are defined according to the methodology from Chapter 6. Regarding the implicit generation class, the gold mapping is derived from the existing traceability model by capturing its expressiveness. Additionally, we add the expected CRUD link types according to the methodology.

This derivation process is executed for a representative from each generation class, that is, for the selected transformation from the Sales Scenario as described above and the *QVTO UML2RDB* transformation. The resulting gold mappings for these transformations are listed in Appendix D.2 and E.2, respectively.

8.1.3. Evaluation Connectors

This section presents three different connectors for Xpand, QVT and ATL, respectively.

Xpand Connector

The Xpand connector (invasive connector) for the Sales Scenario is the same connector as proposed for the *Person2JavaClass* transformation in Section 6.3. In this section, we add additional information, where necessary to give a complete view on the Sales Scenario evaluation. In doing so, we follow the proposed steps from Section 6.3 regarding the two phases from the methodology, namely the *extraction* and *typing* phase.

Extraction phase: Per Definition 36, the extraction phase determines the trace graph structure per transformer through the instantiation of **Artefact** and **TraceLink** of the traceability metamodel. Thus, we need to implement a mapping from each transformer-specific execution to traceability data. The traceability graph is constructed by instantiating operators as attributes of the trace links (see Section 5.4 on alternative ways to modelling traceability graphs).

Following the methodology, the extraction is scenario-driven as well as transfomer-driven. For the purposes of this evaluation, we focus on all traceability scenarios mentioned in Section 8.1.1.

Regarding the transformer-driven extraction in terms of operators, we need to implement the connector for all Xpand statements relevant to traceability. Since Xpand is a model-to-text transformation, we need to consider the operator classification from Table 6.2. Since all operator classes are applicable to Xpand statements, the connector is implemented accordingly. Rules are degenerated in Xpand and therefore not traced. We refer to Section 6.3 for a detailed description on deriving traceability data out of Xpand statements and only provide a summary of selected transformations in Table 8.2. Recalling from Section 6.3, since the target is newly created in model-to-text transformations, a **CreateTraceLink** is used in all above cases, except for the query operator (cf. Table 8.2). Regarding model queries, Xpand allows the definition of *extensions* for user-defined queries on accessing information of the source model, which are modelled through a **RetrieveTraceLink**.

For a technical representation of the traceability data due to the above-mentioned operators regarding the feature (*Quotation Management* (in terms of the *View* DSL), we refer to Appendix D. All operators and trace links are represented including 81 operators distributed over 135 lines of transformation program.

Typing phase: The typing phase determines the types of the artefacts and trace links of the traceability graph on the basis of **Facets** and **Configuration**. The evaluation is based on the following facets:

- **NameFacet** for the name of a model element, if applicable,

Operator	**Xpand-statement**	**Source Artefact**	**Target Artefact**	**Link Type**
Static Text	TEXT	template snippet	text block	CREATE
Model Access	EXPRESSION	model element	text block	CREATE
Data Manipulation	EXPRESSION	model element(s)	text block	CREATE
Control Flow	FOREACH	model element	text block	CREATE
Module Control	FILE	model element	file/text block	CREATE
Query	.ext file[1]	model element	template snippet	RETRIEVE

Table 8.2.: Derivation of Traceability Data out of Xpand Statements

- **TextFileFacet** and **TextBlockFacet** (cf. Figure 5.3) for file or text-block information of a given artefact,
- **ArtefactTypeFacet** for the metamodel type of a given artefact and
- **TransformerFacet** for the name of a given rule or operator, for example, EXPRESSION and its identifier.

QVT Connector

The QVT connector is a blackbox connector based on the integrated traceability solution of the QVT engine. The evaluation is based on QVT Operational from the Eclipse project[2], which offers a dedicated traceability support by generating a so-called trace file. While executing a transformation, operational mappings are logged into the trace file containing the traceability data conforming to the QVT traceability metamodel.

Following the line of argumentation from Section 6.4, we need to implement a mapping transformation for transforming instances of the QVT traceability metamodel to instances of the *GTF traceability metamodel*. Regarding the mapping transformation, we go into more depth regarding the following points. First, we describe the QVT traceability metamodel and its mapping to the GTF traceability metamodel. Secondly, we explain the typing phase and the mapping to facets.

The QVT traceability metamodel contains the following structures (see Figure 8.1 showing a simplified version). The root class of the model is the **Trace**, which contains a collection of **TraceRecords**. A TraceRecord represents a trace link (in terms of the traceability model of our approach) containing four kinds of information: The **EMappingOperation** specifies the QVT operation, which is responsible for the trace link creation. In addition, an **EMappingContext**, **EMappingParameters** and **EMappingResults** are provided. The context contains information regarding the source artefacts of the transformation. Parameters passed to the QVT mapping specification are traced with the help of EMappingParameters. Finally, the resulting model elements are tracked by EMappingResults, which fundamentally correspond to target artefacts. These

---

[2] QVT Operational http://projects.eclipse.org/projects/modeling.mmt.qvt-oml

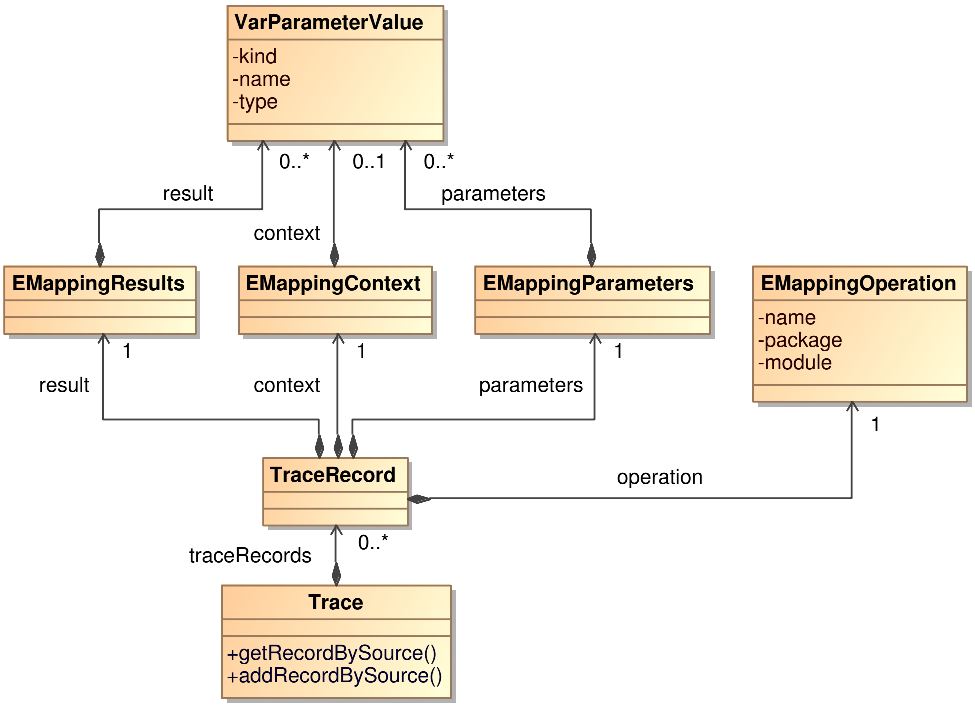


Figure 8.1.: QVT Traceability Metamodel

metamodel elements are correspondingly mapped to the GTF traceability metamodel in terms of the transformation program described in the following excerpt:

```
1  main() {
2   qvt.objectsOfType(QVTTraceDSL::Trace).traceRecords->map TraceRecord2TraceLink
       ();
3  }
4
5  mapping TraceRecord::TraceRecord2TraceLink() : CreateTraceLink {
6    init {
7      var facetsList := List {
8        self.map TraceRecord2TraceLinkNameFacet(),
9        self.map TraceRecord2TraceLinkMappingFacet()
10     }
11   }
12   sources := self.map TraceRecord2SourceArtefact()->asList();
13   targets := self.map TraceRecord2TargetArtefact()->asList();
14   facets := facetsList;
15 }
16
17 mapping TraceRecord::TraceRecord2SourceArtefact() : Artefact {
18   uuid := self._context._context.value.modelElement.map EObject2UUID();
19   facets := self.map TraceRecord2SourceArtefactTypeFacet()->asList();
20 }
21
22 mapping TraceRecord::TraceRecord2TargetArtefact() : Artefact {
23   uuid := self._result._result.value.modelElement->first().map EObject2UUID();
24   facets := self.map TraceRecord2TargetArtefactTypeFacet()->asList();
25 }
26
27 mapping TraceRecord::TraceRecord2SourceArtefactTypeFacet() : Facet {
28   id := "com.sap.gtf.facet.TypeFacet";
29   value := self._context._context.type;
30 }
```

Listing 8.1: **Trace File** to **GTF Traceability Model** Transformation

A TraceRecord maps to a trace link (Line 2). Since the QVTO trace file implies successful executions of model-elements mappings, each trace link corresponds to a **CreateTraceLink**. Hence, TraceRecords are converted into CreateTraceLinks (Line 5-15). Secondly, the source and target artefacts correspond to EMappingContext and EMappingResults, respectively and are transformed in terms of the mappings in Line 17 (for the source) and Line 22 (for the target).

The UUID[3] of source artefacts is drawn from the EMappingContext (Line 18), whereas the identifier for target artefacts is derived from the EMappingResults (Line 23).

---

[3] Since the QVTO implementation does not support a mechanism to get the fragment of an EObject's resource, we only derive the URI. Nevertheless, a possible solution could be to base the implementation on **EcoreUtil**.

Since the QVT traceability metamodel only accounts for traceability on the level of rules, the traceability of operators cannot be extracted and are thus not considered.

The QVT connector's typing phase is based on the following facets, which are stored in the respective artefacts and trace links:

- **TraceLinkNameFacet** to trace the name of a given transformation rule (Line 8),
- **TraceLinkMappingFacet** to trace the transformation rule (Line 9) and
- **ArtefactTypeFacet** to trace the name of the metamodel type of source and target artefacts (Line 19, 24).

In particular, trace links are enhanced with information such as the name of the responsible mapping (TraceLinkNameFacet) and the EMappingOperation (mapping) itself (TraceLinkMappingFacet). Besides that, artefacts store an object of the ArtefactTypeFacet containing the name of the artefact's metamodel type.

ATL Connector

The ATL connector is a blackbox connector and is built in analogy to the QVT connector. The traceability data in the standard ATL virtual machine (EMFVM) is not stored in a model, but is internally represented as classes of type TransientLinkSet and TransientLink as described in the specification of the ATL vitual machine [Foub]. For obtaining traceability data as a model, there are several options:

1. By applying the higher order transformation (HOT) *ATL2Tracer* [Jou05] to render a modified version of the transformation that produces a traceability model conforming to a simple traceability metamodel as depicted in Figure 8.2,
2. By applying the HOT *ATL2WTracer*[4] as in the case above to render a modified version of the transformation that produces a traceability model conforming to a weaving metamodel,
3. By modifying the ATL Virtual Machine to serialize the traceability information conforming to the same metamodel as option 2, or use the experimental ATL virtual machine EMFTVM[5] that uses a metamodel evolved from option 2.

We follow the first option based on the HOT, *ATL2Tracer* in order to gain the advantages of an already integrated traceability solution requiring no additional effort. The used traceability metamodel is depicted in Figure 8.2. The first option uses a subset of the traceability metamodel used in the second option, yet the latter is applied for model weaving, which has not been regarded in this work. To explain the phases of the ATL connector, we first describe the mapping of the ATL traceability metamodel[6] to the

---

[4] http://www.eclipse.org/gmt/amw/examples/#ATL2WTracer

[5] http://wiki.eclipse.org/ATL/EMFTVM#Advanced_tracing

[6] http://www.eclipse.org/atl/atlTransformations/#ATL2Tracer.

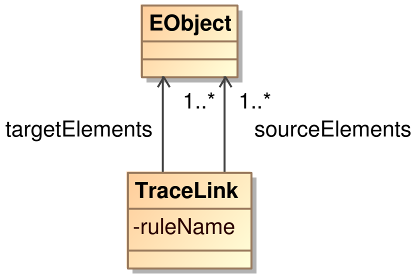


Figure 8.2.: ATL Traceability Metamodel

GTF traceability model, secondly the mapping to the used facets. The ATL traceability metamodel is mapped to the following corresponding GTF metamodel elements:

- The ATL **TraceLink** corresponds to the GTF **TraceLink**.
- The ATL **EObject** corresponds to the GTF **Artefact**: Source and target elements—instances of **EObject**—are mapped to source and target artefacts—instances of **Artefact**.
- The ATL **ruleName** corresponds to the rule name captured as a facet (cf. *typing phase* below) of the GTF CRUD-**TraceLinks**.
- Regarding the **UUID** of artefacts, ATL generates identifiers based on XMI upon serialization of the traceability model.

The ATL connector's *typing phase* is based on the following facets:

- **TraceLinkNameFacet** to trace the name of a given rule and
- **ArtefactTypeFacet** to trace the name of the metamodel type of source and target artefacts.

Since the ATL traceability metamodel only applies to traceability on the level of rules, the traceability of operators cannot be extracted from the generated traceability model of *ATL2Tracer*. Thus, the conditions of typing apply to rules only. With the above ATL traceability solution, only the rule name is traced. For enhanced information, such as a rule identifier, for example, the offset, additional support from the ATL VM is needed, which is out of scope of this work. Thus, a facet as in the case of QVT with the **TraceLinkMappingFacet** is omitted. In summary, trace links are enhanced with the following information: The ATL **ruleName**, being an attribute of **TraceLink**, is gathered through the TraceLinkNameFacet. Besides that, artefacts are linked with an ArtefactTypeFacet which contains the name of the artefact's metamodel type.

More information on the 41 ATL transformations is followed up on in Section 8.2.4 regarding the matching approach. Essentially, the ATL connector is used to collect traceability data, which is used as reference data for the mappings extracted through the matching approach.

8.1.4. Evaluation Results

In the following section, we present the evaluation result for a representative of each generation class, that is, for the Xpand and QVT connector. The results concern the gold-mapping coverage and traceability-graph expressiveness. While the former refers to the achieved quality of trace links in terms of covering the gold mapping, the latter requires an analysis on the expressiveness of the resulting traceability graphs.

Both connector-based approaches were implemented to achieve a complete coverage of the gold mapping (cf. Section 8.1.2). Regarding the data-manipulation operators of the Sales Scenario, the evaluation included simple expressions of string concatenation. The traceability of complex manipulations needs to be further investigated. An analysis of the benefits in relation to the granularity level of traceability is necessary.

Regarding the expressiveness of traceability graphs, the outcome is summarised in the subsequent two sections for the invasive and blackbox connector.

Invasive Connectors

Recalling that traceability scenarios are views on a traceability graph, we show to which extent the connector delivers a sufficiently expressive graph with respect to each traceability scenario.

System Comprehension: Since all operators in the transformation program are traced, the traceability graph's expressiveness provides a complete view on the transformation mapping. The user may navigate along trace links between source and target artefacts of the transformation to understand their logical and functional dependencies. Additionally, the CRUD trace links capture the kind of transformation as classified in Section 5.2 and related operations on model elements, e.g., several elements were created and updated.

Coverage Analysis: To determine, whether all requirements were covered by test cases in a given MDSD process, the traceability of models and model elements is coupled with the task of collecting traceability data for an end-to-end development process. Since the runtime footprint of all operators is traced as per connector design, all transformed source models and elements are traced. Thus, it is possible to follow all outgoing links from a set of design models (e.g., from the feature Quotation Management) to all generated target classes. This expressiveness can be used for a coverage analysis on an end-to-end traceability graph, for example, to decide, whether a certain requirement related to the feature Quotation Management and all of its generated classes are followed up by a set of test cases, which are run for the coverage analysis.

For a coverage analysis, it might be convenient to vary the granularity level of traceability, for example, regarding the generated classes, methods and/or files can be linked to test cases referring to unit and integration tests. Depending on the more suitable view, the granularity of the traceability data needs to be configured, for example, the granularity level is set to file level only. Since it is possible to configure traceability data though the settings of granularity and scope (cf. Section 5.3), a coverage analysis can be conducted on the configured data. This applies to the other scenarios the same way.

Change Impact Analysis: As mentioned in the motivating example a change impact analysis determines how changing one artefact would affect other artefacts. For an end-to-end traceability scenario, the same conditions as for a coverage analysis apply to the traceability graph's expressiveness. In addition, explicit dependencies on model elements need to be considered, e.g., bestowed through requirement dependencies. The derivation of these links is beyond the scope of this work, however, they can be incorporated into our way of modelling in storing them as facet objects in the corresponding CRUD trace links.

Recalling the modelling adaption due to hyperedges and the iterative execution of operators (see p. 88), we capture the additionally needed mappings (from source to target artefacts) in the trace links. Therefore, a change impact analysis can be conducted on individual source-target relationships directly related to a specific operator as explained on p. 88.

Orphan Analysis: An orphan analysis is used to find so-called orphaned artefacts (that are not referenced by, or reference other artefacts), e.g., while deleting a model element, the corresponding code might be outdated. The traceability graph is used to find such orphaned elements by searching for dangling edges, e.g., the deleted model element leads to a dangling edge pointing to outdated code in the generated target. Since our approach does not propose to model dangling edges, their occurrence as described above can be used for identifying orphans exclusively. To update the outdated information due to occurring orphans, the transformation may be re-run in order to synchronise models, code and traceability data.

Transformation Debugging: For debugging model transformations, information about the processing stack must be collected during the execution of the generation process, that is, the implementation of the transformation engine needs to account for this. The set of operators must be recorded in the order of execution by the transformation engine. Nested operators of modularization and control flow operators are stored in this context. This leads to a tree structure of recorded operators.

The traceability graph generated from the Xpand connector is sufficiently expressive for debugging a transformation program, since the output index is traced for each operator as well as its used model input. Let us assume the following debugging scenario as an example. The static text *Person* (Line 1) in the generated code from the running example in Section 6.3.1, requires the processing stack as depicted in Table 8.3. In this example, the index is defined through the line numbers. Given the processing stack of operators, needed to generate a specific code block (e.g., *Person*), it is possible to debug operator by operator up to this specific code block (row by row in Table 8.3). The traceability data of each operator is tracked for its used model element(s) and all code that is generated directly and indirectly by the operator. Indirect generation occurs,

Operator	**Output Index**
«DEFINE javaClass FOR Entity »	1-33
«FILE name +". java"»	1-33
«name »	1

Table 8.3.: Processing Stack

since there may be more operators involved for the generation of a code block, e.g., DEFINE and FILE from Table 8.3 link to the same generated code.

Thus, if a problem was discovered in a generated file, a developer can utilize this traceability data to identify template expressions and model data that caused the generation of the defective part. The defective part can be identified up to a granularity level of static text. Therefore, if a developer uses erroneous static text in the transformation program, which leads to errors in the programming code, it is easily possible to locate this static text.

Blackbox Connectors

The blackbox connector's graph expressiveness is dictated (and potentially restricted) by the nature of the implicit traceability metamodel (of the implicit transformation approach). For instance, this is reflected in the Scope and Granularity of the GTF traceability metamodel. Given a model-to-text transformation approach, if the traceability solution's granularity level is restricted to file level, the traceability data on the level of code blocks in the generated files is unavailable for analysis (cf. Section 6.4.2).

In applying the mapping transformation of the blackbox connector (implicit traceability metamodel is mapped to the GTF traceability metamodel), the expressiveness of the original traceability data is preserved for the target. If this expressiveness was sufficient for conducting the traceability scenarios successfully, before the mapping execution, the same applies after, permitted no errors occur during the mapping. Alternatively, it is advisable to build an invasive connector to have control over the data expressiveness as shown above. The focus of this work is to evaluate traceability graphs due to invasive connectors and not due to implicit traceability metamodels. Hence, we refer to [Kas09], for a description on the conduction of the above-mentioned traceability scenarios for the *QVTO UML2RDB* transformation.

8.1.5. Summary and Analysis

We have demonstrated that the invasive connector has control over the modelling of traceability data, defined through facets, scope and granularity, and its generated traceability-graph expressiveness is user-specific. By following the connector design presented for Xpand, we have shown, how the traceability graph can be used to conduct the traceability scenarios *System Comprehension*, *Coverage Analysis*, *Change Impact Analysis*, *Orphan Analysis* and *Transformation Debugging*.

On the contrary, the blackbox connector has no control over the extraction of traceability data, yet allows for the typing of the resulting graph from the extraction phase. Thus, the resulting traceability graph's expressiveness is dictated (and potentially restricted) by the nature of the implicit traceability metamodel. Consequently, the resulting expressiveness is as expressive as the default solution's traceability graph.

8.2. Model Matching for Traceability

In the following section, we evaluate our approach regarding the second and third category, that is, our model matching-based approach to traceability data extraction.

The goal of this evaluation is to measure the quality of our matching-system and hence, the generated trace links. In doing so, we need to find representative evaluation scenarios and measures to express the matching quality as described in Section 8.2.2. The chosen evaluation scenarios, the Sales Scenario and ATL Zoo are introduced in Section 8.2.3 and 8.2.4, respectively. We present the evaluation results in Section 8.2.5 followed by a summary in Section 8.2.6. Finally, Section 8.2.7 provides a brief overview on the scalability of our matching system.

8.2.1. Evaluation Strategy

As argued in Section 7.2, we base our work on the Matchbox framework as described in Appendix B. In order to measure the quality of our matching system, we need to evaluate the three similarity measures proposed in Chapter 7. Since these similarity measures are realised through 8 matching algorithms integrated into Matchbox, the evaluation is based on finding optimal configurations for a similarity value cube to generate trace links.

In finding optimal configurations, the evaluation relies on the brute force method [LLC10], entailing the variation of all parameters with respect to their values to gain all possible configurations. These configurations are then evaluated with respect to their quality of matching results. One possibility is to choose the configuration with the best quality for each matching scenario (mapping), however on average, this configuration may not deliver the best quality. Therefore, a more realistic estimate would be to calculate the average quality of all mappings. In particular, to estimate the quality of our matching system with respect to the chosen evaluation scenarios, we answer the following questions:

1. What is the average quality of our matching results?
2. What is the best quality per mapping?

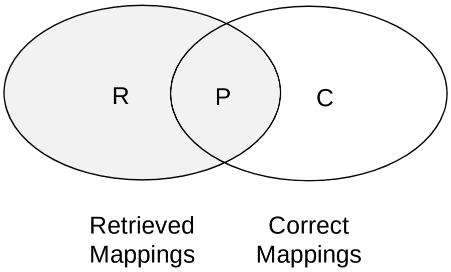


Figure 8.3.: Retrieved Mappings and Correct Mappings represented as Sets

3. What is the influence of metamodel-driven matching?

8.2.2. Evaluation Setup

Regarding the evaluation setup, we provide a detailed description of the varying parameters and quality measures used. Furthermore, we present the evaluation scenarios.

