Adaptation and evaluation of generic model matching strategies

Kelly Garces

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ECOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATERIAUX

Année 2010

N° attribué par la bibliothèque

Une approche pour l'adaptation et l'évaluation de stratégies génériques d'alignement de modèles

---

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Informatique

Présentée
et soutenue publiquement par

Kelly Johany Garcés-Pernett

Le 28 Septembre 2010 à l'École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes

Devant le jury ci-dessous :

Président	:	Mehmet Aksit, Professeur	Université de Twente
Rapporteurs	:	Jérôme Euzenat, Directeur de recherches	INRIA Grenoble Rhône-Alpes
		Isabelle Borne, Professeur	Université de Bretagne-Sud
Examinateur	:	Mehmet Aksit, Professeur	Université de Twente
		Frédéric Mallet, Maître de conférences	Université Nice Sophia Antipolis
Directeurs de thèse	:	Jean Bézivin, Professeur	Université de Nantes
		Pierre Cointe, Professeur	École des Mines de Nantes
Responsable Scientifique	:	Frédéric Jouault, Chargé de recherches	École des Mines de Nantes

Équipes d'accueil : AtlanMod, INRIA, EMN
ASCOLA, INRIA, LINA UMR CNRS 6241
Laboratoire d'accueil : Département Informatique de l'École des Mines de Nantes
La Chantrerie – 4, rue Alfred Kastler – 44 307 Nantes

ED: .....

Une approche pour l'adaptation et l'évaluation de stratégies génériques d'alignement de modèles

Adaptation and evaluation of generic model matching strategies

Kelly Johany Garcés-Pernett

Université de Nantes

Contents

Acknowledgments

I owe my deepest gratitude to Mr. Jérôme Euzenat, Research director at INRIA Grenoble Rhône-Alpes, and to Mrs. Isabelle Borne, Professor at University of Bretagne du Sud, for reviewing my thesis document during summer holidays.

I thank Mr. Mehmet Aksit, Professor at University of Twente, for leaving his dutch laboratory to attend my PhD. defense in Nantes.

I thank Mr. Frédéric Mallet, Associated Professor at University Nice Sophia Antipolis, for accepting the invitation to my defense and for offering me a job where I can continue my researcher career after the PhD.

It was an honor for me to have Jean Bézivin, Pierre Cointe, and Frédéric Jouault like advisers. Jean Bézivin transfered me the passion for innovation in software engineering and MDE. Pierre Cointe helped me to make my proposals understandable to people out of the MDE community. At last, Frédéric Jouault made available his expert support in the technical issues.

I would like to show my gratitude to Joost Noppen and Jean-Claude Royer for their valuable writing lessons in English and French, respectively.

I am indebted to many of my AtlanMod, ASCOLA, and LINA colleagues who have supported me in many ways these 3 years: Wolfgang, Guillaume, Hugo, Mayleen, Ismael, Angel, Patricia, Audrey... Thank you!

Thank to Diana Gaudin, Catherine Fourny, Annie Boilot, Nadine Pelleray, and Hanane Maaroufi for all the administrative help.

I would like to thank Luz Carime and Daniel, Amma and Nana, Dina and Laurent for their friendship, prayers, and good meals.

It is a pleasure to thank those relatives who made this thesis possible: my grandmother, Hilda, who taught me to struggle hard to get on in life, my father who financed me at the very beginning of my PhD., and the rest of my family that encouraged me when the loneliness got me down. I thank my lovely fiance, Michael, for his patient wait. Our love grew despite the 8456 km that separated us the last years.

Finally, I sincerely thank God, he is the real author of this achievement.

Chapter 1

Introduction

Model-Driven Engineering (MDE) has become an important field of software engineering. For MDE, the first-class concept is *model*. A model represents a view of a system and is defined in the language of its metamodel. Metamodels, in turn, conforms to a metametamodel which is defined in terms of itself. A running program, an XML document, a database, etc., are representations of systems found in computer science, that is, they are models.

In addition to model-based organization, MDE introduces the notion of *model transformation*. That is, a set of executable mappings that indicate how to derive an output model from an input model. Mappings are written in terms of concepts from the corresponding input and output metamodels. Model transformations enable (semi)automatic generation of code from models.

MDE has acquired the attention of industry. For instance, the AUTOSAR standard, developed by the automobile manufacturers and containing around 5000 concepts, defines a metamodel to specify automotive software architectures [1].

In response to the scalability challenge (the need for large (meta)models and large model transformations in consequence), academy and industry have invested into tool support. Therefore, a certain level of maturity has been achieved it for (meta)modeling and model transformation development. A next stage is to automate these tasks, above all, the latter. Several approaches have investigated that [2][3][4][5][6]. All of them found a source of inspiration on the *matching* operation which has been throughly studied in databases systems and ontology development.

Instead of manually finding mappings (which is labor intensive and error-prone as metamodels are large), a *matching strategy* (also referred to *matching algorithm*) automatically discovers an initial version of them. The matching strategy involves a set of heuristics, each heuristic judges a particular metamodel aspect, for example, concept names or metamodel structure. The user can manually refine initial mappings, finally, a program derives a model transformation from them.

Didonet del Fabro's thesis represents mappings in the form of *weaving models* [2]. A weaving model contains relationships between (meta)model elements. Furthermore, a matching strategy is implemented as a chain of ATL *matching transformations*. ATL (AtlanMod Transformation Language) is a general purpose model transformation language [7]. Each matching transformation corresponds to a concrete heuristic. A chain can be

tuned by selecting appropriate matching transformations and additional parameters. The AMW (AtlanMod Model Weaver) tool enables the user to refine discovered mappings. At last, HOTs (Higher-Order Transformations) derive model transformations from weaving models [8][9].

The results reported by Didonet del Fabro's thesis and the recently gained importance of matching in MDE are the motivations of this thesis and its starting point. Our approach differs from previous work because it focus not only on *metamodel matching strategies* but also on *model matching strategies*, both of them very useful on MDE. We refer to these kinds of strategies as M2-to-M2 and M1-to-M1, respectively. The former kind of strategy discovers mappings between pairs of metamodels. A main application of M2-to-M2 is model transformation generation. The latter kind of algorithm, in turn, determinates mappings between pairs of models. These mappings are useful in many ways. For example, they can be taken as input by M2-to-M2 matching algorithms to improve their accuracy or they can leverage other important MDE operations, e.g., model synchronization.

To support M2-to-M2 and M1-to-M1 matching algorithm development, it is necessary to tackle issues not addressed in Didonet del Fabro's thesis. The thesis highlights the importance of adapting matching algorithms since no algorithm perfectly matches all pairs of models. Early experimentations demonstrate the feasibility of using transformation chains for such an adaptation. These experimentations nonetheless reveal issues concerning *reusability* of matching heuristics and *evaluation* of customized algorithms.

Firstly, we elaborate on the reusability issue. The ATL matching transformations contributed by [2] match only metamodels conforming to the Ecore metametamodel [10]. Even though some transformations compare very standard features (e.g., names), they may be more or less applicable to metamodels conforming to other metametamodels (e.g., MOF by OMG [11])[1]. In contrast, their applicability substantially decreases when one wants to match models. We call this issue *coupling of matching heuristics to metamodel*.

Related to the issue of evaluation, it is an essential need