Evaluation Measures

For measuring the quality of the resulting mapping, we introduce classical measures known from information retrieval. This involves the measures, *precision*, *recall* and *f-measure* [Rij79]. Formally, we define these measures by considering two sets of mappings, the set of retrieved mappings (actual set) and the set of correct mappings (intended set).

Definition 38 (Precision, Recall, F-measure)**.** *Let R be the number of retrieved mappings, P the number of correct (positive) mappings and C be the total number of correct mappings (see Figure 8.3), then the following measures can be defined:*

- ***Precision**: $prec = P/R$*
- ***Recall**: $rec = P/C$*
- ***F-Measure**: $F\text{-}measure = 2 \cdot \frac{prec \cdot rec}{prec + rec}$*

Precision expresses the number of correct mappings found relative to the number of *found mappings*, whereas recall describes the number of correct mappings relative to the number of *total mappings*. The f-measure is a weighted harmonic mean and requires recall and precision to be balanced in order to achieve a high value. All measures take on values (real numbers) between 0 and 1, where a higher value is a better result. Based on the above definition, one can interpret precision as a measure for the correctness of the retrieved mappings. Analogously, recall is a measure for the completeness of the retrieved mappings. For example, if the precision amounts to 0.9, this means that 90% of the matches are correct.

In order to state the above-defined measures for a given matching scenario, a reference mapping is needed, which is a complete and correct set of mappings. We call this the gold-standard mapping or abbreviated, *gold mapping*.

Evaluation Parameters

The evaluation is setup to use four kinds of parameters, namely *matcher*, *metamodel-driven matching*, *aggregation*, and *selection*. We sketch the possible values of each successively.

The *matcher* parameter has a range of 7 matchers, implemented in Matchbox (cf. Section 7.2.2). Therefore, all 127 combinations of matchers are evaluated (equal to the power set of matchers excluding the empty set).

Secondly, the evaluation comprises configurations with respect to variations of metamodel-driven matching (cf. Section 7.1.5), namely *blocking*, *instance-of* (see p. 120), and *filtering*. Additionally, configurations without metamodel-driven matching are considered. Thereby, the matching process at meta-level is based on the gold-standard mapping, and therefore an additional metamodel-matching process is avoided.

Thirdly, aggregation and selection strategies are applied (cf. Appendix B.3). The aggregation strategies include *Max* and *Average*. Finally, mappings are identified by the selection strategy, *Threshold*, *selDelta*, and *selN*. In this way, the following variations are considered. The *Threshold* values lie between 0.1 and 1.0 ascending in steps of 0.1. Accordingly, the considered *selDelta* starts at 0.0 and ends at 0.15 in steps of 0.05. In order to include configurations with a disabled selDelta, additionally, *selDelta* is set to 1.0. Furthermore, the *selN* is set to infinity, 1, 3 and 5. The chosen parameter values are derived from experiments prior to the evaluation as well as best practises of [Do06, VIR10].

To summarize, 203200 (= 127 matcher combinations $\times 4$ metamodel-driven matching techniques $\times 2$ aggregation strategies $\times 10$ selection thresholds $\times 5$ selDelta values $\times 4$ selN values) parameter configurations per source-target combination are evaluated.

Evaluation Scenarios

In the following, we describe the used scenarios or use cases to demonstrate the evaluation and motivate their choice. Our evaluation is based on 41 model transformations from the ATL Zoo [Fouc] as well as a SAP business application, called Sales Scenario. For a description of the latter, we refer to Appendix C. The motivation for these scenarios stems from our requirements regarding the mapping tasks:

1. varying level of abstraction between source and target models,
2. broad language scope and
3. broad application domain.

While the ATL Zoo covers model-to-model transformations, the Sales Scenario covers Xpand model-to-text transformations. Since models are more abstract than text, both scenarios have a different level of abstraction, thus fulfilling the first requirement. Both mapping scenarios complement each other with respect to the language scope and applicability. Regarding the Sales Scenario domain-specific models are transformed to Java source code in order to generate a complete business application with large-scale transformations. On the other hand, the ATL Zoo comprises common languages like UML, XML, and KM3 as well as domain-specific ones with transformations ranging from technical-space bridges and refactorings to model refinements.

8.2.3. Sales Scenario

In this section, we characterise the Sales Scenario with respect to the size of its models. Furthermore, we show the retrieval of the gold-standard mappings that needs to be automated in case large model repositories are being used.

As described in Appendix C, the Sales Scenario entails a range of model-to-text transformations. The size of source models varies between $72 - 1231$ model elements, while the target models contain $95 - 2593$ model elements. The considered source models are instances of six domain specific languages (DSLs) describing different domains of the business application:

- **Action DSL** to declare invokable behaviour
- **Business Object DSL** for data structure modelling
- **Context DSL** for session contexts to buffer in-memory data models in addressable spaces
- **State DSL** for transitions based on state machines
- **Dialog DSL** for page flow definitions
- **View DSL** for graphical user interfaces

Regarding the specification of application features, such as account management, quotation management, order management and product management, domain-specific models per DSL are defined for each application feature. Given this repetitive nature for each application feature, similar domain-specific models with respect to structure and values are created. Thus, the same transformations are performed on similar domain-specific models. Therefore, the evaluation scope is limited to one application feature, namely, quotation management, without a loss of generality.

Furthermore, the models of the Context and View DSL are characterised by large source and target models and therefore require long matching times. With respect to 203200 configurations per source-target model combination, according transformations are omitted in the evaluation due to runtime concerns. In summary, the model repository of the

DSL	Source models	Target models	Mappings
Action	2	12	7
BusinessObject	1	14	3
Context	1	38	-
Dialog	1	1	1
State	2	2	1
View	6	81	-
Total	13	148	12

Table 8.4.: Sales Scenario Models

Sales Scenario consists of 12 model combinations (see Table 8.4), that conform to Action, BusinessObject, Dialog and State DSL.

Evaluation Setup

In order to evaluate the Sales Scenario a gold mapping is needed to be able to measure the quality of the matching results, that is, to calculate the above-mentioned quality measures. Therefore, the evaluation setup needs to account for the gold mapping as well as the mapping created by our matching system. In Figure 8.4, we sketch the evaluation setup. To extract this gold mapping, the GTI is utilized with an invasive connector for Xpand, as depicted on the left side. Firstly, the transformation is considered as model element mapping that explicitly captures matches between model elements. Secondly, the target being Java source code is transformed into a java model. For the latter, the Java Model Parser and Printer (JaMoPP[7]) is used for parsing Java source code into an EMF-based model. Therefore, the connector is instructed to capture this mapping of source and target model elements. The used connector does not follow the fine-grained approach as presented in Section 8.1.3, yet is instrumented to only trace respective source and target model elements that result from the mapping transformation, for example the mapping in Figure 7.5 (lower level). In other words, the captured source and target model elements need to conform to a corresponding metamodel element.

Thus, an automatic approach is used in terms of model transformations as opposed to a manual process, which is more time-consuming and likely error-prone. In doing so, we consider only source and target model combinations that correspond to each other. This constraint significantly eases the task of matching and leads to better matching results.

On the other hand, the traceability matching system delivers the mapping results (right side). Afterwards, the assessment of matching quality works on the mapping and gold mapping (lower part) both populated to a traceability repository. For this evaluation

---

[7] JaMoPP http://www.jamopp.org/index.php/JaMoPP

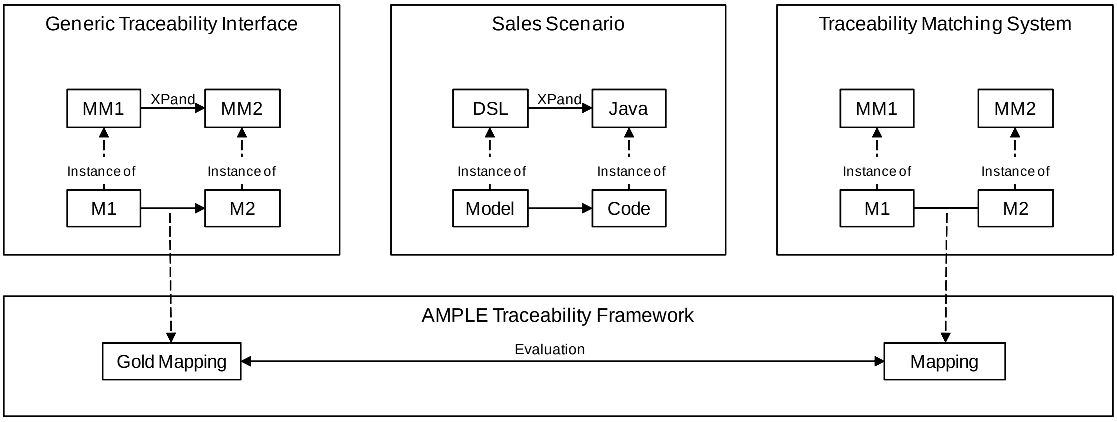


Figure 8.4.: Evaluation Setup for the Sales Scenario

the repository of the AMPLE Traceability Framework (ATF) was used (see p. 157) for reusing repository-specific functionalities and therefore integrated into the GTF. Thus, two sources of traceability data exist in the ATF and are compared against each other, the gold mapping and mapping.

8.2.4. ATL Zoo

This section introduces the ATL Zoo [Fouc] used as evaluation scenario. The ATL Zoo consists of a collection of 103 model transformations. The selected transformations for evaluation are based on the following criteria:

- EMF-compatibility,
- existing instances and
- feasible evaluation effort.

Since the implementation of the matching system for traceability builds up on Matchbox, the inherited technical space of EMF restricts the evaluation of transformations based on Netbeans (e.g., UML-to-Java). Furthermore, transformations without examples of model instances are omitted. Moreover, the source model from the UML2-to-Measure transformation comprises more than 30 thousand model elements and therefore can not be matched within the evaluation setup and scalability properties. All in all, 41 examples from the ATL Zoo are selected. The size of source models ranges from 41 – 3253 model elements, while the target models consist of 14 – 1813 model elements.

Evaluation Setup

Analogously to the Sales Scenario setup, two sources of traceability data are needed: the gold mapping and the mapping due to the model matching system. Both are compared

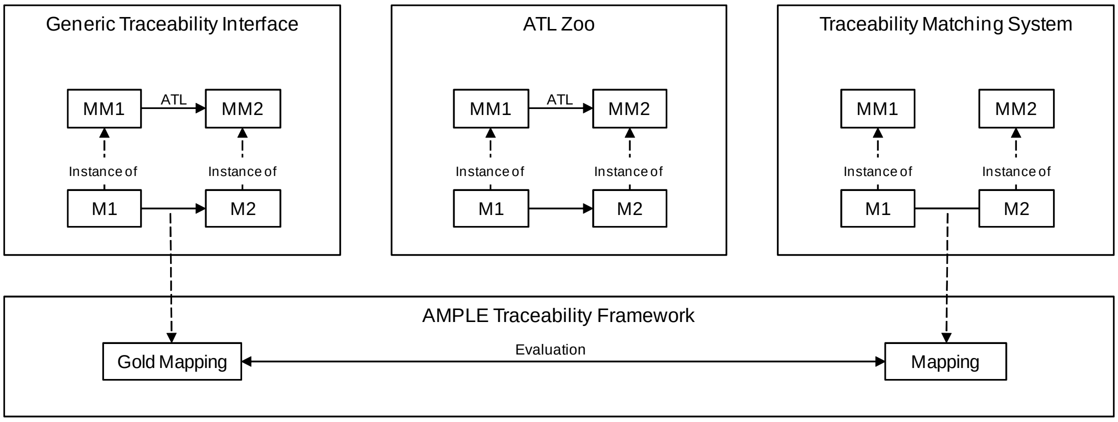


Figure 8.5.: Evaluation Setup for the ATL Zoo

and serve as foundation for the evaluation of quality. Throughout the evaluation of the ATL Zoo, we only consider source and target model combinations that correspond to each other. The evaluation setup is depicted in Figure 8.5.

In order to extract the gold mapping, we make use of an automatic approach as opposed to a manual extraction to minimize effort and potential errors. Thereby, we instantiate the ATL traceability support as presented in [Jou05], using a high-order transformation to incorporate traceability-specific transformation rules into the transformation program. As a result, the ATL transformation produces a traceability model next to its regular output. A blackbox connector of the GTI populates this traceability model to the traceability repository of the ATF as depicted on the left side. The connector corresponds to the ATL connector from Section 8.1.3. Consequently, the ATF contains the gold mapping.

Again, the traceability matching system creates the mapping results (right side) and subsequently populates traceability data into the ATF, where upon gold mapping and mapping are compared to each other and serve as foundation for the evaluation of quality.

8.2.5. Evaluation Results

Since the goal of the matching system is to generate traceability data, we are interested in finding optimal configurations that provide a high quality of matching results. To ease this task for the traceability user, we aim at finding configurations that have a broad applicability for matching tasks and achieve a high quality. The traceability user can then apply these configurations per default. For this investigation, we turn to the questions raised in the beginning of Section 8.2.1:

1. What is the average quality of our matching results?

2. What is the best quality per mapping?
3. What is the influence of metamodel-driven matching?

These questions provide an outline for the next three sections:

Default Configuration

Regarding the first question, the average f-measure per configuration over the Sales Scenario and ATL Zoo mappings is examined. In other words, all configurations are applied to each source-target combination (mapping). Then, the f-measures of all mappings with the same configuration are considered and their average is calculated. This is repeated for all configurations. Based on these resulting *average* f-measures, the maximum average f-measure is identified. Finally, configurations assigned to this maximum average f-measure are selected. These configurations are called default configurations and provide a measure for the average quality of the matching results.

It turns out, that the Sales Scenario and ATL Zoo are characterised by significantly different default configurations. Regarding the matcher combination, the results are particularly striking. While the *Attribute Matcher* is most successful for the Sales Scenario, the *Graph Edit Distance Matcher* best accounts for the ATL Zoo. Furthermore, different selection strategies achieve the best average results in both scenarios as shown in Table 8.5. Both default configurations are independent of the aggregation strategy, i.e., either *Average* or *Max* can be chosen, which is a common observation according to [Do06].

	Sales Scenario	**ATL Zoo**
Matcher	Attribute Matcher	GraphEditDistanceMatcher
SelN	-1	-1, 1, 3, 5
SelDelta	0, 1	0, 0.05, 0.1, 0.15, 1
Threshold	0.5	0.7
Number of configurations	8	40

Table 8.5.: Default configurations for Sales Scenario and ATL Zoo

The resulting matching quality by using the default configurations for the Sales Scenario and ATL Zoo is depicted in Figure 8.6 and Figure 8.7, respectively. Both figures illustrate the maxima of recall, precision, and f-measure against the according mappings. Thereby, the vertical axis indicates the value of recall, precision and f-measure, while the combinations of source and target models are plotted on the horizontal axis. Regarding the Sales Scenario, the mappings are named after corresponding DSLs (cf. Table 8.4), while the ATL Zoo naming is derived from the respective transformation names.

Regarding the Sales Scenario, the average of recall ranges from $0.16 - 1$ (column 4 and 3) and the average of precision from $0.05 - 0.48$ (column 10 and 4). In accordance, the average f-measure lies between the value of $0.1 - 0.4$ (column 10 and 5). BusinessObject (column $1 - 3$) and Action (column $6 - 12$) DSL achieve high recall values as opposed to Dialog (column 4) and State (column 5), that in return feature a higher precision. In total, the default configuration of the Sales Scenario achieves an average recall of 0.5, an average precision of 0.17 and a resulting average f-measure of 0.2.

Regarding the ATL Zoo, the average of recall and precision ranges between $0.03 - 1$ and $0.33 - 1$, respectively. In accordance, the average f-measure lies between 0.05 (Ant-to-Maven3) and 1 (Book-to-Publication). Also, all source-target combinations have a higher precision than recall (except for Book-to-Publication). This means that the degree of correctness of a mapping is greater than its completeness. This implies that default configurations lead to a more pessimistic approach in rendering mappings with correct matches rather than incorrect ones.

Further, the default configuration did not find any correspondences for Families-to-Persons, Software Quality Control-to-Bugzilla, Software Quality Control-to-Mantis Bug Tracker, and Syntax-to-SBVR-to-UML1. This points out that finding a single configuration for all source-target combinations, which delivers matching results of sufficient quality, is not realistic. Therefore, mapping-specific configurations need to be investigated as described in the next section. In total, the default configuration of the ATL Zoo achieves an average recall of 0.14, an average precision of 0.72 and a resulting average f-measure of 0.21.

Profiles

In answer to the second question concerning the best quality per mapping, we investigate how mapping-specific configurations achieve a higher quality in matching results. We call these configurations profiles. Regarding the best quality per mapping, the configurations (profiles) with the highest f-measure for each mapping are considered. The recall, precision and f-measure due to these configurations are depicted in Figure 8.8 for the Sales Scenario and in Figure 8.9 for the ATL Zoo. Again, the vertical axis indicates the value of recall, precision and f-measure and accordingly ranges from 0 to 1. In addition, combinations of source and target models are plotted on the horizontal axis. In general, the average f-measure of profiles increases by half in comparison to the default configuration. In particular, the average f-measure rises from 0.20 to 0.34 for the Sales Scenario and from 0.21 to 0.31 for the ATL Zoo.

Metamodel-driven Matching

Recalling from Section 7.1.5, metamodel-driven matching has an impact on the quality of matching results as well as the scalability of the matching process. Since the focus of this work lies on the former, the delta of applying metamodel-driven matching to

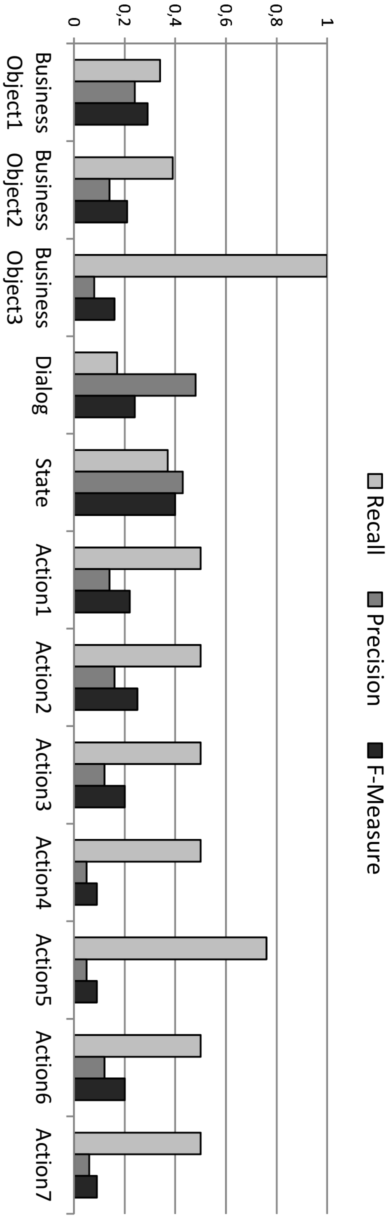


Figure 8.6.: Average Results of Sales-Scenario Default Configuration against Mappings

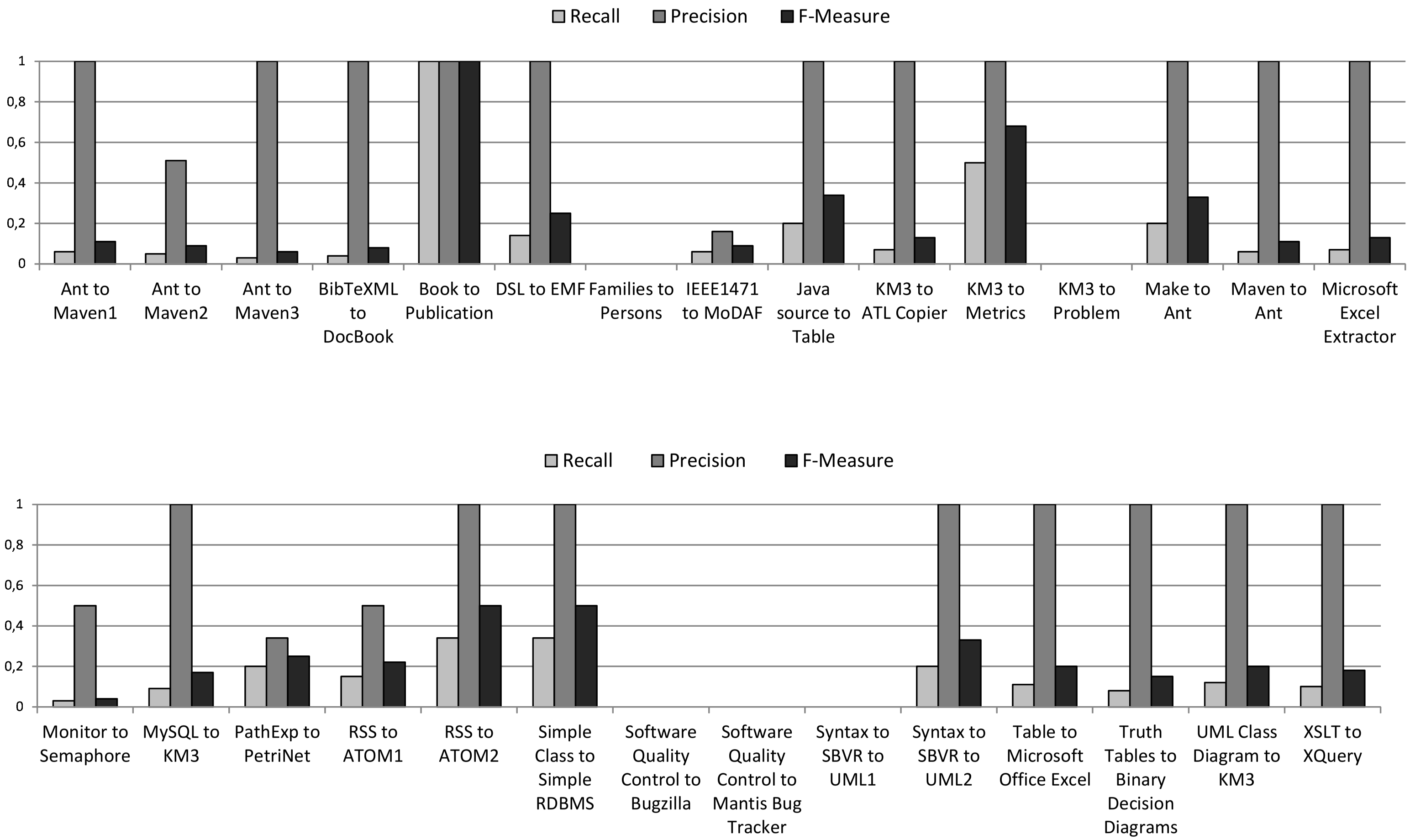


Figure 8.7.: Average Results of ATL-Zoo Default Configuration against Mappings

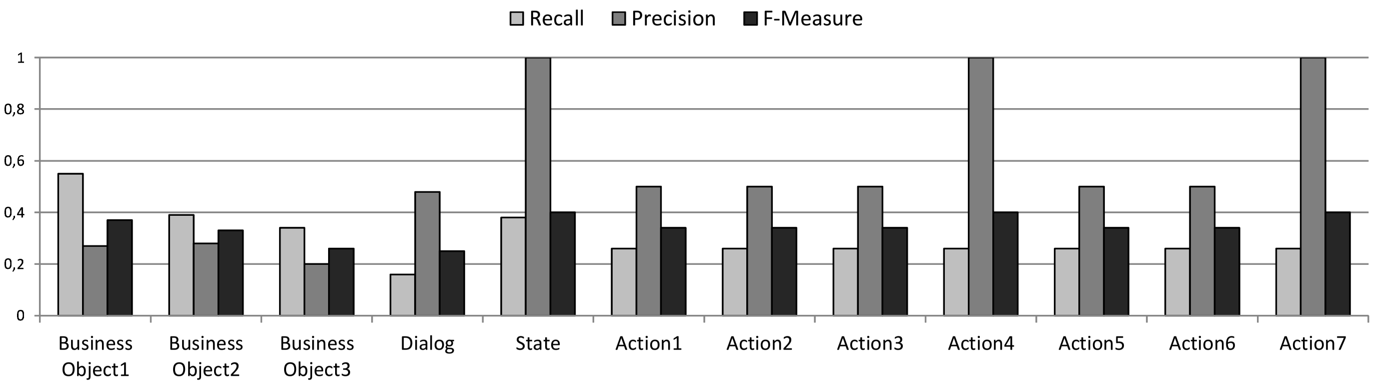


Figure 8.8.: Maximum Results of Sales-Scenario Profiles against Mappings

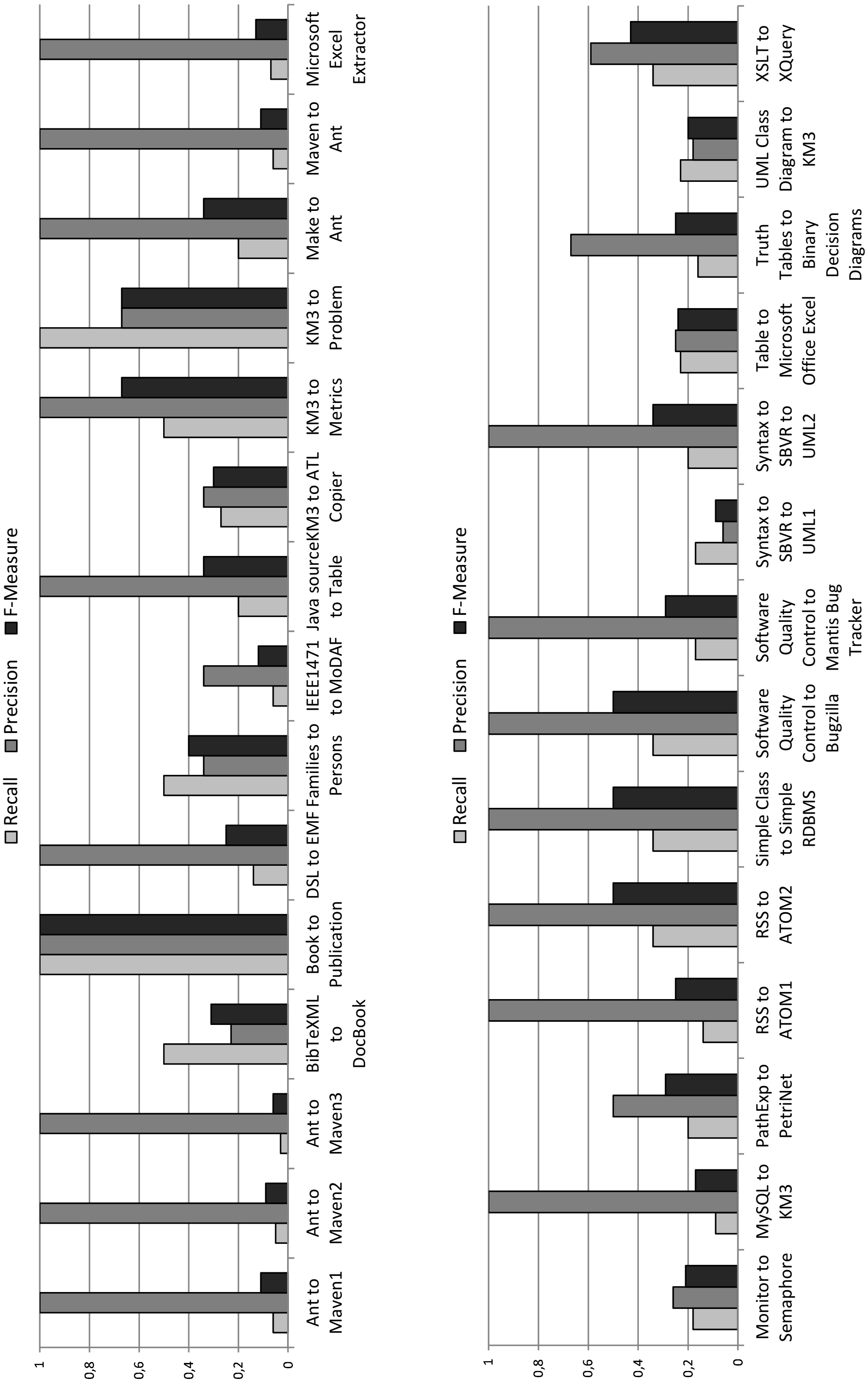


Figure 8.9.: Maximum Results of ATL-Zoo Profiles against Mappings

non-metamodel-driven matching is examined. The delta is calculated with respect to the results of the default configurations. Thus, the maximum recall, precision and f-measure between non-metamodel-driven matching and *blocking*, *filtering* and *instance-of* matching are discussed. The differences in terms of quality measures are depicted in Figure 8.10 for the Sales Scenario and for the ATL Zoo in Figure 8.11 and Figure 8.12. In all diagrams, the vertical axis indicates the delta of maximum recall, precision and f-measure and accordingly ranges from -1 to 1, while the combinations of source and target models are plotted on the horizontal axis.

Regarding the Sales Scenarios, Figure 8.10 is divided into an upper and lower part, where the upper part illustrates the delta of recall, precision and f-measure against according mappings in comparison to *blocking* and *filtering*. In contrast, the lower chart illustrates the delta of recall, precision and f-measure against the same mappings in comparison to *instance-of* matching. Blocking and filtering produce the same evaluation results, since both techniques essentially remove matches having unmatched types. Hence, both are plotted into a single figure. The same separation applies to the ATL Zoo, except that the results have been split into two different figures, that is, *blocking* is presented in Figure 8.11 and *instance-of* matching in Figure 8.12.

Regarding blocking and filtering for the Sales Scenario, the maximum values of recall, precision and f-measure are improved or remain the same for all considered mappings. In contrast, only 4 out of 29 models of the ATL Zoo show an increase in maximum precision, yet most mappings already achieved a maximal precision of 1 (cf. Figure 8.7) without blocking techniques. Nevertheless, for 21 mappings of the ATL Zoo, the maximum recall is raised.

As a result, when metamodel-driven blocking is performed on the *default configuration* mappings, the average f-measure increases from 0.20 to 0.26 for the Sales Scenario and from 0.21 to 0.48 for the ATL Zoo.

Moreover, by applying blocking on *profile configurations*, the average f-measure increases from 0.34 to 0.85 for the Sales Scenario and from 0.31 to 0.90 for the ATL Zoo. Hence, blocking is more effective for models of the ATL Zoo than for the Sales Scenario. The reason for this is the lower number of instances per metamodel element regarding the ATL Zoo. Recalling from Section 7.1.5, blocking decreases the number of Cartesian pairs considered in the matching process. Regarding more instances, this number is increased and may lead to the calculation of false positives (found, yet incorrect matches).

In applying instance-of matching to the Sales Scenario, the f-measure decreases (5 mappings), or remains the same (7 mappings). Regarding the ATL Zoo only three mappings show an increase in f-measure, while the other mappings illustrate an unchanged (16 mappings), or decreased f-measure (10 mappings).

In summary, the maximum f-measure is lower, than without metamodel-driven matching for both evaluation scenarios. An exception is the Families-to-Persons mapping of the ATL Zoo, which may be substantiated by the small model sizes. In particular, the average recall and precision of the Sales Scenario mappings decreases minimally. This

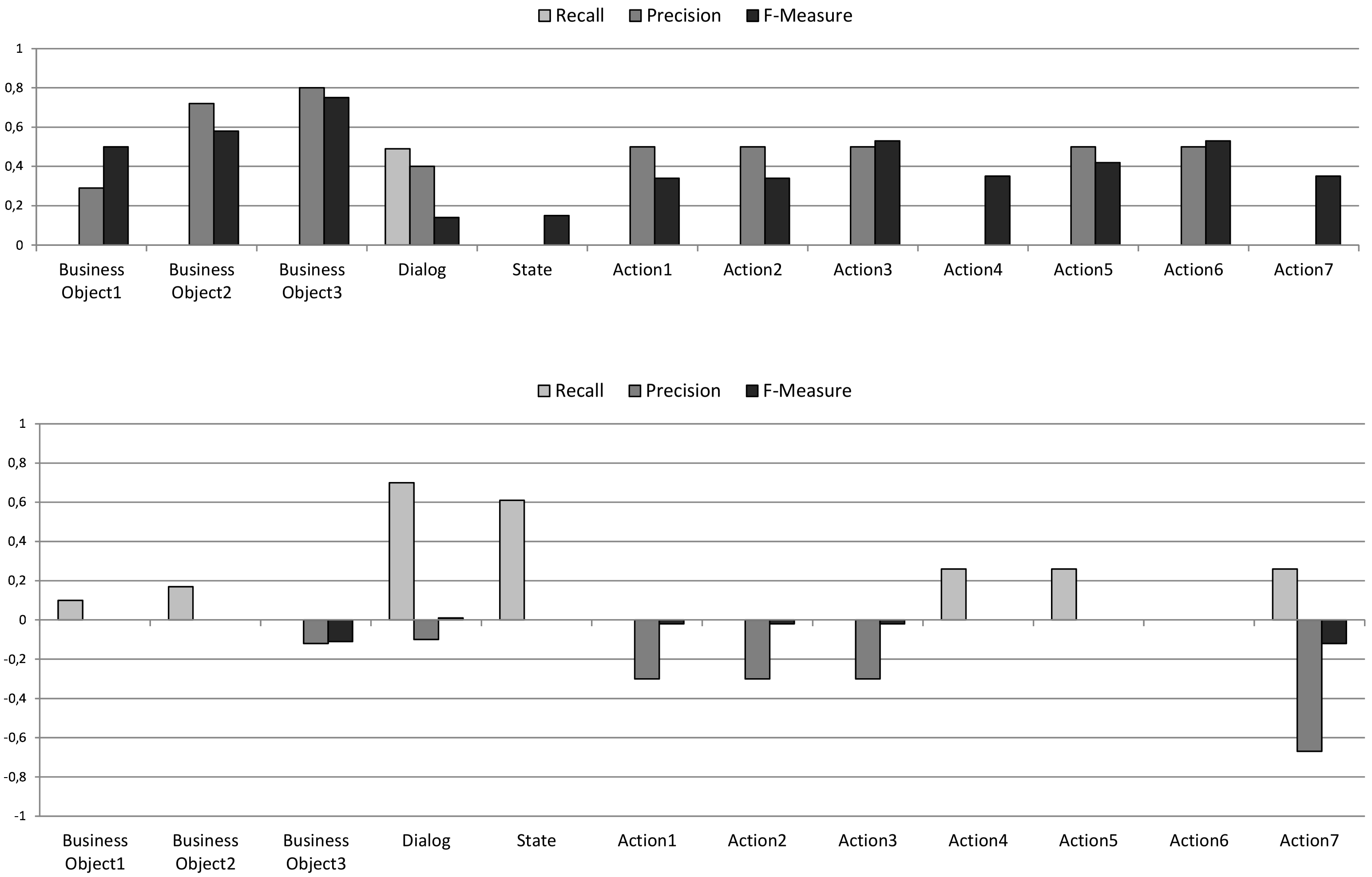


Figure 8.10.: Differences for Blocking and Instance-of applied to the Sales Scenario

results in a decrease of the average f-measure from 0.21 to 0.07. In contrast, regarding mappings of the ATL Zoo, the average recall remains almost the same, while the average precision decreases. As a consequence, the average f-measure decreases from 0.21 to 0.08.

To summarise, the application of instance-of matching decreases the matching quality for our considered evaluation scenarios. For an explanation, we recall the instance-of matcher from the running example on page 112 with example results from Table 7.4. By introducing matches to model elements that conform to matching metamodel elements, false positives are introduced, for example, ***feature1*** to ***field2*** and ***feature1*** to ***method2***. If these matches are not filtered out through the configuration of the SVC, they reduce the matching quality. With a higher number of instances per metamodel element, false positives are likely to increase and can decrease the matching quality.

8.2.6. Summary and Analysis

To conclude the evaluation of the quality of our matching system, we provide a summary and analysis of our results.

Summary

The summary of our results are presented in Table 8.6. Firstly, we presented default configurations, which stand for a more pessimistic assessment of quality as the profile configurations. The latter configurations are derived from the maximum f-measure of each mapping. Finally, we evaluated metamodel-driven matching, that is, *blocking*, *filtering* and *instance-of* matching.

The average results pertaining to the default configurations are listed under default. In general, the average of the highest f-measure for each mapping (called profile in Table 8.6) increases by a factor of 1.6 in comparison to the average results of default configurations. Applying metamodel-driven *blocking* with a default configuration (denoted with Blocking^D) improves matching results by a factor of 1.8. Executing *blocking* with profile configurations (cf. Blocking^P) even improves results by a factor of 4.2 (compared to default configurations). Moreover, *instance-of* matching turns out to be effective only for models with a low number of instances per metamodel element. Blocking and filtering produce the same results, yet blocking has the advantage of reducing the complexity of a matching process by decreasing the number of visited model-element pairs.

Analysis

In order to successfully conduct traceability scenarios, the collected traceability data needs to be sufficiently expressive. The above results show that metamodel-driven matching effective as *blocking* is the key-enabler for leveraging model matching for trace link generation by raising the matching quality up to an f-measure of 0.9. In doing

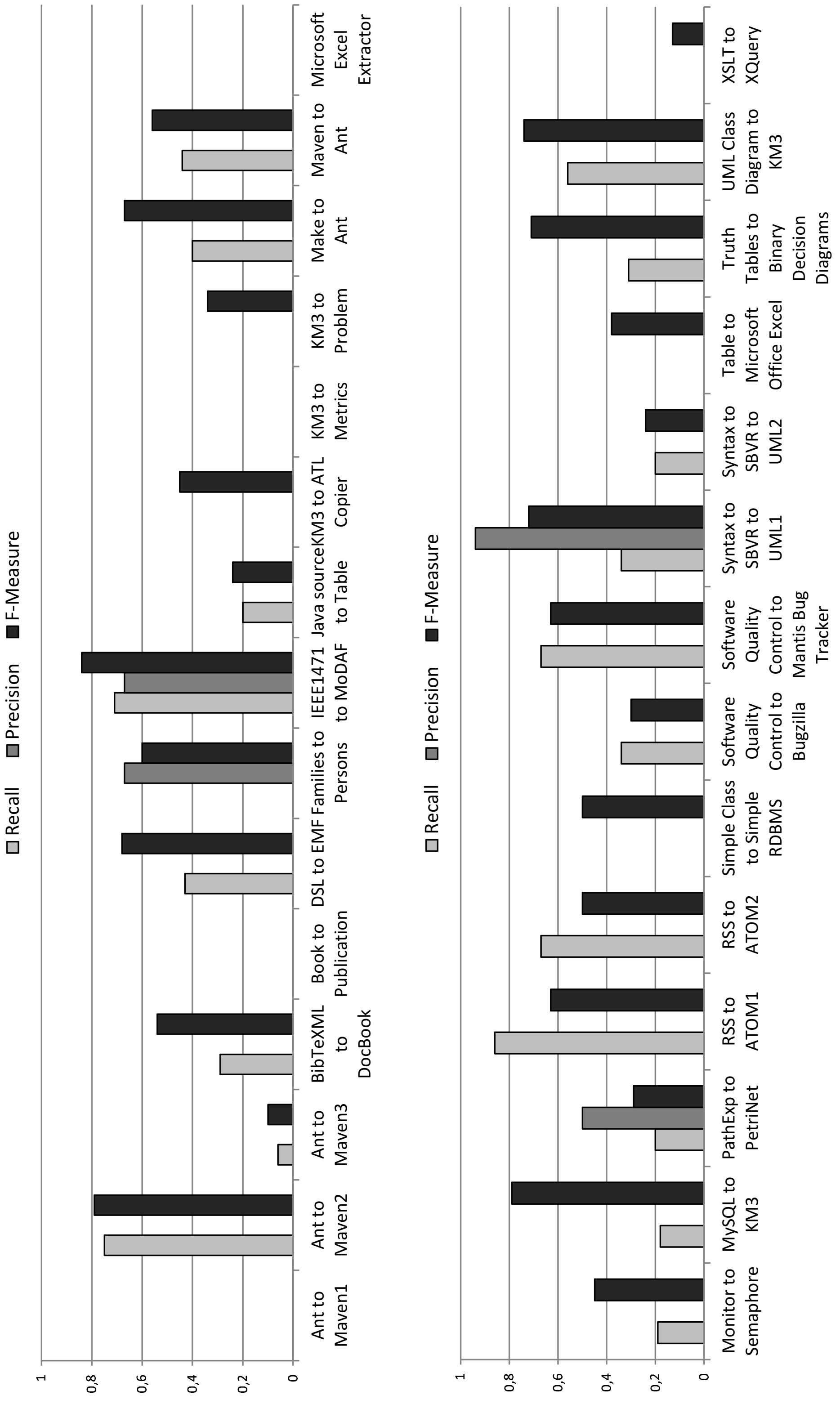


Figure 8.11.: Differences for Blocking applied to the ATL Zoo

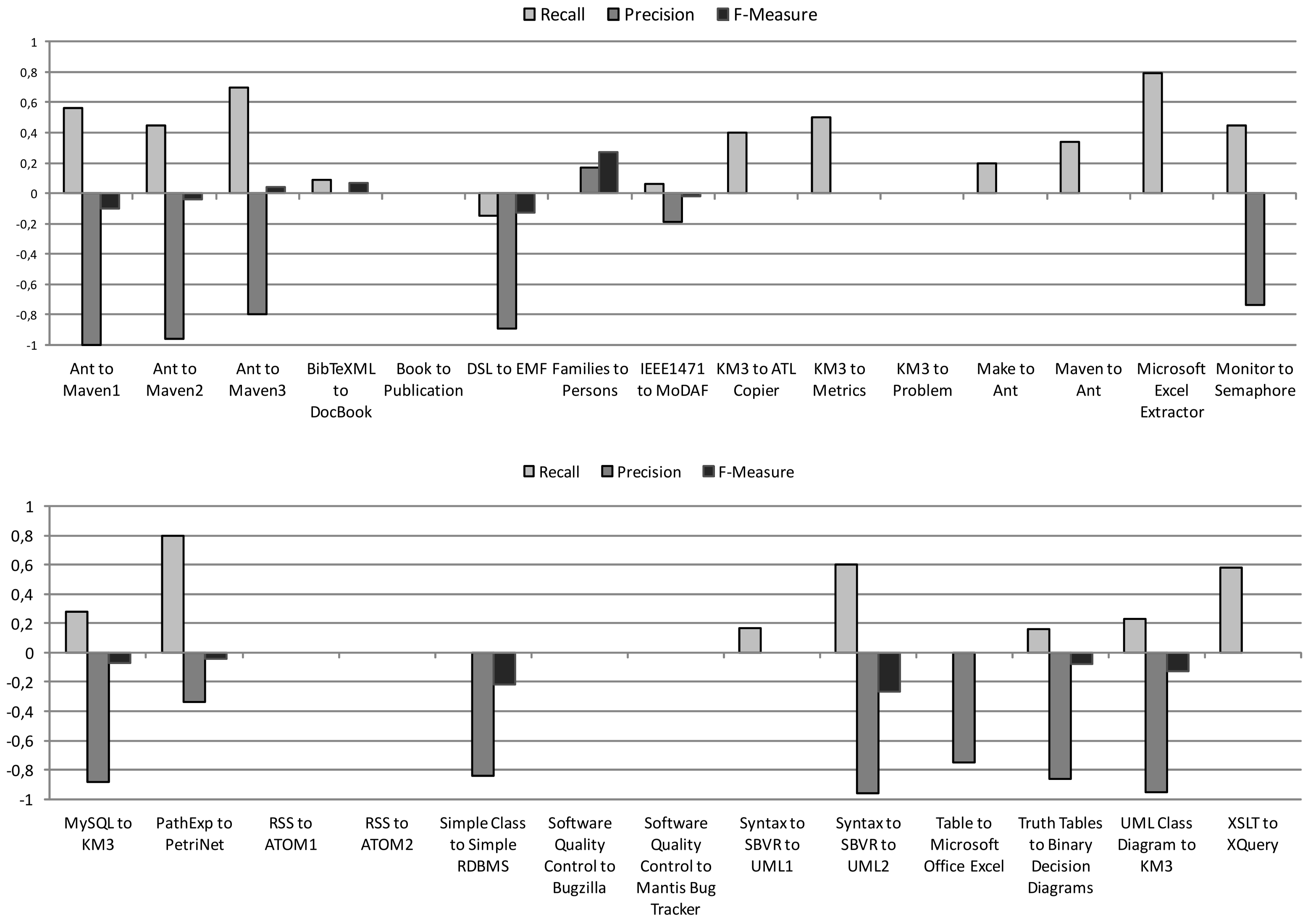


Figure 8.12.: Differences for Instance-of applied to the ATL Zoo

		Default	**Profile**	**Blocking^D**	**Blocking^P**	**Instance-of**
Sales Scenario	Recall	0.500	0.291	0.193	0.787	0.136
	Precision	0.174	0.539	0.556	0.971	0.104
	F-measure	0.204	0.338	0.257	0.851	0.072
ATL Zoo	Recall	0.144	0.261	0.366	0.885	0.143
	Precision	0.724	0.734	0.951	0.975	0.170
	F-measure	0.210	0.313	0.477	0.901	0.084

Table 8.6.: Summary of Average Results for Sales Scenario and ATL Zoo

so, our approach achieves a traceability-data expressiveness of approximately 90% with respect to the correctness and completeness of trace links.

8.2.7. Scalability

The scope of our work concerns the evaluation of matching quality, yet we provide a brief discussion of the scalability of our matching system in this section.

Figure 8.13 depicts the average time for performing single matching configurations against the model complexity. In this context, model complexity means the number of source-target model element combinations, since each matcher in the matching system needs to calculate the Cartesian product of source and target elements (without blocking). Based on the measured matching times of single configurations, an approximate exponential function is calculated. The latter is represented through a line, whereas the times are depicted by crosses. The vertical as well as horizontal axis is logarithmic. The runtimes for aggregation and selection are neglected. A further analysis on the runtime of individual matchers with respect to specific model combinations needs to be investigated.

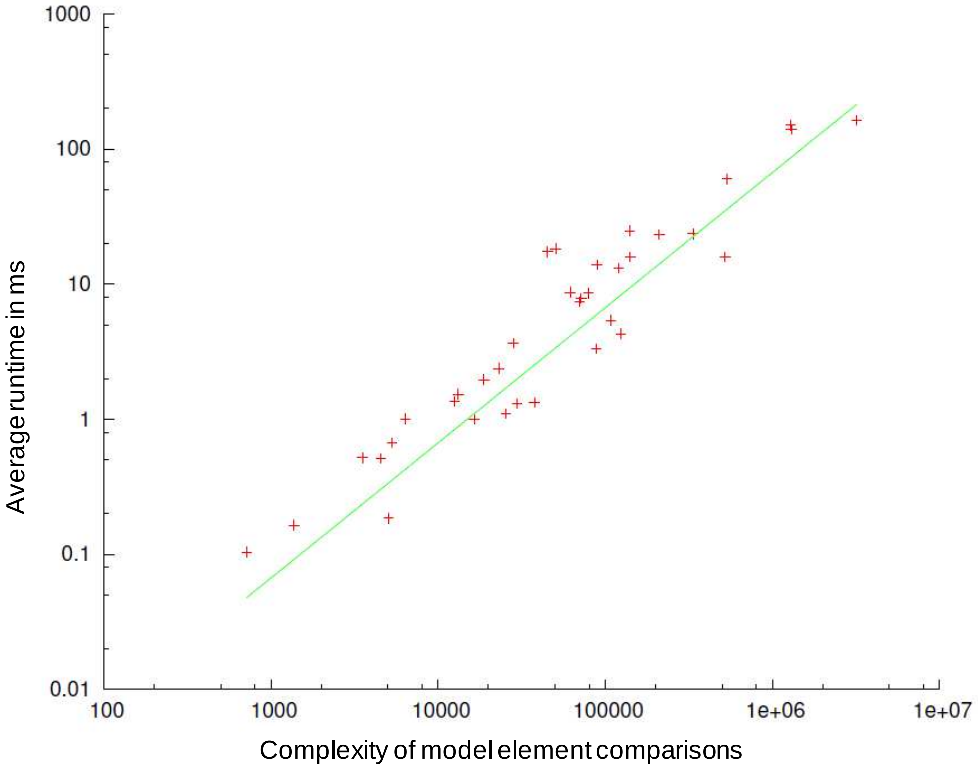


Figure 8.13.: Average Matching Time of Single Configurations against Complexity

9

Related Work

In this chapter, we present related work in conjunction with the GTF. Following the three categories of trace link generation and their solution approaches, we delve into related work with respect to the connector-based approach in Section 9.1 and the model matching approach in Section 9.2. Where appropriate, we describe the related work according to the objectives from our requirements analysis of Chapter 3.

9.1. Automatic CRUD Trace Link Generation

In the following, we outline related work on traceability solutions in MDSD. Detailed survey papers have been contributed by Aizenbud-Reshef et al. [ARNRSG06] and Galvao et al. [GG07]. In particular, we focus on approaches generating trace links through model transformations in view of those that address the problem of poor integration and quality of traceability data. In doing so, these approaches are most closest related to our work as described in Section 9.1.2.

If applicable, we describe the related work with respect to the following characteristics:

- Effort to achieve a traceability solution (cf. R1)
- Unification and extensibility of traceability metamodels (cf. R2 and R4)
- Integration of traceability data and existing traceability solutions (cf. R3)

In the following sections, we classify related work into how trace links are generated starting with general approaches, followed by traceability frameworks, that is, approaches that consider the integration of traceability data from arbitrary transformation approaches. Lastly, we present different works on the modelling of trace links.

9.1.1. Generation of Trace Links

Numerous model transformation approaches provide support for the generation of trace links. We classified these into the two classes of implicit and explicit generation. While all approaches provide a traceability solution, the integration of traceability data from arbitrary transformation approaches is not tackled. Our approach makes a point of reusing existing traceability solutions to consolidate the benefits of implicit and explicit trace-link generation and to tackle their disadvantages and challenges as motivated in the requirements analysis from Section 3.3.1. We mention several prominent examples in the following.

The ATLAS Transformation Language (ATL) uses a higher-order transformation (HOT) to augment the transformation program with traceability-specific code [Jou05]. The advantage of this approach minimizes the effort to achieve traceability for different transformations. In comparison, our approach is concerned with keeping the logic for traceability apart from the transformation logic to avoid potential pollution. Apart from this, the extensibility of the traceability metamodel is not considered as well as the integration with other traceability data. The ATLAS Model Weaver (AMW) [FV07] can be used to automatically generate trace links conforming to a specific weaving metamodel. In fact, the traceability metamodel used for the above-mentioned HOT is a specific extension of the core weaving metamodel of the AMW [OBF08]. Thus, the AMW accounts for an extensibility mechanism in that the user can capture different types of links in a specific weaving model. However, the actual augmentation of the model transformation with a traceability mechanism is realised through the HOT, as opposed to our approach. Apart from this, we additionally focus on the integration of traceability data due to model-to-text transformation approaches.

The Epsilon framework [KP13] provides an integrated traceability solution for numerous model operations, such as, model merging with the Epsilon Merging Language (EML), model-to-model transformation with the Epsilon Transformation Language (ETL) and model-to-text transformation with the Epsilon Generation Language (EGL) [RPKP08]. The traceability data can be kept in a separate traceability model as described in [KPP06b]. Since the former languages are part of a model management tool chain, the traceability models can be accessed by other model management tasks, such as validation or visualization [PDK+11]. In this sense, integration of traceability approaches is achieved for Epsilon languages. In contrast, our approach takes into account arbitrary transformation languages. Yet, with a blackbox connector for epsilon, the integration with other approaches could be achieved. Furthermore, the Epsilon traceability approach allows to model case-specific traceability models based on a domain-specific metamodelling language for traceability [DKPF08]. In comparison, our solution is based on a facet-based extensibility mechanism. The Epsilon comparison language is discussed separately in Section 9.2.4 as part of our matching approach.

The following approaches are from the class of implicit generation. MOFScript is a model-to-text transformation language with integrated traceability support [OO07]. Acceleo [Foua] is a commercial tool for code generation with traceability support for model-to-text transformations. Both tools do not provide support for the user to define case-specific traceability models.

9.1.2. Traceability Frameworks

The work of Falleri et al. [FHN06] describes a traceability framework written in Kermeta for generating trace links due to chains of model transformations. In this approach a model transformation is defined as a relation between the set of source and target model elements, therefore, the traceability data due to a single model transformation is reflected as a bipartite graph. The framework includes a simple traceability metamodel to describe trace links based on chains of model transformations. It includes a link, which references a source and target object. The object is the most general kind of element in Kermeta. A set of links due to one transformation composes a step, whereas a set of steps (due to a chain of transformations) composes a Trace. The approach of [FHN06] falls into the class of explicit trace-link generation, since the traceability-specific code needs to be added to the transformation program.

In summary, the proposed traceability metamodel is generic (not case-specific) with similar structures as the GTI traceability metamodel (cf. Artefact, TraceLink and UUID). However, it is not possible to model hyperedges (links) with the former. Apart from this, no extensibility mechanism has been foreseen, in contrast to our faceted-based approach.

Since, [FHN06] et al. suggest a traceability solution requiring the adaption of the transformation program with traceability-specific code, efforts to achieve traceability are not minimized, as argued for the explicit trace-link generation in Section 3.3.1. Furthermore, our approach regards traceability as a separate concern and therefore does not allow a potential pollution of transformation programs through traceability-specific code.

The AMPLE traceability framework (ATF) is a development as part of the AMPLE (Aspect-Oriented, Model-driven Product Line Engineering) project [Con]. Through applying the combination of aspect-oriented software development and MDSD, the project's focus is a software product line (SPL) development methodology that offers improved modularisation of variations and their holistic treatment across the software lifecycle. Integral to this methodology is the maintenance of traceability for these variations during SPL evolution as described in [AGG+08].

The traceability framework allows for a flexible specification of trace links between different kinds of SPL artefacts. In doing so, the definition of a variability model is required for the traceability of common and variable features along the domain and application engineering stages. Yet, although the framework focuses on the traceability of SPL artefacts, the design of the framework is generic to support traceability outside of SPL development [SKR+10]. The main functionalities are concerned with the creation and maintenance of trace links based on a traceability repository as well as front-end functionalities for querying and visualising traceability data. In order to collect traceability data for the repository an extractor needs to be implemented and registered to the ATF. For example, an extractor might be implemented to capture feature dependencies to related artefacts.

Furthermore, the framework is based on a generic traceability metamodel with an extensibility mechanism. A description of the traceability framework including the architecture and traceability metamodel can be found in [SKR+10]. In comparison to our approach on the unification and extensibility of metamodels, the ATF uses a fixed traceability metamodel, yet allows extensibility to encode any artefact and link type as an extension in the type hierarchy through the definition of profiles registered to the repository. The latter is an XML file apart from the metamodel and may be re-used for domains with similar traceability requirements. In contrast, our approach uses facets and faceted hierarchies, which can be varied independently for extending the type system for artefacts and links. Additionally, facet definitions may be re-used.

Regarding the effort to achieve traceability, a corresponding extractor needs to be implemented manually and registered to the ATF at a foreseen extension point. Nevertheless, yielding trace links automatically through transformation to minimize efforts is not the focus of AMPLE. Our connector-based approach provides a methodology for invasive and black-box connectors based on AOSD and a minimal set of CRUD link types. These connectors can be used as extractors given the correct accessibility.

The MODELPLEX (MODELing solution for comPLEX software systems) project's focus is on the development of an open solution for complex systems engineering improving quality and productivity. MDSD is a key-enabling technology for the development and subsequent management of complex systems. Numerous works regarding traceability in MDSD have been covered in Modelplex, [OO07], [Jou05], [FV07] (as discussed earlier), [WJSA06], [Ols09] etc. The latter is a trace analyser tool, TRAMDE, for storing trace links of different traceability solutions. Furthermore, this tool provides support for defining trace links manually, which is out of scope of this work. In contrast, our work focuses not only on the integration of existing traceability solutions, yet the means to achieve a traceability solution through different connector designs including an a posteriori traceability mechanism based on matching.

Walderhaug et al. [WJSA06] present a generic solution for traceability in MDSD that offers a set of services for specifying traceability data as well as related applications [WJSA06]. Their approach addresses, the definition of traceable artefacts, automatic and manual trace management, trace analysis services and trace query and navigation. In doing so, these concepts are supported by a so-called *trace model*, which needs to be implemented for a traceability solution. The main components of the trace model incorporate: a traceability metamodel for specifying trace models; a system component for creating, storing and using traces and finally, a repository component for users to interface the traceability repository. Based on these components, Walderhaug derives the mentioned set of traceability services, all related to the life-cycle management of traceability data. These concern the management of trace models, traceability repositories including functionality for data analysis and monitoring of traceability repository events.

In view of the unification and extensibility of traceability metamodels (R2 and R4), the approach in [WJSA06] requests the user to specify a trace model in advance to the tracing process as mentioned above. This model has a fixed definition during a given tracing process and thus necessitates the user to define which kind of artefacts should be traced. The GTI requires the definition of a traceability model as well, however the definition is a compromise between a fixed and variable and/or extensible definition. The latter is achieved through the use of facets and their power to re-combination. Therefore, a previous specification of what should be traced can be modified in contrast to [WJSA06].

Furthermore, Walderhaug et al. present a directive for modelling traceability data, which assumes attributes (name and type) for all model elements. For artefacts not based on a formal definition (i.e., metamodel) this does not hold, for example, the target of model-to-text transformations. Since our methodology for modelling traceability data accounts for model-to-model as well as model-to-text transformations, the GTI traceability metamodel needs to be able to define artefacts that are unstructured next to those, based on a formal definition. Therefore, the GTI traceability metamodel offers are more generic approach, where the only obligation to an artefact is a universal unique identifier.

Lastly, the requirement of minimizing efforts to achieve traceability is not covered in [WJSA06].

9.1.3. Modelling of Trace Links

In [PB08, POKZ08] a derivation process for a classification of trace links is presented. The authors define a classification as a view on the technical space of trace links. In doing so, two classifications are defined, for implicit and explicit trace links (see p. 16). The former is derived by defining and classifying possible MDSD operations. These are classified to different kinds of implicit traceability relationships. The identified operations refer to a model query, model transformation (as defined for our work), model composition, model update, model creation, model deletion and model serialisation (i.e., model-to-text). Furthermore, all operations can be concatenated to a chain of operations. The derivation of the explicit trace links differentiates between trace links inheriting from internal-model link and external-model links, which reflect model-to-artefact relationships. An detailed description can be found in [PB08].

In contrast, our work analyzes model transformations in detail based on the classification of source-target relationships of [KC05] and CRUD operations. Therefore, the view was adopted to derive link types to describe operations within the execution of a model transformation with an adequate expressiveness for conducting the presented traceability scenarios. A classification of explicit links is out of scope of this work. Furthermore, the focus of our work is to factorize hierarchies of link and artefact types and at the same time provide a feasible extension mechanism for artefact and link types through facets. However, the presented type hierarchies could be incorporated into the faceted hierarchies of our approach. Looking at the presented hierarchies, links are classified according to the kind of creation (explicit and implicit), implicit links based on their model operation and explicit links in terms of their durability (static and dynamic). Therefore, as an example, facets could be defined for the kind of creation, model operation, and durability.

We mention further link classifications in the following. In Ramesh et al. [RJ01] reference models for requirements traceability are defined. Users are categorized into low-end and high-end users of traceability. Thus, corresponding reference models are presented. The former offers a smaller set of different link types as the latter, yet both are defined with a fixed set of link types. No extensibility concept is discussed.

In [Ade12], Adersberger proposes a modelling language for traceability information. The approach describes an organisation scheme for traceability information and the process of traceability through an ontology and a value chain. In contrast, our work uses a facet-based approach for reasons as explained previously.

Limon et al. propose a unifying traceability scheme in [Lim09]. The derivation process is based on a the following analysis criteria, similar to those in the work of [RJ01]: process-related or product-related links; pre-requirements and post-requirements traceability relations categories; the traceability link purpose; and the items or objects to which the traceability link will relate. The unifying schema results from the common features of the above analysis of different schemes and is composed of a traceability link type set (to define which attributes a link type will define), a minimal set of traceability links (for a specific project or traceability baseline), a metrics set to verify quality requirements and a traceability link dataset. The traceability scheme (metamodel) is extensible with respect to its language constructs based on the power type pattern.

9.2. Model Matching for Trace Link Generation

In the following section, we give an overview on approaches related to or dealing with model and/or metamodel matching. None of the following approaches have addressed nor tackled the problem of generating trace links on the basis of model and/or metamodel matching with the same language genericity, quality (precision and recall) of acquired matches and evaluation (authenticity and model size of evaluation scenario(s)) as our approach.

Model Matching is related to the field of *schema matching* and *ontology matching* (also called alignment). Yet, the ideas of these matching approaches are based on finding correspondences between source and target being on the same abstraction level. For a detailed survey, we refer to [CSH06, RB01, SE05]. The same applies to entity matching

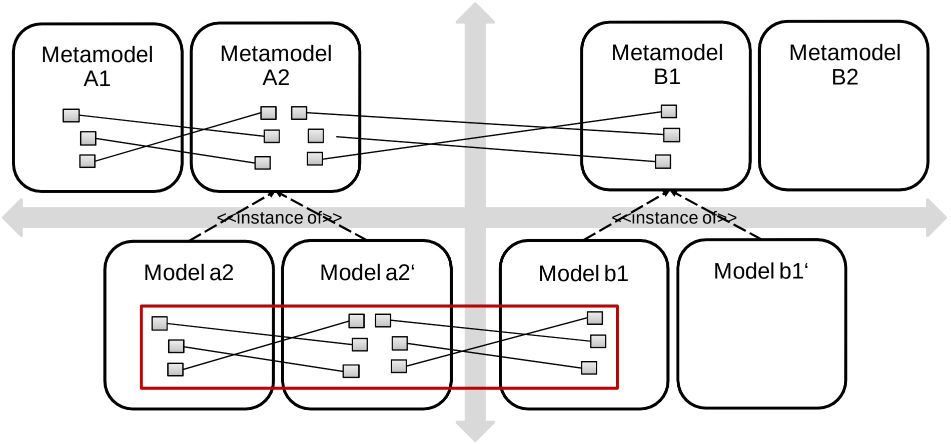


Figure 9.1.: Dimensions of Model Matching

(also referred to as duplicate identification, record linkage, entity resolution or reference reconciliation), which focuses on identifying entities (objects, data instances) referring to the same real-world entity for the sake of data integration and data cleaning [KR09]. Our approach additionally allows for source and target models to conform to different metamodels, thus allowing models to be on different levels of abstraction. The demarcated red area in Figure 9.1 shows matching between models conforming on to the same as well as different metamodels. To underline the difference of these matching cases, we use the notion of *intra-matching* in the first case and *inter-matching* in the second.

More closely related is the field of metamodel matching [VIR10]. Yet, these approaches do not account for model-specific matching requirements, such as, leveraging on model-specific attributes for matching, or import mechanism for instance-of relations from a model into a graph structure. Hence, these need to be extended, for example as in the case of our approach with MatchBox.

In the following, we focus on model matching, the most closest related to our work, where we evaluate the different approaches according to the quality and genericity of approaches. In particular:

1. the *quality* with respect to correctness and completeness of acquired matches,
2. the *genericity* regarding the language applicability aligned with R7 on the language-independence of a traceability solution and
3. the user *effort* needed for the matching approach (cf. R6 on the automation of achieving a traceability solution).

A range of technologies are available for comparing (differencing) and merging of models as classified by [KRPP09], both pertaining to model matching. In the following sections, we compare these classes to our approach.

9.2.1. Static Identity-Based Matching

One possibility to computing model differences is static identity-based matching. This approach necessitates a closed development environment in which all model editors and other tools, which modify models, assign and maintain a persistent unique identifier at each model element. In such a context, one can efficiently compute differences on the basis of persistent unique identifiers. Thus, model matching is reduced to finding equal identifiers, however relies on the maintenance of the persisted identifiers. For example, in [AP03, EPK06] a metamodel-independent algorithm is proposed to calculate the difference and union of MOF-based models in the context of a version control system.

Identity-based matching provides the following advantages. There is no need for configuration. Furthermore, the used algorithm is simplified due to the given uniqueness of elements, which increases the algorithm's efficiency. However, the approach relies on the maintenance of identifiers and is dependent on the history of changes applied to models. The latter may cause invalid results, when models are created independently from each other, e.g., when both versions have been created with the same contents, yet independently by different users. Furthermore, this matching approach is tool-dependent, thus requiring a standardized handling for the management of unique identifiers, if used across tools.

Our approach to matching takes a different direction, in terms of the advantages of an import-based matching system, that is, import and matching of models conforming to arbitrary metamodels. In case the above-mentioned disadvantages are avoided, it is however possible to integrate a static identity-based matching algorithm into our framework. The implementation would follow an attribute matcher, that checks whether the UUID of two given model elements is equal, for example on the basis of XMI identifiers within the EMF framework.

9.2.2. Signature-Based Matching

Signature-based model matching approaches do not rely on a static identifier, as described above, yet compute a signature for each model element. Such a signature is based on certain characteristics of model elements that are used for uniquely identifying model elements. Hence, a certain effort for configuration is required for defining the way of calculation for the signature. In [RFFB05], an algorithm for model composition is presented using signature-based matching.

Signature-based matching approaches do not use an import-based matching system as in our approach. While our approach applies multiple similarity measures to a model element, a signature may be interpreted as a single similarity measure, which analyses the same model characteristics for each model element. Our approach investigated the benefits of applying numerous matching algorithms.

A signature-based algorithm requires configuration effort for defining the signature calculation. If this configuration effort is language-specific or domain-specific, this stands in contrast to the language-agnostic approach of our matching system. Nevertheless, a signature-based algorithm could be used as an additional matcher in our matching framework.

9.2.3. Similarity-Based Matching

Similarity-based matching focuses on aggregating different similarity metrics applied on model elements to obtain matches. The metrics are defined on the features of model elements. A similarity-based matching algorithm usually determines the similarity between two elements by either local features or by elements in the near proximity, e.g., referenced- or child-elements. Typically inherent to the matching algorithm is a certain configuration that specifies the relative weight of each feature. The data model upon which matching takes place may be represented as typed attributed graphs. Typical examples that follow these approaches are:

SiDiff [KWN05, SG08] is based on a generic difference algorithm for UML models (generally, metamodel independent), supporting the three major state of the art matching strategies, i.e., ID-based, signature-based and similarity-based matching. It is also possible to combine the different approaches where applicable, e.g., using an id-based matching first and apply a signature-based matching on the remaining unmatched model elements.

SiDiff uses an intra-matching approach with an algorithm traversing a tree bottom-up and top-down similar to the propagation of our Connection similarity measure. A possible simulation could be achieved through a parent matcher combined with a leaf matcher. However, SiDiff uses language and/or domain-specific characteristics to weight the similarity results, while our approach follows a language-agnostic matching framework for intra and inter-matching. Regarding the matching of UML models, SiDiff makes use of UML-characteristics, by weighting similarity results according to element types and therefore evaluation results are likely to outperform our approach in this certain domain. In contrast to our graph model, the data model of SiDiff is limited to containment (and reference) relationships and lacks, e.g., representations of inheritance or instantiation as used in metamodel-driven matching in our approach.

DSMDiff [LGJ07] is a differentiation tool for domain-specific models. The underlying metamodel-independent algorithm detects mappings and differences between domain-specific models (intra/inter-matching) with the use of signature-based matching. In doing so, the algorithm follows a level-wise approach, starting at the root nodes of source and target model and continuing at the level of their children. A level thus reflects a parent-child relationship. At each level, node comparison is performed to detect the node mappings by using a) *signature-based matching*, followed by b) *structural matching*, to detect the edge mappings and differences. These steps are repeated on the mapped child nodes until the bottom level is reached.

a) *Signature matching* refers to the comparison of nodes and edges with the help of the node and edge signature, respectively.

- Node signature matching: The node signature is the concatenation of the type, kind and name of a node.
- Edge signature matching: The edge signature is the concatenation of the type, kind and name of an edge as well as of the signature of its source and destination node.

In both above cases, matching refers to finding equivalent strings of the signatures. In case two given node signatures (or edge signatures) are equivalent, their corresponding nodes (or edges) define a match.

b) After signature-based matching, a given node from the source model may have more than one matching candidate. In this case, the signature cannot identify a unique mapping. Therefore, *structural matching*, using edge similarity, is performed in order to find the most similar candidate. In calculating the edge similarity, each edge connecting to a given candidate node is matched (through edge signature matching) to each edge of the source node. This is repeated for all candidate nodes. The candidate with the maximum edge similarity is chosen to match the corresponding node from the source model.

In summary, the DSMDiff algorithm is based on two metrics (i.e., signature matching and structural similarity), which are applied at each level. The algorithm is a local graph-based matching approach with the restriction to only compute one-to-one matches, that is, a node can only belong to one match relationship as opposed to our approach. Furthermore, signature matching relies on the calculation of string equivalence, a fixed condition with no possibility of configuration to use more flexible element-level matching approaches, e.g., Trigram, or string-edit distance. On the contrary, our approach is based on a configurable parallel matching system, with the capability of gaining the maximum possible quality of matching results through configuration. In addition, our approach employs global graph-based matching algorithms next to local ones.

Another approach from the category of similarity-based matching is presented in [RV08] by Rivera et. al. As DSMDiff, this approach is a generalization of the above-described works from [SG08, XS05, AP03]. The authors of [RV08] propose an algorithm for calculating differences between models conforming to arbitrary metamodels, next to a modelling management environment based on the Maude language. The matching part of [RV08] is based on persistent identifiers (as in [AP03] using static identity-based matching) and structure-level algorithms. Regarding the latter, the following 5 measures for structural similarities are incorporated into the matching process (for class and structural features). Two metaclasses match if they are the same, or there exists an inheritance relation between them. *Boolean attributes* and enumerations match if they have the same value. *String attributes* are matched, based on the Levenshtein edit-distance [WF74] of their values. *Numerical attributes* are matched by applying a relative distance function to their values. *References* are matched recursively, by applying the same match operation to non-reference attributes (to avoid cycles).

The algorithm compares each model element pair from the Cartesian product of the elements of source and target model. Thereby, each pair is compared on the basis of the above-mentioned measures. Afterwards, their results are aggregated as weighted average to an overall similarity. The final matches are derived by applying a certain threshold to all values and secondly by selecting the highest similarity value for each model element, which restricts the approach to one-to-one matches. In comparison, our approach is not limited to this cardinality, yet allows 1:n matches (depending on the configuration of the similarity value cube). In fact, in model-to-text transformations, 1:n matches need to be considered.

Furthermore, the structural similarity metric is restricted to the comparison of referenced elements. In contrast, our approach applies graph matchers (graph edit distance and pattern matcher) considering the complete graph structure (global-based graph matching). The graph edit distance matcher turns out to achieve best results for the model-to-model-transformations from the ATL Zoo (cf. Section 8.2).

Similarity Flooding [MGMR02] The similarity flooding algorithm is a hybrid matching algorithm that is based on the premise, that, if two elements are similar, then there is also a certain similarity between their neighbouring elements. In fact, similarity flooding was first implemented for schema matching as in [MGMR02], but has since then been adapted to be used in other domains, e.g., ontologies [ZZLT10] and metamodels [FHLN08]. The internal data model used in [MGMR02] is able to represent directed labelled graphs and the algorithm runs in an iterative fix-point computation to produce a mapping between nodes of the input graphs. First, string-based matching techniques (analyzing common prefixes and suffixes) are applied to obtain an initial mapping for the fix-point computation. Starting from similar elements, the similarity is propagated to neighbouring elements through propagation co-efficients. The process is finished, when a fix point is reached, upon which the final mapping is returned. In addition, the resulting mapping may be filtered by applying constraints, namely typing, cardinality and selection metrics.

In contrast to the matching system for traceability, similarity flooding provides an internal data model representing directed labelled graphs. Since directed labelled graphs need to first be extended to express typed attributed graphs [EEPT06], our requirements to a generic internal model are not met. Furthermore, the similarity flooding algorithm analyses different metrics and is configurable (in terms of the filter mechanism mentioned above). To draw a parallel, our matching system is configurable and aggregates matching results of numerous matchers. In particular, the application of the typing constraint is similar to the metamodel-driven matching techniques blocking and filtering. Yet, similarity flooding does not apply algorithms considering the complete graph structure as the graph matchers (graph edit distance and pattern matcher) of our approach, but rather investigate neighbouring relations. Additionally, the algorithm depends on an initial mapping being propagated. Hence, an occurring error is multiplied through the propagation. As shown in the work of [Do06], a parallel matching system tends to be superior to similarity flooding. For these reasons, we have not based our work on similarity flooding.

EMF Compare [Foue] provide a similarity-based matching algorithm for model comparison and merging of EMF models. The algorithm is built-in and thus requires no configuration effort for the user. In general however, the EMF Compare framework is extensible for integrating custom specific-language algorithms (cf. Section 9.2.4). The framework engine is composed of a generic matching engine as well as a differencing engine, hence the algorithm constitutes two corresponding phases. Furthermore, the algorithm pertaining to the above-mentioned engines is metamodel-independent and is defined through four similarity metrics, which are aggregated to an overall result:

- Name similarity: first the attribute, which is the best candidate to be the identifying name is retrieved, afterwards string-based matching of the names takes place
- Type similarity: compares the meta-class features
- Value similarity: uses all attributes values for the similarity calculation (incorporates a filter mechanism, as described below)
- Relations similarity: considers the linked instances based on containment and non-containment relations

Beyond the similarity metrics, EMF Compare improves its results by applying a filter mechanism on certain model characteristics, for example, the setting of default values in model instances. If used for the similarity calculation, this kind of information potentially lowers the matching quality, such as for the value-similarity metric. Thus, filters may be applied to the to-be matched models to exclude inferring information from the subsequent matching process.

Moreover, EMF Compare reduces complexity to gain performance in limiting the number of combinations of source and target elements. Instead of executing comparisons sequentially for the Cartesian product of source and target elements, EMF Compare analyses both models at the same time while matching elements within the limits of a given search space. This approach follows the assumption, that the probability of moving an element outside a certain ”neighborhood” is low. Upon completion of the analysis, elements, which have not been matched, are in turn compared with each other.

EMF Compare is the closest related to our approach for the following reasons. The combination of numerous similarity metrics is applied in accordance with the combination effect of our parallel matching system. The internal data model of EMF Compare is able to express typed attributed graphs. Furthermore, the language-independent approach of the generic match engine covers the language-independent aspect of our approach. No configuration effort is requested for the user regarding the built-in generic algorithm. Yet, the potential to configure similarity metrics is given, as in the case of our matching system (matcher combination, aggregation and selection strategies etc.).

In comparison to EMF Compare, our matching system has not invested in scalability to an extent as the above-mentioned for EMF Compare. However, the focus of our matching system lies on the quality of matches. Given that the matching system for traceability is required to derive traceability data from the matching results, clearly, the quality of matches has a very high priority (even over scalability, if there is a trade-off between scalability and quality). Poor quality of traceability data is likely to disqualify certain traceability scenarios. Regarding the quality of matching results, a comparative evaluation was conducted between EMF Compare and our matching framework to evaluate this quality. The evaluation is based on the evaluation scenarios from Section 8.2.4, that is, the ATL Zoo and corresponding gold mappings. The evaluation's results show that EMF Compare identified correct matches for 29% of the available source-target combinations, with no correspondences in the Sales Scenario and a total of 12 combinations from the ATL Zoo. Regarding the latter, the resulting recall, precision, and f-measure is depicted in Figure 9.2. The vertical axis indicates the values of recall, precision, and f-measure and accordingly ranges from 0 to 1. In addition, combinations of the source and target model are plotted on the horizontal axis. The resulting average f-measure of 0.1 (average recall of 0.10 and precision of 0.18) is doubled by the *default* configuration and tripled by the *profiles* of our matching system. When applying metamodel-driven matching techniques, in particular, *blocking*, EMF Compare is outperformed by a factor of at least 4.7 (cf. Table 8.6). As a result, the matching system for traceability is superior over EMF Compare in terms of matching quality and the proposed evaluation scenarios.

9.2.4. Custom Language-Specific Matching Algorithms

Custom specific-language algorithms are specialized on a particular modelling language. For example, the algorithm in [NSC$^+$07] specializes on state charts, while **UMLDiff** [XS05] is a domain-specific algorithm for calculating differences of UML models. The latter algorithm takes two class models of a Java software system as input and produces a change tree as output. Thereby, the algorithm works on an internal data model conforming to a Java representation and thus, requires an initial transformation (reverse engineering) of corresponding Java source code into the input class models. Furthermore, UMLDiff is based on a structure-level algorithm with top-down traversal, combining a name matcher and relying on UML semantics.

Another approach within the category of custom language-specific algorithms is **ECL**, called, Epsilon Comparison Language[1] [Kol09, KPP06a]. ECL is a domain-specific language for specifying comparison algorithms that are language-specific, that is, incorporate the semantics of a targeted language. In specifying the algorithm, a user is required to define domain-specific *match rules*, which account for the comparison logic. In doing so, the user profits from the language's specificity to the domain of matching, furthermore, the semi-automatic approach for implementing algorithms as well as the

---

[1] Epsilon Comparison Language: http://www.eclipse.org/epsilon/doc/ecl/

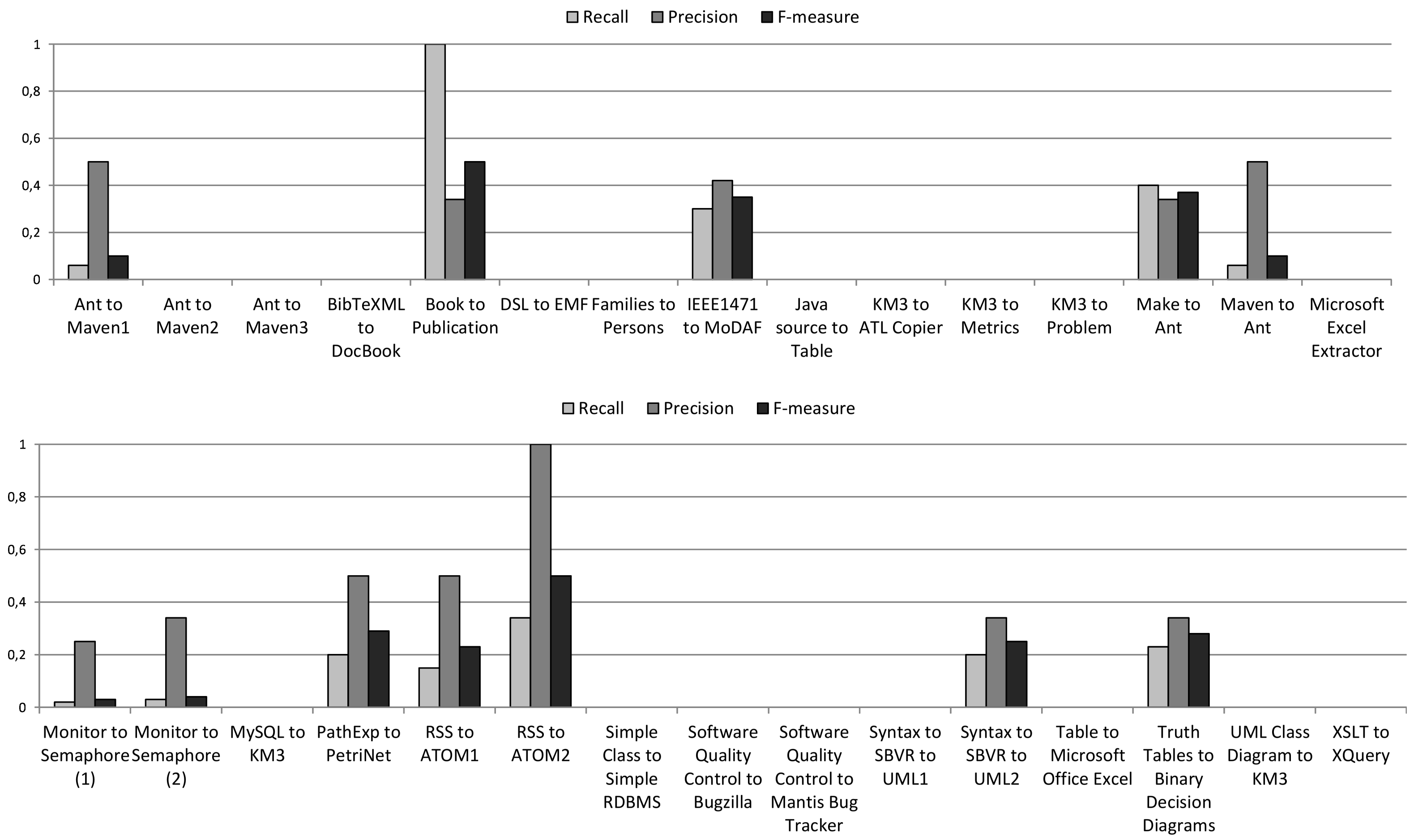


Figure 9.2.: Evaluation Results of EMFCompare for the ATL Zoo

high abstraction level of specification. Thus, the effort of implementation is reduced as opposed to other custom language-specific matching algorithms such as UMLDiff, that require the implementation of a complete algorithm. Beyond that, ECL is integrated into the Epsilon[2] framework, providing a whole range of domain-specific languages and tools for tasks revolving around MDSD.

In summary, custom language-specific algorithms use a domain-specific matching approach, as opposed to our parallel matching system with the claim of language generality. Regarding UMLDiff, the internal data model is constrained to a Java-specific model and is thus bound to language-specific matching (and comparison) of UML class models. ECL algorithms depend on metamodel-specific information to a high degree. Furthermore, custom language-specific algorithms require a considerable effort for implementing an algorithm. In UMLDiff, this entails the complete algorithm, while ECL uses a semi-automatic approach through the specification of match rules. The latter reduces the effort to some extent. On the other hand, the advantages of using language semantics in the matching process increases the algorithm's efficiency (quality of matches and search space) [KRPP09, FW07] for a given language-specific use case. However, our approach takes a different point of view by requiring no effort for the user and applying a parallel matching system to achieve the maximum possible quality of matching results. To benefit from the above-mentioned advantages, an integration of custom language-specific algorithms into our matching system seems promising, provided the applicability of algorithms to a given matching task. In analogy, an integration of the proposed matchers of our work into ECL is conceivable. Given that ECL supports element-level matching algorithms, such as (fuzzy) string-based and linguistic resource-based algorithms, in effect, these language constructs could then be extended towards our matching algorithms.

---

[2] Epsilon: http://www.eclipse.org/epsilon/

Approach	Genericity	Abstraction Level	Configuration	Metamodel-Driven Matching
Alanen et al., Engel et al.	MOF/EMF-specific	inter/intra	–	–
SiDiff	language-independent/UML	intra	weights and threshold per type	–
DSMDiff	language-independent	inter/intra	–	–
Rivera et al.	language-independent	inter/intra	weighted average and threshold	metaclass feature measure
Melnik et al.	language-independent	inter/intra	filtering through constraints	filtering through typing constraint
EMF Compare	language-independent	inter/intra	fixed configuration/specification of custom algorithm	–
UMLDiff	language-specific/UML	intra	–	–
ECL	language-independent	inter/intra	semi-automatic specification of algorithm through custom matching rules	–

10

Conclusion

In this chapter, we conclude this thesis by revisiting our stated contributions of the introduction and show, how these validate the hypotheses of our work. Finally, we provide an outlook on future work.

10.1. Conclusion and Contributions

In this thesis, we introduced a generic traceability framework for automatically generating trace links in MDSD. The framework accounts for different use cases, which are grouped into three categories of trace link generation. The fist category deals with the generation of trace links by using the integral mapping of a transformation to derive trace links in parallel to its execution. The second category handles the generation of trace links after the transformation execution and source and target artefacts have evolved. Finally, the third category covers the generation of trace links independent from a transformation engine due to its non-existence or inaccessibility. The latter cases occur, for example, if transformations are implemented manually (non-existence), or the transformation engine is proprietary (inaccessibility).

In dealing with the above categories two independent solutions are presented. In view of the first category, we proposed a design pattern in terms of a methodology to augment arbitrary model transformations with a specific traceability mechanism. The design pattern is realized through a generic traceability interface for arbitrary model transformation engines. Regarding the second and third category, we presented a matching component. Both solutions are based on a generic traceability metamodel to which all generated traceability data conforms to and are integrated into the generic traceability framework.

In the introduction, we claimed three contributions:

- C1 Facet-based Modelling of Traceability Data with CRUD Trace Links
- C2 Design Pattern on Augmentation of Model Transformations with Trace Link Generation
- C3 Parallel Model Matching for Trace Link Generation

Each contribution is directly related to one of the above-mentioned components: The traceability metamodel (C1), the generic traceability interface for arbitrary transformation engines (C2) and the model matching component (C3).

In the following sections, we discuss these contributions with respect to their fulfillment of the requirements analysis and the evaluation results. The goal of this discussion is to validate, the stated hypotheses of the introduction. To recapitulate, the stated hypotheses are:

- H1 The quality of traceability data is improved through:
  - a generic interface for integration of traceability data
  - a generic traceability metamodel
  - a facet-based extensibility mechanism of the traceability metamodel.
- H2 Efforts to achieve traceability are minimized (for the explicit generation class) through augmentation of the transformation engine with the means of trace-link generation.
- H3 Parallel model matching can be leveraged for the generation of trace links regarding inaccessible, or non-existing transformation engines.

H1: Increasing Quality of Traceability Data

As per requirements analysis, three objectives need to be fulfilled to increase the quality of traceability data. These are the unification (R2) and extensibility (R4) of traceability metamodels. Furthermore, a unified interface for integration of traceability data (R3) is needed. In answer to the first two requirements we proposed C1, while for the latter C2, as we will discuss in more detail in the next two subsections.

C1: Facet-based Modelling of Traceability Data with CRUD Trace Links

The unification of metamodels (R2) is fulfilled through the generic traceability metamodel of the GTF. To account for a sufficient expressiveness of traceability data and in answer to R4, an extensibility mechanism based on facets is proposed. The specification of user-defined artefacts and link types in traceability models is achieved by defining corresponding types as facets. Since facets factorize inheritance hierarchies and thus, simplify them, we use this advantage for the sake of simplifying artefact and link type hierarchies. Furthermore, since facets can be varied independently, the extensibility of traceability metamodels is achieved. This has the advantage that a priorly defined set of types may be varied through the re-combination of facets and thus does not have to be fixed to hold for all possible traceability scenarios. Therefore, case-specific traceability models may be defined on a selection of facets. In addition, the configuration of *granularity* (for selecting the granularity level, and thus checking for the existence of certain facets, e.g., text-file facets excluding text-block facets) and *scope* (for selecting a facet-specific property) contribute to the expressiveness of case-specific traceability models. Finally, the minimal set of elementary links based on CRUD actions captures the semantics of all possible relationships between source and target elements of a model transformation. The set of CRUD links account for an adequate expressiveness to conduct the traceability scenarios evaluated in Section 8.1.4. Thus, we conclude that C1 fulfills R2 and R4.

C2: Design Pattern on Augmentation of Model Transformations with Trace Link Generation

Regarding the realisation of R3, we proposed a design pattern and a methodology to augment arbitrary model transformations (model-to-model as well as model-to-text transformations) with a specific traceability mechanism. The design pattern is realized through a generic traceability interface for arbitrary model transformation engines. For the explicit and implicit generation approach, we presented two possible augmentation methods to achieve a traceability mechanism, respectively: a) Augmentation of the transformation-engine logic based on aspect-oriented programming and b) Augmentation of the traceability-data output through the use of a model transformation. In terms of the generic traceability interface, these augmentation methods require the implementation of two different kinds of connectors, that is, *blackbox* and *invasive* connectors for implicit and explicit generation, respectively. Once the augmentation method has been applied in either of the two cases, the transformation approach is featured with a traceability mechanism that generates traceability data conforming to the proposed traceability metamodel. We evaluted the design pattern in terms of three connectors for QVT and ATL (blackbox connectors) as well as Xpand (invasive connector).

Effectively, the design pattern (generic interface) provides the means to integrate traceability data from arbitrary transformation approaches, either through blackbox or invasive connectors by enforcing the conformance to the traceability metamodel. Therefore, R3 is fulfilled.

To validate hypothesis H1, we provided a generic traceability interface including a generic traceability metamodel to tackle the problem of *aggravated standardization and integration of traceability data* (P2) by realizing R2 and R3 (cf. Figure 3.4 on ZOPP approach). Secondly, we solved the problem of *inexpressiveness of traceability data* (P3) through the facet-based extensibility mechanism in conjuction with R4. All these fulfilled objectives contribute to generating traceability data with an adequate traceability expressiveness and conformance to a generic traceability metamodel. Essentially, this allows easier reasoning over traceability data with respect to traceability scenarios, leading to an increase in the quality of traceability data.

H2: Minimization of Efforts to achieve Traceability

In answer to the minimization of efforts (R1) for the explicit generation class, we proposed to use an invasive connector (cf. C2). After taking into account the initial effort of developing the connector, the following advantages hold: a) no likely error-prone and time-consuming factors of manual encoding of traceability-specific rules, b) no reoccurring efforts for individual model transformations and c) no pollution of transformation programs. Thus, R1 is fulfilled. Concluding, we have tackled the problem of reoccurring efforts to achieve traceability for the explicit class (P1) and thus confirmed H2.

We now turn to the second and third category of trace link generation.

H3: Leveraging Parallel Model Matching for Traceability Data

The derived requirements for these categories are: *unification and extensibility of traceability metamodels* (R5), *automation of achieving traceability without transformation engine* (R6) and *language-independence of a traceability solution* (R7).

C3: Parallel Model Matching for Trace Link Generation

In realization of the above-listed requirements, we proposed a parallel model matching system for generating trace links for arbitrary source and target models. As claimed in the introduction, this system tackles the problem of *lacking traceability data due to non-existing or inaccessible transformation engines* (P4). We look into C3 to discuss this claim.

The matching system is founded on the idea of using a graph-based internal data model based on typed attributed graphs upon which the matching process takes place. On the grounds of this data model, we derived three novel, language-agnostic similarity measures for model matching:

1. **Attribute Similarity Measure**: Similar *data nodes* from source and target graphs, indicate shared characteristics, referred to as *attributes*, and thus, a potential similarity between the *graph nodes* that the *data nodes* are connected to.
2. **Connection Similarity Measure**: The similarity between a set of source and target children nodes acts as a similarity measure. The measure is based on the rationale that similar *children* graph nodes, have similar *parent* graph nodes. Thus,

the connectivity of a graph node to its children graph nodes is used to propagate the similarity from child to parent node.

3. **Instance-of Similarity Measure**: We base the matching process on model level on the results of metamodel matching by making use of the *instance-of* relation. Thus, we investigate the outcome of propagating the similarity of metamodel elements to their conforming model elements.

Furthermore, the implementation of our approach is build upon the metamodel matching framework, Matchbox [Voi11], which we extended with eight matchers in realisation of the three derived similarity measures. Since metamodel matching does not incorporate the use of attribute values of model instances, the metamodel matching algorithms from [Voi11] cannot be used to instantiate the proposed similarity measures. Therefore, eight new matching algorithms are implemented and integrated into Matchbox. Furthermore, we exploited metamodel-driven matching with *filtering*, *blocking* and *instance-of* matching.

Returning to the claimed requirements, (R5) is fulfilled, since the mappings resulting from matching are transformed into traceability data conforming to the traceability metamodel of the GTI (C1). Regarding the automation of achieving traceability (R6), no user effort is required, once a suitable matching configuration has been found. Yet, finding a suitable configuration is bound to effort. Once the configuration is set and the matching system is pre-configured, the generation of trace links works automatically, letting aside potentially incorporated manual processes of the extraction phase. Hence, our proposed system allows for the generation of trace links in a semi-automatic manner. We will reflect on this issue in Section 10.2 on future work. Finally, the matching system is language-independent, therefore (R7) is satisfied.

The evaluation of our approach is based on the ATL Zoo (29 mappings) and a SAP business application (12 mappings). The results show that configuration *profiles* achieve $1.5 - 1.7$ times better matching results with respect to *default* configurations. Furthermore, metamodel-driven *blocking* on default configurations improves matching results by a factor of $1.3 - 2.3$, whereas on profile configurations the results are even improved by a factor of $4.2 - 4.3$. The above results show that metamodel-driven matching effective as *blocking* is the key-enabler for leveraging model matching for trace link generation by raising the matching quality up to an f-measure of 0.9. In doing so, our approach achieves a traceability data expressiveness of approximately 90%[1] with respect to trace link correctness and completeness.

Regarding the validation of hypothesis H3, this proves that parallel model matching can be leveraged to generate trace links. We thus tackled the *problem of lacking traceability data due to non-existing or inaccessible transformation engines*. However, in order to successfully conduct traceability scenarios, the collected traceability data needs to account for adequate data expressiveness. Whether the matching quality (with an f-

---

[1] requiring that all mappings are extracted as trace links

measure of 0.9 regarding the evaluated scenarios) of our matching system suffices for the conduction of traceability scenarios is bound to future work.

Nevertheless, we contributed an augmentation method for inaccessible or non-existing transformation engines (blackbox systems) to achieve a traceability mechanism. This augmentation method is based on a novel combination of similarity measures for matching in the category of similarity-based matching algorithms.

In conclusion to this thesis, we turn back to our requirements analysis reflecting all three categories of trace link generation (cf. Figure 3.5). In fulfilling all requirements, as we have shown, the quality of traceability data is increased and costs to achieve traceability are decreased. This leads to the overall goal of increasing traceability practice and with this, the quality of software.

10.2. Future Work

In the following, we discuss possible directions for future work on the generic traceability framework pertaining to the matching system as well as the generic traceability interface.

Matching System

Improvements and future development is discussed regarding the effort to gain suitable configurations for matching, secondly the quality and scalability of matching.

Configuration Effort Regarding the matching component, the best possible quality is inherent to an optimal configuration. However, finding such a configuration is bound to effort. Once the matching system is pre-configured, the generation of trace links works automatically, letting aside potentially incorporated manual processes of the extraction phase. Hence, our proposed system allows the generation of trace links in a semi-automatic manner. Since from a traceability-user perspective, the effort should be as low as possible and in the ideal case automatic as required by our requirements analysis, we envision future work on minimizing the effort of defining configurations with automation.

We list numerous possibilities on this endeavour: We look into the automatic derivation of configuration profiles and their relation to similar matching scenarios. The investigation aims at finding correlations between a) certain characteristics of models and modelling languages and b) configurations or configuration subsets. Results on this research should be beneficial in determining a suitable configuration without effort. That is, if a matching scenario fulfils the above-mentioned characteristics, the matching system can be pre-configured out of the box.

Another possibility is to incorporate language-specific matchers (cf. Section 9.2.4 on custom language-specific matchers) into our matching framework, for example, UMLDiff, ECL, SiDiff etc. By using a language-specific matcher out of the box, this may spare the configuration effort for a given language-specific use case and opens up the advantage of using language semantics in the matching process to increase the efficiency of algorithms.

Furthermore, a research direction on defining suitable configurations based on self-configuring matching workflows is conceivable, such as [PBR10] seem to be a promising. This allows users to specify, for example, the matcher interaction and combination in a suitable way to optimize default configurations.

Matching Quality Apart from generally improving the current set of matchers to achieve a higher precision and/or recall, the integration of language-specific matchers is worth investigating. Hence, for a language-specific matching scenario an increase in quality is expected due to the algorithms efficiency. Further analysis is necessary, on how to leverage language-specific configurations for an increase in matching quality. This applies to others matching approaches (cf. Section 9.2 on related work) as well. An evaluation based on parallel matching of the current and newly integrated matchers could lead to valuable results.

Another direction aims at the modularization of matching algorithms into a language independent and dependent part. In this sense, a language-independent matching algorithms is enhanced with language-specific user-defined mappings, for instance, in terms of seed mappings or language-specific *match rules* as used in ECL [Kol09]. Further investigation is necessary on evaluating this hybrid approach to our approach with a set of language-independent algorithms tuned through configuration to specific use cases and to the other extreme case of language-specific algorithms.

In our evaluation, we have considered only source and target model combinations that correspond to each other as opposed to matching a certain source model to the remaining models within a repository. How the matching quality (and scalability) differs without this constraint, needs to be evaluated. Furthermore, matching models of other languages e.g. UML is part of future work.

Last, but not least further investigation is necessary on conducting traceability scenarios on the basis of traceability data gained through our matching system to see how profitable the evaluation results are.

Matching Scalability So far, only *blocking* and *filtering* have been applied to reduce the number of comparisons und thus to reduce runtime. However, more research is seen in the following areas. To allow matching of elements within the limits of a given search space (e.g., matching of neighbouring elements) to reduce the matching context as in EMF Compare [Foue], or [HA09]. Thus, the Cartesian product is not calculated sequentially, but split into phases or reduced. As mentioned above, self-configuring matching workflows, such as [PBR10] are worthwhile looking into. Regarding this direction it has to be explored, which matching configurations for matcher interaction and combination are profitable for reducing the runtime. All the above-mentioned approaches reduce runtime. Therefore, investigation into approaches additionally tackling memory consumption issues is necessary, for example, through partitioning. Thereby, the matching input is separated into parts to be processed in successive matching steps, as in

[DR07, SvKJ06, GHKR10] or structure-preserving partitioning by [Voi11] to account for graph matchers. Given the above directions for scalability, again, the dependency to matching configurations and quality needs to be studied.

Generic Traceability Interface

In this thesis, we have investigated generating trace links for model transformations. Another research direction is the extension of our work to other model operations such as model merging and model composition.

Regarding our concept on building connectors and the claim of a language-independent generic traceability interface, the set of available connectors needs to be extended with respect to other languages. Since connectors can be re-used, this promotes the applicability of our framework and spares the development for other users. Furthermore, the means to create explicit (manual) links needs to be incorporated to allow for end-to-end traceability and trace links to non-MDSD artefacts. We recommend existing tools, such as, the Trace Analyser Tool TRAMDE [Ols09], which is capable of establishing user-defined links conforming to a generic traceability metamodel.

Another research direction analyzes the boundary between transformation engine and connector with the aim to minimize efforts of the connector implementation. Therefore, it needs to be investigated, whether certain parts of the connector logic can be generally incorporated into the transformation engine. One possibility might be, for the connector logic to be taken over completely by the transformation engine and thus belonging to the implicit generation class of trace links. However, this approach should not come at the price of losing the accomplished goal of the generic traceability interface in providing a connection point for arbitrary transformation engines to overall achieve the standardization and integration of traceability data.

Experimenting with aspect-oriented programming to weave traceability-specific logic into the transformation engine led to the problem of incompatible pointcut definitions due to version mismatches between the implementation of transformation language (e.g., oAW) and aspect-oriented language (e.g., AspectJ). Hence, more stable proposals on this topic should be investigated.

Finally, we propose to extend the GTF to support more functionalities as part of a holistic lifecycle management for trace links. In this sense, administration, storage and visualization of traceability data need to be addressed. Regarding visualization, our facet-based extensibility approach is well suited for faceted browsing of traceability data as foreseen in [Hri09]. The faceted browsing applied on traceability data allows users to access (navigate) the data via multiple paths corresponding to different orderings of facets.

11

Appendix

A. Attributed Graph Definitions

A.1. Source Definitions

Definition 39 ($AG_{sourceModel}$)**.** *Let* $G_{sourceModel}$ *be an E-Graph with*

- $G_{sourceModel} = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}})$
- $V_G = \{entity, feature1, feature2\}$
- $V_D = \{Person, name, age\}$
- $E_G = \{features_1, features_2\}$
- $E_{NA} = \{ename, fname_1, fname_2\}$
- $E_{EA} = \varnothing$
- $source_G : E_G \to V_G : x \mapsto \begin{cases} entity & : \quad x = features_1 \\ entity & : \quad x = features_2 \end{cases}$
- $target_G : E_G \to V_G : x \mapsto \begin{cases} feature1 & : \quad x = features_1 \\ feature2 & : \quad x = features_2 \end{cases}$
- $source_{NA} : E_{NA} \to V_G : x \mapsto \begin{cases} entity & : \quad x = ename \\ feature1 & : \quad x = fname_1 \\ feature2 & : \quad x = fname_2 \end{cases}$

- $target_{NA} : E_{NA} \rightarrow V_D : x \mapsto \begin{cases} Person & : \quad x = ename \\ name & : \quad x = fname_1 \\ age & : \quad x = fname_2 \end{cases}$
- $source_{EA} : E_{EA} \rightarrow V_G : x \mapsto \varnothing$
- $target_{EA} : E_{EA} \rightarrow V_D : x \mapsto \varnothing$

Furthermore, let D be an algebraic signature with
$D = STRING$ as defined in Section 2.4.2, p. 29.
Then $AG_{sourceModel}$ is an attributed graph with
$AG_{sourceModel} = (G_{sourceModel}, D)$

Definition 40 ($AG_{sourceMetamodel}$)**.** *Let $G_{sourceMetamodel}$ be an E-Graph with*

- $G_{sourceMetamodel} = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}})$
- $V_G = \{Entity, Feature\}$
- $V_D = D_{string}$
- $E_G = \{features\}$
- $E_{NA} = \{ename, fname\}$
- $E_{EA} = \varnothing$
- $source_G : E_G \rightarrow V_G : x \mapsto \begin{cases} Entity & : \quad x = features \end{cases}$
- $target_G : E_G \rightarrow V_G : x \mapsto \begin{cases} Feature & : \quad x = features \end{cases}$
- $source_{NA} : E_{NA} \rightarrow V_G : x \mapsto \begin{cases} Entity & : \quad x = ename \\ Feature & : \quad x = fname \end{cases}$
- $target_{NA} : E_{NA} \rightarrow V_D : x \mapsto \begin{cases} string & : \quad x = ename \\ string & : \quad x = fname \end{cases}$
- $source_{EA} : E_{EA} \rightarrow V_G : x \mapsto \varnothing$
- $target_{EA} : E_{EA} \rightarrow V_D : x \mapsto \varnothing.$

Furthermore, let D be an algebraic signature with
$D = STRING$ as defined in Section 2.4.2, p. 29.
Then $AG_{sourceMetamodel}$ is an attributed graph with
$AG_{sourceMetamodel} = (G_{sourceMetamodel}, D)$

Definition 41 ($ATG_{sourceMetamodel}$)**.** *Let Z be the final D-algebra.*
Then $ATG_{sourceMetamodel}$ is an attributed type graph with
$ATG_{sourceMetamodel} = (AG_{sourceMetamodel}, Z)$.

Definition 42 ($TAG_{sourceModel}$)**.** *Let t be a graph morphism*
$t : AG_{sourceModel} \rightarrow ATG_{sourceMetamodel}$ with

- $t = (t_{G,V_G}, t_{G,V_D}, t_{G,E_G}, t_{G,E_{NA}})$

- $t_{G,V_G}(entity) = Entity$,
- $t_{G,V_G}(feature_1) = t_{G,V_G}(feature_2) = Feature$,
- $t_{G,V_D}(Person) = t_{G,V_D}(name) = t_{G,V_D}(age) = string$,
- $t_{G,E_G}(features_1) = t_{G,E_G}(features_2) = features$
- $t_{G,E_{NA}}(ename) = ename$
- $t_{G,E_{NA}}(fname_1) = t_{G,E_{NA}}(fname_2) = fname$

Then $TAG_{sourceModel}$ is a typed attribute graph over $ATG_{sourceMetamodel}$ with $TAG_{sourceModel} = (AG_{sourceModel}, t)$.

A.2. Target Definitions

Definition 43 ($AG_{targetModel}$)**.** *Let $G_{targetModel}$ be an E-Graph with*

- $G_{targetModel} = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j\in\{G,NA,EA\}})$
- $V_G = \{class, field_1, field_2, method_1, method_2\}$
- $V_D = \{Person, name, age, getName, getAge\}$
- $E_G = \{fields_1, fields_2, method_1, method_2\}$
- $E_{NA} = \{cname, fname_1, fname_2, mname_1, mname_2\}$
- $E_{EA} = \varnothing$
- $source_G : E_G \to V_G : x \mapsto \left\{ \begin{array}{rcl} class & : & x \in \{fields_1, fields_2, methods_1, methods_2\} \end{array} \right.$
- $target_G : E_G \to V_G : x \mapsto \left\{ \begin{array}{rcl} field_1 & : & x = fields_1 \\ field_2 & : & x = fields_2 \\ method_1 & : & x = methods_1 \\ method_2 & : & x = methods_2 \end{array} \right.$
- $source_{NA} : E_G \to V_G : x \mapsto \left\{ \begin{array}{rcl} class & : & x = cname \\ field_1 & : & x = fname_1 \\ field_2 & : & x = fname_2 \\ method_1 & : & x = mname_1 \\ method_2 & : & x = mname_2 \end{array} \right.$
- $target_{NA} : E_G \to V_G : x \mapsto \left\{ \begin{array}{rcl} Person & : & x = cname \\ name & : & x = fname_1 \\ age & : & x = fname_2 \\ getName & : & x = mname_1 \\ getAge & : & x = mname_2 \end{array} \right.$
- $source_{EA} : E_G \to V_G : x \mapsto \varnothing$
- $target_{EA} : E_G \to V_G : x \mapsto \varnothing$

Furthermore, let D be an algebraic signature with
$D = STRING$ as defined in Section 2.4.2, p. 29.
Then $AG_{targetMetamodel}$ is an attributed graph with
$AG_{targetMetamodel} = (G_{targetMetamodel}, D)$

Definition 44 ($AG_{targetMetamodel}$). *Let $G_{targetMetamodel}$ be an E-Graph with*

- $G_{targetMetamodel} = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j\in\{G,NA,EA\}})$
- $V_G = \{Class, Field, Method\}$
- $V_D = D_{string}$
- $E_G = \{fields, methods\}$
- $E_{NA} = \{cname, fname, mname\}$
- $E_{EA} = \varnothing$
- $source_G : E_G \to V_G : x \mapsto \begin{cases} Class & : \quad x = fields \\ Class & : \quad x = methods \end{cases}$
- $target_G : E_G \to V_G : x \mapsto \begin{cases} Field & : \quad x = fields \\ Method & : \quad x = methods \end{cases}$
- $source_{NA} : E_G \to V_G : x \mapsto \begin{cases} Class & : \quad x = cname \\ Field & : \quad x = fname \\ Method & : \quad x = mname \end{cases}$
- $target_{NA} : E_G \to V_G : x \mapsto \begin{cases} string & : \quad x = cname \\ string & : \quad x = fname \\ string & : \quad x = mname \end{cases}$
- $source_{EA} : E_G \to V_G : x \mapsto \varnothing$
- $target_{EA} : E_G \to V_G : x \mapsto \varnothing$

Furthermore, let D be an algebraic signature with
$D = STRING$ as defined in Section 2.4.2, p. 29.
Then $AG_{targetMetamodel}$ is an attributed graph with
$AG_{targetMetamodel} = (G_{targetMetamodel}, D)$

Definition 45 ($ATG_{targetMetamodel}$). *Let Z be the final D-algebra.*
Then $ATG_{targetMetamodel}$ is an attributed type graph with
$ATG_{targetMetamodel} = (AG_{targetMetamodel}, Z)$.

Definition 46 ($TAG_{targetModel}$). *Let t be a graph morphism*
$t : AG_{targetModel} \to ATG_{targetMetamodel}$ with

- $t = (t_{G,V_G}, t_{G,V_D}, t_{G,E_G}, t_{G,E_{NA}})$
- $t_{TG,V_G}(class) = Class$
- $t_{TG,V_G}(field_1) = t_{TG,V_G}(field_2) = Field$

- $t_{TG,V_G}(method_1) = t_{TG,V_G}(method_2) = Method$
- $t_{TG,V_D}(Person) = t_{TG,V_D}(name) = t_{TG,V_D}(age) = t_{TG,V_D}(getName)$
  $= t_{TG,V_D}(getAge) = string$
- $t_{G,E_G}(fields_1) = t_{G,E_G}(fields_2) = fields$
- $t_{G,E_G}(methods_1) = t_{G,E_G}(methods_2) = methods$
- $t_{G,E_{NA}}(cname) = cname$
- $t_{G,E_{NA}}(fname_1) = t_{G,E_{NA}}(fname_2) = fname$
- $t_{G,E_{NA}}(mname_1) = t_{G,E_{NA}}(mname_2) = mname$

Then $TAG_{targetModel}$ *is a typed attribute graph over* $ATG_{targetMetamodel}$ *with* $TAG_{targetModel} = (AG_{targetModel}, t)$*.*

B. Matchbox

Matchbox is a metamodel matching framework based on the combination of different matching algorithms [Voi09, VIR10]. This framework is build up on the *SAP Auto Mapping Core*, an implementation inspired by COMA++ [DR07], a schema matching framework.

Matchbox is a parallel matching system and follows the classical process steps from Section 2.3 (cf. Figure 2.3). In the following sections, we provide an overview on the internal data model and matching algorithms of Matchbox as well as the possibilities on aggregation of the matcher results to create an output mapping.

B.1. Model Import

The internal data model of Matchbox is a typed graph called Genie. A comprehensive description is given in [Iva10]. To be matched metamodels are transformed into this internal data model. A model element is reflected as an *entity* with an unique resource identifier (amongst other properties) to the original model element of representation. Additonally, the entity has a name, which is set to the metamodel elements name. Relationships between model elements are defined through containment, association or inheritance. Important for the scope of this work, the instance-of relationship between model elements (instances) and metamodel elements can be expressed. Instances are characterized through their metamodel type and captures its *value* as a list of strings. Moreover, attributes are assigned to entities and feature a primitive type, e.g., Integer, Float, String etc.

B.2. Matching Algorithms

Matchbox supplies a range of different matchers based on element-level algorithms (*name matcher*, *name path matcher* and *data type matcher*) as well as structure-level algorithms (*parent matcher*, *children matcher*, *sibling matcher*, *leaf matcher*, *graph edit distance matcher* and *pattern matcher*).

Element-level Algorithms

The **Name Matcher** compares the names of source and target-metamodel elements and calulates similarities based on *Trigrams*. Generally, an *n*-gram is a contiguous sequence of *n* tokens from a given sequence of a string. The Trigram approach calculates equal character sequences of size three (trigram) in the names. The sum of all occurances of trigrams is then compared to the overall number of trigrams. Furthermore, the user may specify synonyms, that are stored in a dictionary and support the calculation of similarities. Alternatively, a string edit distance based on Levenshtein [WF74] can be used.

The **Name Path Matcher** is an extension of the Name Matcher and calculates the similarities of metamodel elements with the help of their path (containment path of an element). A path is the concatenation of the element names from the root to the current element, called name path. The name matcher is applied on the name paths of the metamodel elements.

The **Data Type Matcher** makes use of a transformation table that provides the degree of compatibility between types of metamodel elements. Therefore, the similarity is based on the type of metamodel elements.

Structure-level Algorithms

The **Parent Matcher** calculates similarities on the rationale that elements with similar parents indicate a similarity of the contained elements. That is, the parent matcher computes the similarity of a source and target element by applying a specific matcher (e.g., the name matcher) to the parents of source and target element. Afterwards, this calculated similarity is propagated to its containing elements.

The **Children Matcher** is based on the inverse rationale of the parent matcher and assumes that similar model elements have similar children. That is, the similarity between contained elements of source and target are computed and propagated to the parents of the source and target. Matchbox uses the leaf matcher for an initial calculation of similarities for the set of children elements of source and target. This resulting matrix of similarities is then combined by using the average of all values.

The **Sibling Matcher** follows an approach similar to the children matcher. The matcher works on the assumption that similar model elements have similar siblings. In Matchbox, the calculation of similarity values for the siblings is calculated through the leaf matcher. The resulting similarites are combined the same way as for the children matcher. Regarding the children and sibling matcher, other matchers than the leaf matcher could be used to calculate the initial similarity, however the leaf matcher provides the best results [Voi11].

The **Leaf Matcher** calculates similarities based on similar leaves. That is, the similarities of the set of all leaves (elements with no children) to a given source and target element are calculated. This is achieved through the name matcher. Again, the resulting matrix of similarities is aggregated by using the average of all values. This resulting similarity is propagated to the currently processed source and target element. Finally, these aggegated results are aggregated again by computing their average result. The leaf matcher is used to detect structural re-organisations.

The **Graph Edit Distance Matcher** applies a sequence of edit operations, composed of: add, remove, and relabel (rename). A sequence of such operations defines a mapping from one graph onto another, thus calculating the maximal common subgraph along with the necessary operations.

The **Pattern Matcher** computes a complex structural similarity between model elements based on existing patterns. Patterns indicate redundant information or the usage of established design structures.

B.3. Configuration of Similarity Cube

Each matcher produces an output in the form of a matrix with cells holding the similarity values. In doing so, the matcher calculates a similarity value for each pair of the cartesian product between the set of all elements of a certain source and target model. The similarity values are from the interval of real numbers between 0 and 1 denoted by $[0 \ldots 1]$. All similarity matrices are arranged along the set of matcher types to form a similarity value cube. To derive a mapping between source and target elements out of these results, the similarity value cube needs to be configured. This configuration is dependent on the following strategies and parameters. The strategies are *Aggregation*, *Selection*, *Direction*. Furthermore, these strategies may be combined.

- **Aggregation**: The aggregation reduces the similarity cube to a matrix, by aggregating all matcher's resulting matrices into one. In Matchbox, four different strategies for aggregation are provided: *Max*, *Min*, *Average* and *Weighted*. The *Max* strategy follows an optimistic mode and selects the highest similarity value calculated by a matcher. On the contrary, the *Min* strategy selects the lowest value. *Average* evens out the matcher's results by calculating the average for each source and target element. Through user-defined weights on certain matcher results, accordingly, the *Weighted* aggregation computes the weighted sum of the results w.r.t. each source and target element.
- **Selection**: The selection filters possible matches from the *aggregated matrix* according to a defined strategy. Possible strategies are *Threshold*, *selN* and *selDelta*: *Threshold* selects all matches exceeding a particular threshold. *selN* returns the highest N similarity values compared to all matches. Finally, *selDelta* firstly chooses, the match with the highest similarity value; secondly returns those similarity values within a certain relative range. This range is defined through the specified value for delta, that is, through the interval $[Max - Max \cdot Delta, Max]$. Furthermore, the above-mentioned selection strategies can be combined, such as Threshold and selDelta.
- **Direction**: The direction is dedicated to a ranking of matched model elements according to their similarity. In the *Forward* strategy elements of the source are selected for each element of the target model. The *Backward* strategy is the inverse of the Forward strategy. The above strategies can be combined in the *Both* strategy, where a match then has to have a similarity value for each direction forward and backward.

C. Case Study: Sales Scenario

This section is dedicated to the description of the Sales Scenario. The Sales Scenario is a model-driven software product line, demonstrating SAP's core business regarding application engineering in the domain of enterprise software. From exemplary solutions centered around this domain including Product Life Cycle Management, Supply Chain Management or Supplier Relationship Management, the Sales Scenario mainly focuses on Customer Relationship Management in combination with the aforementioned domains. The main purpose of this case study is the holistic management of business data, including central storage and access controlled retrieval in accordance with the official guide book to SAP CRM of Buck-Emden and Zencke [BEZ04].

The engineering side of the Sales Scenario is based on feature-oriented product line engineering [KLD02]. Consequently, the end user has to configure a feature tree in order to select a valid set of features for a resulting application. The required mapping to achieve a customer-specific application is twofold, that is, two types of model transformations take place: a) transformations from the configured feature tree to a set of domain-specific models, which require an explicit mapping from problem to solution space and b) model-to-text transformations essentially building the executable business application.

For the scope of our work, we make use of the second transformation type, while assuming a feature configuration, where every feature is selected. The features of the sales scenario entail account management, quotation management, order management and product management. The the input models of the model-to-text transformations are instances of six domain specific languages (DSLs), describing different domains of the business application. In particular, these are:

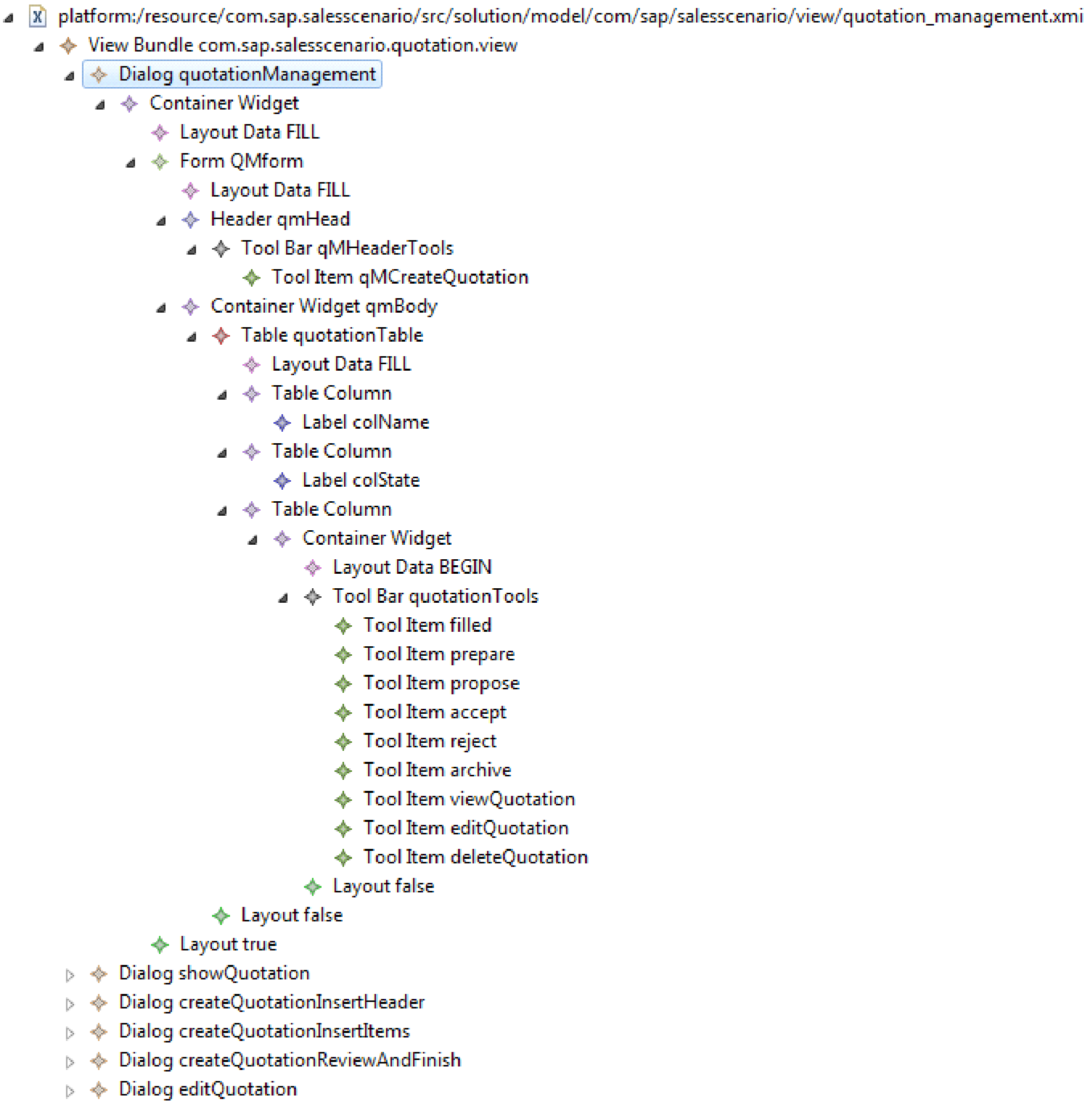
- **Action DSL** to declare invokable behaviour
- **Business Object DSL** for data structure modelling
- **Context DSL** for session contexts to buffer in-memory data models in addressable spaces
- **State DSL** for transitions based on state machines
- **Dialog DSL** for page flow definitions
- **View DSL** for graphical user interfaces

For more information about the nature of the above-mentioned DSLs, we refer to [LG08, EJF$^+$09]. To specify the application features, domain-specific models per DSL are defined for each application feature.

D. Evaluation: XPand Connector

D.1. Data Models and Transformation Program

From the source model in Figure D.1, two Java classes are generated *QuotationManagementDialog* (see Listing 11.1) and *QuotationManagementComposite* (see Listing 11.2).



- platform:/resource/com.sap.salesscenario/src/solution/model/com/sap/salesscenario/view/quotation_management.xmi
  - View Bundle com.sap.salesscenario.quotation.view
    - Dialog quotationManagement
      - Container Widget
        - Layout Data FILL
        - Form QMform
          - Layout Data FILL
          - Header qmHead
            - Tool Bar qMHeaderTools
              - Tool Item qMCreateQuotation
          - Container Widget qmBody
            - Table quotationTable
              - Layout Data FILL
              - Table Column
                - Label colName
              - Table Column
                - Label colState
              - Table Column
                - Container Widget
                  - Layout Data BEGIN
                  - Tool Bar quotationTools
                    - Tool Item filled
                    - Tool Item prepare
                    - Tool Item propose
                    - Tool Item accept
                    - Tool Item reject
                    - Tool Item archive
                    - Tool Item viewQuotation
                    - Tool Item editQuotation
                    - Tool Item deleteQuotation
                  - Layout false
            - Layout false
        - Layout true
    - Dialog showQuotation
    - Dialog createQuotationInsertHeader
    - Dialog createQuotationInsertItems
    - Dialog createQuotationReviewAndFinish
    - Dialog editQuotation

Figure D.1.: Quotation Management Source Model

The latter classes relate to a container widget with a tool bar including tool items. In the generated code, a ToolItem to prepare a quotation is added to the container widget. The ToolItem is labelled with the tool tip text ”Propose quotation”. The transformation program for QuotationManagementDialog and QuotationManagementComposite is presented in Listing 11.3.

```
package com.sap.salesscenario.quotation.view;

public class QuotationManagementDialog
    extends
      Dialog<QuotationManagementComposite> {

  public QuotationManagementDialog(Display display) {
    super(display);
  }

  @Override
  protected int getHeight() {
    return 250;
  }

  @Override
  protected void createControl() {
    this.control = new QuotationManagementComposite(this, this.shell,
        SWT.NONE);

    this.shell.setText("Quotation Management");
  }

  protected void shellDisposed(/*Transition closingTransition*/) {
    System.out.println("Closing QuotationManagementComposite shell.");
  }

  @Override
  public void update() {
    // implement your UI updates here
  }
}
```

Listing 11.1: Generated Target: QuotationManagementDialog.java

```
package com.sap.salesscenario.quotation.view;

public class QuotationManagementComposite extends Composite {

  private boolean isWizard = false;

  private int maxHeight = 0;

  private final QuotationManagementDialog dialog;

  private FormToolkit toolkit = null;

  public QuotationManagementComposite(QuotationManagementDialog dialog,
    Composite parent, int style) {
    super(parent, style);
    this.dialog = dialog;
    toolkit = new FormToolkit(parent.getDisplay());
    this.initialize();
  }

  public QuotationManagementDialog getDialog() {
    return dialog;
  }

  public TableLine<?> tableQuotationTableCreateLine() {
    TableItem item = new TableItem(tableQuotationTable, SWT.NONE);
    final TableLine<?> result = this
        .tableQuotationTableCreateTableLine(item);

    item.setData(result);

    final TableEditor tableColumn7826d729_0a89_4b89_9a88_b99052992e8d = new
        TableEditor(
        tableQuotationTable);

    Composite containerWidget2d85dcb9_7248_46f1_b4e5_c106b6077ae1 = new
        Composite(
        tableQuotationTable, SWT.NONE);

    ToolBar toolBarQuotationTools = new ToolBar(
        containerWidget2d85dcb9_7248_46f1_b4e5_c106b6077ae1, SWT.FLAT);

    ToolItem toolItemPropose = new ToolItem(toolBarQuotationTools,
        SWT.HORIZONTAL);
    toolItemPropose.setToolTipText("Propose quotation");
    toolItemPropose.setData(toolBarQuotationTools);

    return result;
  }
}
```

Listing 11.2: Generated Target: QuotationManagementComposite.java

```
<<REM>> --------------------- generateDialogs: ---------------------- <<ENDREM
  >>
<<REM>> Dialog classes for dialogs                                <<ENDREM>>
```

```
<<REM>> -------------------------------------------------------------- <<ENDREM>>
<<DEFINE swtFile(ViewBundle bundle) FOR Dialog>>
<<FILE getFileForPackage(bundle.id, this.id.toFirstUpper() + "Dialog")>>
package <<bundle.id->>;

public class <<this.id.toFirstUpper()->>Dialog extends Dialog<<<this.id.toFirstUpper()->>Composite> {

  public <<this.id.toFirstUpper()->>Dialog(Display display) {
    super(display);
  }

  <<EXPAND getSize FOR this->>
    <<DEFINE getSize FOR Dialog>>
      <<IF this.width >= 0->>
      @Override
      protected int getWidth() {
        return <<this.width->>;
      }
      <<ENDIF->>

      <<IF this.height >= 0->>
      @Override
      protected int getHeight() {
        return <<this.height->>;
      }
      <<ENDIF->>
  <<ENDDEFINE>>

  @Override
  protected void createControl() {
    <<EXPAND swtElement(bundle, this) FOR this.content->>
    this.shell.setText("<<this.title>>");
  }

  protected void shellDisposed(/*Transition closingTransition*/) {
    System.out.println("Closing <<getCompositeName(this)->> shell.");
  }

  @Override
  public void update() {
    // implement your UI updates here
  }
}
<<ENDFILE>>
<<ENDDEFINE>>

```

```
<<REM>> --------------------- generateComposites: -------------------- <<ENDREM>>
<<REM>> Composite classes for the ContainerWidget child of a dialog <<ENDREM>>
<<REM>> -------------------------------------------------------------- <<ENDREM>>
<<DEFINE swtElement(ViewBundle bundle, Dialog dialog) FOR ContainerWidget->>
  this.control = new <<getCompositeName(dialog)->>(this, this.shell, SWT.NONE);
<<FILE getFileForPackage(bundle.id, getCompositeName(dialog))->>

package <<bundle.id->>;

public class <<getCompositeName(dialog)->> extends Composite {

  private boolean isWizard = false;

  private int maxHeight = 0;

  private final <<dialog.id.toFirstUpper()->>Dialog dialog;

  private FormToolkit toolkit = null;

  public <<getCompositeName(dialog)->>(<<dialog.id.toFirstUpper()->>Dialog dialog, Composite parent, int style) {
    super(parent, style);
    this.dialog = dialog;
    toolkit = new FormToolkit( parent.getDisplay() );
    this.initialize();
  }

  public <<dialog.id.toFirstUpper()->>Dialog getDialog() {
    return dialog;
  }

  <<EXPAND TableGenerator FOR this>>
  }
}
<<ENDFILE>>
<<ENDDEFINE>>

<<DEFINE TableGenerator FOR Table>>
    public TableLine<?> <<getWidgetName(this)->>CreateLine() {

    TableItem item = new TableItem(<<getWidgetName(this)->>, SWT.NONE);
    final TableLine<?> result = this.<<getWidgetName(this)->>CreateTableLine(item);
    item.setData(result);

    <<FOREACH this.columns AS column ITERATOR it->>
    <<EXPAND tableEditor(getWidgetName(this), it.counter0) FOR column->>
    <<ENDFOREACH>>

    return result;
    }
<<ENDDEFINE>>

<<DEFINE tableEditor(String swtParent, Integer index) FOR TableColumn>>
    final TableEditor <<getWidgetName(this)->> = new TableEditor(<<swtParent->>)
        ;

    <<EXPAND localSWTElement(swtParent, true) FOR this.itemDefinition->>

    <<getWidgetName(this.itemDefinition)->>.pack();
    <<getWidgetName(this.itemDefinition)->>.setData(
      TableHelper.TABLE_LINE, result);
    result.setEditor(<<index->>, <<getWidgetName(this.itemDefinition)->>, <<
        getWidgetName(this)->>);
<<ENDDEFINE>>

<<DEFINE localSWTElement(String swtParent, boolean inTable) FOR ContainerWidget
    >>
    <<EXPAND swtElement(swtParent, inTable) FOR this->>
<<ENDDEFINE>>

<<DEFINE swtElement(String swtParent, boolean inTable) FOR ContainerWidget>>
    Composite <<getWidgetName(this)->> = new Composite(<<swtParent>>, SWT.NONE);
    <<EXPAND swtElement(getWidgetName(this), inTable) FOREACH this.widgets->>
<<ENDDEFINE>>

<<DEFINE localSWTElement(String swtParent, boolean inTable) FOR ToolBar>>
     <<EXPAND swtElement(swtParent, inTable) FOR this->>
<<ENDDEFINE>>
<<DEFINE swtElement(String swtParent, boolean inTable) FOR ToolBar>>
    ToolBar <<getWidgetName(this)->> = new ToolBar(<<swtParent->>, SWT.FLAT);
  <<EXPAND localSWTElement(getWidgetName(this), inTable) FOREACH this.items->>
<<ENDDEFINE>>

<<DEFINE localSWTElement(String swtParent, boolean inTable) FOR ToolItem>>
    ToolItem <<EXPAND swtElement(swtParent, inTable) FOR this->>
<<ENDDEFINE>>
<<DEFINE swtElement(String swtParent, boolean inTable) FOR ToolItem>>
    <<getWidgetName(this)->> = new ToolItem(<<swtParent->>, SWT.HORIZONTAL);
    <<getWidgetName(this)->>.setToolTipText("<<this.toolTipText->>");
    <<getWidgetName(this)->>.setData(<<swtParent->>);
<<ENDDEFINE>>
```

Listing 11.3: Transformation Program for QuotationManagementDialog and QuotationManagementComposite

D.2. Gold Mapping

In this section, the reference traceability data, so-called gold mapping, is documented. First, we present the gold mapping for the generation of **QuotationManagementDialog**.java in Table 11.1 and Table 11.2, followed by the gold mapping for **QuotationManagementComposite**.java in Table 11.3 to Table 11.5. We omitted containment links for the sake of clarity. Secondly, static text snippets generated between expressions (marked by guillemets) are concatenated to a unit of traceability data (one row in the table), rather than listing single text snippets in the processed string.

Source Artefact	**Operator**	**Target Artefact**	**Trace Link**
template snippet: public class (Line 8)	static text: public class (Line 8)	public class (Line 3)	Create
Dialog quotationManagement.id	«this.id.toFirstUpper()-» (Line 8)	QuotationManagement (Line 3)	Create
quotationManagement.id	query: toFirstUpper() (Line 8)	template snippet: toFirstUpper() (Line 3)	Retrieve
template snippet: Dialog extends Dialog< (Line 8)	static text: Dialog extends Dialog< (Line 8)	Dialog extends Dialog< (Line 3-5)	Create
Dialog quotationManagement.id	«this.id.toFirstUpper()-» (Line 8)	QuotationManagement (Line 5)	Create
quotationManagement.id	query: toFirstUpper() (Line 8)	template snippet: query: toFirstUpper() (Line 5)	Retrieve
template snippet: Composite>{ (Line 8)	static text: Composite>{ (Line 8)	Composite>{ (Line 5)	Create
template snippet: public (Line 10)	static text: public (Line 10)	public (Line 7)	Create
Dialog quotationManagement.id	«this.id.toFirstUpper()-» (Line 10)	QuotationManagement (Line 7)	Create
Dialog quotationManagement.id	query: toFirstUpper() (Line 10)	template snippet: toFirstUpper() (Line 7)	Retrieve
template snippet: Dialog(Display display) { (Line 10)	static text: Dialog(Display display) { (Line 10)	Dialog(Display display) { (Line 7)	Create
template snippet: super(display); (Line 11)	static text: super(display); (Line 11)	super(display); (Line 8)	Create
template snippet: { (Line 12)	{ (Line 12)	{ (Line 9)	Create

Table 11.1.: Gold Mapping for QuotationManagementDialog: Part 1/2

Source Artefact	**Operator**	**Target Artefact**	**Trace Link**
Dialog quotationManagement	EXPAND Block (Line 14)	text block (Line 11-14)	Create
Dialog quotationManagement	DEFINE Block (Line 15-29)	text block (Line 11-14)	Create
Dialog quotationManagement.height	IF Block (Line 23-28)	text block (Line 11-14)	Create
template snippet: @Override protected int getHeight() { return (Line 24-26)	static text: @Override protected int getHeight() { return (Line 24-26)	@Override protected int getHeight() { return (Line 11-13)	Create
Dialog quotationManagement.height	«this.height -» (Line 26)	250 (Line 13)	Create
template snippet: ;} (Line 26-27)	static text: ;} (Line 26-27)	;} (Line 13-14)	Create
template snippet: @Override protected void createControl() { (Line 31-32)	static text: @Override protected void createControl() { (Line 31-32)	@Override protected void createControl() { (Line 16-17)	Create
Container Widget	EXPAND Block (Line 33)	text block (Line 18-19)	Create
Container Widget	DEFINE Block (Line 52-53)	text block (Line 18-19)	Create
template snippet: this.shell.setText(” (Line 34)	static text: this.shell.setText(” (Line 34)	this.shell.setText(” (Line 21)	Create
Dialog quotationManagement.title	«this.title» (Line 34)	Quotation Management (Line 21)	Create
template snippet: ”); } (Line 34-35)	static text: ”); } (Line 34-35)	”); } (Line 21-22)	Create
template snippet: protected void [...] println(”Closing (Line 37-38)	static text: protected void [...] println(”Closing (Line 37-38)	protected void [...] println(”Closing (Line 24-25)	Create

Table 11.2.: Gold Mapping for QuotationManagementDialog: Part 2/2

Source Artefact	Operator	Target Artefact	Trace Link
Dialog quotationManagement.id	«getCompositeName(this)-» (Line 38)	QuotationMComposite (Line 25)	Create
Dialog quotationManagement.id	query: getCompositeName(this) (Line 38)	template snippet: getCompositeName(this) (Line 25)	Retrieve
template snippet: shell.”); } (Line 38-39)	static text: shell.”); } (Line 38-39)	shell.”); } (Line 25-26)	Create
template snippet: @Override [...] } (Line 41-45)	@Override [...] } (Line 41-45)	@Override [...] } (Line 28-32)	Create
Dialog quotationManagement.id	FILE Block (Line 54)	QuotationMComposite.java (Line 1-48)	Create
template snippet: package (Line 56)	static text: package (Line 56)	package (Line 1)	Create
View Bundle.id	«bundle.id-» (Line 56)	com.sap.salesscenario.quotation.view (Line 1)	Create
template snippet: public class (Line 58)	static text: public class (Line 58)	public class (Line 3)	Create
Dialog quotationManagement.id	«getCompositeName(dialog)-» (Line 58)	QuotationMComposite (Line 3)	Create
Dialog quotationManagement.id	query: getCompositeName(dialog) (Line 58)	template snippet: getCompositeName(dialog) (Line 3)	Retrieve
template snippet: extends Composite { [...] private final (Line 58-64)	static text: extends Composite { [...] private final (Line 58-64)	extends Composite { [...] private final (Line 3-9)	Create
Dialog quotationManagement.id	«dialog.id.toFirstUpper()-» (Line 64)	QuotationManagement (Line 9)	Create

Table 11.3.: Gold Mapping for QuotationManagementComposite: Part 1/3

Source Artefact	**Operator**	**Target Artefact**	**Trace Link**
Dialog quotationManagement.id	query: toFirstUpper() (Line 64)	template snippet: toFirstUpper() (Line 64)	Retrieve
template snippet: Dialog dialog; [...] public (Line 64-68)	static text: Dialog dialog; [...] public (Line 64-68)	Dialog dialog; [...] public (Line 9)	Create
Dialog quotationManagement.id	«getCompositeName(dialog)-» (Line 68)	QuotationManagementComposite (Line 13)	Create
Dialog quotationManagement.id	query: getCompositeName(dialog) (Line 68)	template snippet: getCompositeName(dialog) (Line 68)	Retrieve
template snippet: { (Line 68)	static text: { (Line 68)	{ (Line 13)	Create
Dialog quotationManagement.id	«dialog.id.toFirstUpper()-» (Line 68)	QuotationManagement (Line 13)	Create
template snippet: Dialog dialog; [...] public (Line 68-75)	Dialog dialog; [...] public (Line 68-75)	Dialog dialog; [...] public (Line 13-21)	Create
Dialog quotationManagement.id	«dialog.id.toFirstUpper()-» (Line 75)	QuotationManagement (Line 21)	Create
template snippet: Dialog getDialog() [...] }	Dialog getDialog() [...] } (Line 75-77)	Dialog getDialog() [...] } (Line 21-23)	Create
Dialog quotationManagement	«EXPAND TableGenerator FOR this» (Line 79)	text block (Line 25-48)	Create
Table quotationTable	«DEFINE TableGenerator FOR this» (Line 85-89)	text block (Line 25-48)	Create
template snippet: public TableLine<?> (Line 86)	public TableLine<?> (Line 86)	public TableLine<?> (Line 25)	Create

Table 11.4.: Gold Mapping for QuotationManagementComposite: Part 2/3

Source Artefact	**Operator**	**Target Artefact**	**Trace Link**
Table quotationTable	«getWidgetName(this)-» (Line 86)	tableQuotationTable (Line 25)	Create
Table quotationTable	query: getWidgetName(this) (Line 86)	template snippet: getWidgetName(this) (Line 86)	Retrieve
template snippet: CreateLine() { [...] TableItem((Line 68-88)	static text: CreateLine() { [...] TableItem((Line 68-88)	CreateLine() { [...] TableItem((Line 25-26)	Create
Table quotationTable	«getWidgetName(this)-» (Line 88)	tableQuotationTable (Line 26)	Create
template snippet: , SWT.NONE) [...] this. (Line 88-89)	static text: , SWT.NONE) [...] this. (Line 88-89)	, SWT.NONE) [...] this. (Line 26-27)	Create
Table quotationTable	«getWidgetName(this)-» (Line 89)	tableQuotationTable (Line 28)	Create
template snippet: CreateTableLine [...] (result); (Line 89-90)	static text: CreateTableLine [...] (result); (Line 89-90)	CreateTableLine [...] (result); (Line 28-30)	Create
Table Column (3) (1-2 omitted)	FOR EACH Block (Line 92-94)	text block (Line 32-44)	Create
Table Column (3)	«EXPAND tableEditor(getWidgetName(this), it.counter0) FOR column-» (Line 93)	text block (Line 32-44)	Create
Table Column (3)	query: getWidgetName(this) (Line 93)	template snippet: getWidgetName(this) (Line 93)	Retrieve

Table 11.5.: Gold Mapping for QuotationManagementComposite: Part 3/3

E. Evaluation: QVT Connector

E.1. Data Models

The QVTO UML2RDB transformation is based on a simplified UML and relational database model. The transformation specification is discussed in [Obj11].

Example models and the transformation program can be found at:
http://wiki.eclipse.org/QVTo#Examples

E.2. Gold Mapping

The gold mapping is represented in Table 11.6.

Source Artefact	**Target Artefact**	**Trace Link**
Model ”model”	Model ”model”	Create
Package ”com”	Schema ”class”	Create
Package ”borland”	Schema ”class”	Create
Package ”tg”	Schema ”class”	Create
Package ”mda”	Schema ”class”	Create
Package ”sample”	Schema ”class”	Create
Package ”domain”	Schema ”class”	Create
Package ”class””	Schema ”class”	Create
Class ”Call” [persistent]	Table ”Call”	Create
Class ”Call”	Call.TableColumn ”callId”	Create
Class ”Call”	Call.TableColumn ”callShortDescription”	Create
Class ”Call”	Call.TableColumn ”callTimestamp”	Create
Class ”Call”	Call.TableColumn ”callResolved”	Create
Class ”Call”	Call.TableColumn ”callDescription”	Create
Class ”Call”	Call.TableColumn ”callSeverity”	Create
Call.Property ”callId”	Call.TableColumn ”callId”	Create
Call.Property ”callShortDescription”	Call.TableColumn ”callShortDescription”	Create
Call.Property ”callSeverity”	Call.TableColumn ”callSeverity”	Create
Call.Property ”callTimestamp”	Call.TableColumn ”callTimestamp”	Create
Call.Property ”callResolved”	Call.TableColumn ”callResolved”	Create
Call.Property ”callDescription”	Call.TableColumn ”callDescription”	Create
Call.Property ”customer”	Call.TableColumn ”customer_customerId”	Create
Call.Property ”callId” [isPrimaryKey]	Call.PrimaryKey ”PKCall”	Create
Call.Property ”customer” [IsForeignKey]	Call.ForeignKey ”FKcustomer”	Create
Class ”Abstract”	Call.TableColumn ”common”	Create
Abstract.Property ”common”	Call.TableColumn ”common”	Create

Table 11.6.: Gold Mapping compared to Traceability Data of Uml2Rdb Scenario

List of Figures

List of Tables

Bibliography

[AB93] Robert S. Arnold and Shawn A. Bohner. Impact Analysis - Towards a Framework for Comparison. In *Proceedings of the Conference on Software Maintenance (ICSM '93)*. IEEE Computer Society, 1993.

[ABE$^+$06] G. Antoniol, B. Berenbach, A. Eyged, S. Ferguson, J. Maletic, and A. Zisman. Problem Statement and Grand Challenges. Center of Excellence for Traceability, 2006.

[Ade12] J. Adersberger. *Modellbasierte Extraktion, Repräsentation und Analyse von Traceability-Informationen.* PhD thesis, Universitiy of Erlangen-Nürnberg, 2012.

[AGG$^+$08] N. Anquetil, B. Grammel, I. Galvao, J. Noopen, S. S. Khan, H. Arboleda, A. Rashid, and A. Garcia. Traceability for Model-driven, Software Product Line Engineering. In *Proceedings of the ECMDA Traceability Workshop*, 2008.

[Ale02] I. Alexander. Toward Automatic Traceability in Industrial Practice. In *Proceedings of the International Workshop on Traceability in Emerging Forms of Software Engineering*, 2002.

[AP03] M. Alanen and I. Porres. Difference and Union of Models. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863. Springer-Verlag, 2003.

[ARNRSG06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model Traceability. *IBM Systems Journal*, volume 45, No 3, 2006.

[Aßm03] U. Aßmann. *Invasive Software Composition.* Springer, 2003.

[BCW12] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 2012.

[BEZ04] R. Buck-Emden and P. Zencke. *mySAP CRM: The Official Guidebook to SAP CRM 4.0.* Galileo Press, 2004.

[BLP00] P. A. Bernstein, A. Y. Levy, and R. A. Pottinger. A Vision for Management of Complex Models. In *SIGMOD Record*, volume 29, 2000.

[BMPQ04] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength Schema Matching. In *SIGMOD Record*, volume 33, 2004.

[CH06] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, Vol 45, No 3, 2006.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13:377–387, 1970.

[Con] AMPLE Consortium. AMPLE Research Project. `http://ample.holos.pt/`.

[CSH06] N. Choi, I.Y. Song, and H. Han. A Survey of Ontology Mappings. In *SIGMOD Record*, volume 35, pages 34–41, 2006.

[DKPF08] N. Drivalos, D. S. Kolovos, R. Paige, and K. Fernandes. Engineering a Domain-Specific Language for Software Traceability. In *Proceedings of the International Conference on Software Language Engineering (SLE'08)*, 2008.

[Do06] H. H. Do. *Schema Matching and Mapping-based Data Integration*. PhD thesis, University of Leipzig, 2006.

[DR02] H. H. Do and E. Rahm. COMA – A System for flexible Combination of Schema Matching Approaches. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'02)*, 2002.

[DR07] H. H. Do and E. Rahm. Matching Large Schemas: Approaches and Evaluation. In *Information Systems*, volume 32, 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[EFB01] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented Programming: Introduction. *Communications of the ACM*, 44(10):29–32, October 2001.

[EH91] M. Edwards and S. Howell. A Methodology for System Requirements Specification and Traceability for Large Real-time Complex Systems. In *Technical Report, U.S. Naval Surface Warfare Center Daahlgren Division*, 1991.

[EJF$^+$09] C. Elsner, M. Jaeger, L. Fiege, C. Schwanninger, and B. Grammel. Deliverable D5.4: Description of all Case Studies. Technical report, AMPLE, 2009.

[EPK06] K. Engel, R. Paige, and D. Kolovos. Using a Model Merging Language for Reconciling Model Versions. In *Proceedings of the European Conference on Model Driven Architecture Foundations and Applications (ECMDA'06)*, 2006.

[EPT04] H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In *Proceedings of the International Conference on Graph Transformation*, number 3256 in Lecture Notes in Computer Science, pages 161–177. Springer, 2004.

[Fel98] C. Fellbaum. *WordNet: An Electronic Lexical Database*. The MIT Press, 1998.

[FHLN08] J. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems, (MoDELS'08)*, 2008.

[FHN06] J. Falleri, M. Huchard, and C. Nebut. Towards a Traceability Framework for Model Transformations in Kermata. In *Proceedings of ECMDA Traceability Workshop*, 2006.

[Foua] Eclipse Foundation. Acceleo. `http://www.eclipse.org/acceleo/`.

[Foub] Eclipse Foundation. ATL Virtual Machine specification. `http://www.eclipse.org/atl/documentation/old/ATL_VMSpecification[v00.01].pdf`.

[Fouc] Eclipse Foundation. ATL Zoo of Transformations. `http://www.eclipse.org/m2m/atl/atlTransformations/`.

[Foud] Eclipse Foundation. Atlas Transformation Language (ATL). `http://www.eclipse.org/atl/`.

[Foue] Eclipse Foundation. EMF Compare. `http://www.eclipse.org/emf/compare/`.

[Fouf] Eclipse Foundation. MOFscript Transformation Language. `http://www.eclipse.org/gmt/mofscript/`.

[Foug] Eclipse Foundation. Xpand from openArchitectureWare. `http://www.eclipse.org/modeling/m2t/`.

[FV07] M. D. D. Fabro and P. Valduriez. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In *Proceedings of SAC*, 2007.

[FW07] S. Förtsch and B. Westfechtel. Differencing and Merging of Software Diagrams, State of the Art and Challenges. In *Proceedings of ICSOFT*, 2007.

[GG07] I. Galvao and A. Goknil. Survey of Traceability Approaches in Model-Driven Engineering. In *Proceedings of the Eleventh IEEE International EDOC Enterprise Computing Conference*, 2007.

[GHKR10] A. Gross, M. Hartung, T. Kirsten, and E. Rahm. On Matching Large Life Science Ontologies in Parallel. In *Proceedings of Data Integration in the Life Sciences (DILFS'10)*, 2010.

[GHZ$^+$11] O. Gotel, J. Cleland-Huang J. Huffman Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic. The Grand Challenge of Traceability. Technical Report CoEST-2011-001, Center of

Excellence for Software Traceability, 2011.

[GK09] B. Grammel and K.Voigt. Foundations for a Generic Traceability Framework in Model-Driven Software Engineering. In *Proceedings of the ECMDA Traceability workshop*, 2009.

[Gro06] Object Management Group. Meta object facility (mof) 2.0: Core specification. OMG, 2006.

[GS04] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools.* Wiley & Sons, 2004.

[GXTL10] X. Gao, B. Xiao, D. Tao, and X. Li. *A Survey of Graph Edit Distance*, pages 1–17. Pattern Analysis & Applications, 2010.

[HA09] S. Hanif and M. Aono. An Efficient and Scalable Algorithm for Segmented Alignment of Ontologies of Arbitrary Size. *Journal of Web Semantics*, 2009.

[Har11] F. Hartmann. *Safe Template Processing of XML Documents.* PhD thesis, Technical Universitiy of Dresden, 2011.

[Hei10] T. Heinze. Structural Similarity Algorithms using planar/reducible Graphs for Metamodel Matching. Master's thesis, Technical Universitiy of Dresden, 2010.

[HG97] S. Helming and M. Göbel. ZOPP Objectives-oriented Project Planning. *Objectives-oriented Project Planning*, 1997.

[HJD11] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering.* Springer, 2011.

[Hri09] I. I. Hristov. Towards Multi-Dimensional Analysis of Trace Relations. Master's thesis, Technical University of Dresden, 2009.

[HZL$^+$07] W. Hu, Y. Zhao, D. Li, G. Cheng, H. Wu, and Y. Qu. Falcon-ao: Results for OAEI 2007. In *Proceedings of the 2nd International Workshop on Ontology Matching (OMŠ07)*, 2007.

[IEE90] IEEE Standards Association. *Standard Glossary of Software Engineering Terminology, Std 610.12-1990.* IEEE Computer Society, 1990.

[Int08] International Organization for Standardization. *ISO/IEC 12207 Standard: Systems and Software Engineering - Software life cycle processes*, 2008.

[Iva10] P. Ivanov. Data Model Enhancement of a Matching System based on comparative Analysis of related Tools. Master's thesis, Technical University Dresden, 2010.

[Jou05] F. Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the ECMDA Traceability Workshop 2005*, 2005.

[Kas09] S. Kastenholz. Generic Traceability Interface for Arbitrary Transformation Engines in Context of Model-driven Software Development. Master's thesis, Technical Universitiy of Dresden, 2009.

[Kas11] S. Kastenholz. Model Matching for Traceability in the Context of Model-driven Software Development. Master's thesis, Technical Universitiy of Dresden, 2011.

[KC05] C. H. P. Kim and K. Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *Proceedings of the European Conference on Model Driven Architecture*, 2005.

[KLD02] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented Product Line Engineering. In *Proceedings of IEEE Software*, 2002.

[Kle08] A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison Wesley, 2008.

[Kol09] D. S. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Proceedings of ECMDA-FA '09, the 5th European Conference on Model Driven Architecture - Foundations and Applications*. Springer-Verlag, 2009.

[KP02] A. Knethen and B. Paech. A Survey on Tracing Approaches in Practice and Research. Technical report, Frauenhofer IESE, 2002.

[KP13] D.S. Kolovos and R.F. Paige. *Extensible Platform for Specifcation of Integrated Languages for mOdel maNagement (Epsilon)*. Epsilon, `http://www.eclipse.org/epsilon/`, 2013.

[KPP06a] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proceedings of GaMMa'06, International Workshop on Gobal Integrated Model Management*. ACM Press, 2006.

[KPP06b] D.S. Kolovos, R.F. Paige, and F. Polack. On-demand merging of traceability links with models. In *Proceedings of ECMDA Workshop on Traceability*, 2006.

[KR09] H. Köpcke and E. Rahm. Frameworks for Entity Matching: A comparison. In *Journal of Data and Knowledge Engineering*, volume 69, pages 197–210, 2009.

[KRPP09] D. Kolovos, D. Ruscio, A. Pierantonio, and R. Paige. Different Models for Model Matching: An Analysis of Approaches to support Model Differencing. In *Proceedings of CVSM09*, 2009.

[Kru56] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society*, volume 7, 1956.

[KWN05] U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for Uml Models. In *In Proceedings of the Software Engineering Conference*, 2005.

[Lan] XML Path Language. XPath. `http://www.w3.org/TR/xpath/`.

[LG08] H. Lochmann and B. Grammel. The Sales Scenario: A Model-Driven Software Product Line. In *Proceedings of the Software Engineering Conference*, 2008.

[LGJ07] Y. Lin, J. Gray, and F. Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models. In *European Journal of Information Systems*, 2007.

[LHL02] M.R Lyu, J.R. Horgan, and S. London. A Coverage Analysis Tool for the Effectiveness of Software Testing. In *Proceedings of IEEE Transactions on Reliability*, volume 43, pages 527 – 535, 2002.

[Lim09] A. E. Limon. *An Advanced Traceability Schema as Baseline to Improve Supporting Lifecycle Processes.* PhD thesis, Technical University of Madrid, 2009.

[LLC10] Books LLC. *Search Algorithms: Dijkstra's Algorithm, Search Algorithm, Hash Table, Hash Function, Binary Search Algorithm, Grover's Algorithm.* General Books LLC, 2010.

[LTLL08] J. Li, J. Tang, Y. Li, and Q. Luo. RiMOM: A dynamic multistrategy ontology alignment frameworkm. In *IEEE Transactions on Knowledge and Data Engineering*, 2008.

[MG06] T. Mens and P. Van Gorp. A Taxonomy of Model Transformations. In *Electronic Notes in Theoretical Computer Science*, 2006.

[MGMR02] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile Graph Matching Algorithm and its Application to Schema Matching. In *Proceedings of the International Conference on Data Engineering*, 2002.

[MM63] J. C. Miller and C. J. Maloney. Systematic Mistake Analysis of Digital Computer Programs. In *Communications of the ACM*, volume 6, pages 58–63, 1963.

[Muc10] P. Mucha. Musterbasiertes Abbilden von Metamodellen. Master's thesis, Technical Universitiy of Dresden, 2010.

[MW05] Merriam-Webster. *The Merriam-Webster Dictionary.* 2005.

[NB04] M. Neuhaus and H. Bunke. An Error-tolerant Approximate Matching Algorithm for Attributed Planar Graphs and its Application to Fingerprint Classification. In *Proceedings of the 10th International Workshop on Structural and Syntactic Pattern Recognition*, 2004.

[NSC⁺07] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *Proceedings of the International Conference on Software Engineering*, 2007.

[OBF08] G. K. Olsen, H. Bruneliere, and M. D. D. Fabro. Model Engineering Deliverable D3.2.b: Trace Establishment Support. Technical report, MODELPLEX, 2008.

[Obj11] Object Management Group. *MOF 2.0 Query/View/Transformation Specification*, 2011.

[Ols09] G. K. Olsen. Model Engineering Deliverable D3.2.c: Trace model analysis Report and Tool. Technical report, MODELPLEX, 2009.

[OO07] G. K. Olsen and J. Oldevik. Scenarios of Traceabilty in Model-to-Text Transformations. In *Proceedings of ECMDA*, 2007.

[PB08] R. Paige and M. Barbero. Model Engineering: Deliverable D3.2a Traceability Techniques. Technical report, MODELPLEX, 2008.

[PBR10] E. Peukert, H. Berthold, and E. Rahm. Rewrite Techniques for Performance Optimzation of Schema Matching Processes. In *Proceedings of 13th International Conference on Extending Database Technology (EDBT'10)*, 2010.

[PDK⁺11] R. F. Paige, N. Drivalos, D. S. Kolovos, K. J. Fernandes, C. Power, G. K. Olsen, and S. Zschaler. Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. In *Software and Systems Modeling*, volume 10. Springer Verlag, 2011.

[PMK10] E. Peukert, S. Massmann, and K. Koenig. Comparing Similarity Combination Methods for Schema Matching. In *Proceedings of the Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, 2010.

[POKZ08] R.F. Paige, G.K. Olsen, D.S. Kolovos, and S. Zschaler. Building Model-Driven Engineering Traceability Classifications. In *Proceedings of the ECMDA-Workshop on Traceability*, 2008.

[PRG07] C. Pohl, A. Rummler, and B. Grammel. Improving Traceability in Model-Driven Development of Business Applications. In *Proceedings of the ECMDA Traceability Workshop*. ECMDA, 2007.

[Pri00a] U. Priss. Faceted Information Representation. In *Proceedings of Conference on Conceptual Structures*, 2000.

[Pri00b] U. Priss. Faceted Knowledge Representation. In *Electron. Trans. Artif. Intell.*, volume 4, 2000.

[Pri08] U. Priss. Facet-like Structures in Computer Science. In *Axiomathes 18*, volume 18 of *2*, pages 243–255, 2008.

[Ran33] S. R. Ranganathan. *Colon Classification.* Madras Library Association, 1933.

[RB01] E. Rahm and P.A. Bernstein. A Survey of Approaches to Automatic Schema Matching. In *the VLDB Journal*, volume 10, pages 334–350, 2001.

[RFFB05] R. Reddy, R. France, F. Fleuery, and B. Baudry. Model Composition - A Signature-based Approach. In *Proceedings of the Aspect Oriented Modeling workshop held with MODELS/UML*, 2005.

[Rij79] C. J. Van Rijsbergen. *Information Retrieval.* Butterworth-Heinemann, 1979.

[RJ01] B. Ramesh and M. Jarke. Towards Reference Models for Requirements Traceability. In *IEEE Transactions on Software Engineering*, volume 21, 2001.

[RPKP08] L. Rose, R. Paige, D. S. Kolovos, and F. A. C. Polack. The Epsilon Generation Language. In *Proceedings of the ECMDA-FA*, 2008.

[RV08] J. E. Rivera and A. Vallecillo. Representing and Operating with model differences. In *Proceedings of TOOLS EUROPE 2008.* Springer Verlag, 2008.

[SBPM08] D. Steinberg, F. Budinski, M. Paternostro, and E. Merks. *EMF Eclipse Modelling Framework.* Addison Wesley, 2008.

[SE05] P. Shvaiko and J. Euzenat. A Survey of Schema-based Matching Approaches. In *Journal on Data Semantics*, volume 3730, pages 146–171, 2005.

[SG08] M. Schmidt and T. Gloetzner. Constructing Difference Tools for Models using the SiDiff Framework. In *Proceedings ICSE Companion '08: Companion of the 30th international conference on Software engineering*, 2008.

[SKR$^+$10] A. Sousa, U. Kulesza, A. Rummler, N. Anquetil, R. Mitschke, A. Moreira, V. Amaral, and J. Araujo. A Model-Driven Traceability Framework for Software Product Lines. *Software and Systems Modeling*, Volume 9, Issue 4, 2010.

[SV06] Thomas Stahl and Markus Voelter. *Model Driven Software Development.* John Wiley & Sons, 2006.

[SvKJ06] M. Smiljanic, M. van Keulen, and W. Jonker. Using Element Clustering to increase the Efficiency of XML Schema Matching. In *Proceedings of the ICDE Workshops*, 2006.

[SW08] B. Smith and L. Williams. A Survey on Code Coverage as a Stopping Criterion for Unit Testing. Technical report, North Carolina State University, 2008.

[Tun09] D. Tunkelang. *Faceted Search*. Morgan and Claypool Publishers, 2009.

[VBJB07] B. Vanhooff, S. Van Baelen, W. Joosen, and Y. Berbers. Traceability as Input for Model Transformations. In *Proceedings of the ECMDA Traceability Workshop*, 2007.

[VIR10] K. Voigt, P. Ivanov, and A. Rummler. MatchBox: Combined Metamodel Matching for Semi-automatic Mapping Generation. In *Proceedings of the SAC2010*, 2010.

[Voi09] K. Voigt. Towards Combining Model Matchers for Transformation Development. In *Proceedings of FTMDD at ICEIS'09*, 2009.

[Voi11] K. Voigt. *Structural Graph-based Metamodel Matching*. PhD thesis, Technical Universitiy of Dresden, 2011.

[WF74] R. A. Wagner and M. J. Fischer. The String to String Correction Problem. *Journal of the ACM*, 21:168–173, 1974.

[WJSA06] S. Walderhaug, U. Johansen, E. Stav, and J. Aagedal. Towards a Generic Solution for Traceability in MDD. In *Procedings of the ECMDA Traceabilty Workshop*, 2006.

[XS05] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for object-oriented Design Differencing. In *Proceedings of the IEEE/ACM international Conference on Automated Software Engineering*, 2005.

[ZZLT10] Xiao Zhang, Qian Zhong, Juanzi Li, and Jie Tang. RiMOM results for OAEI 2010. In *Proceedings of the 5th International Workshop on Ontology Matching (OM'10)*, 2010.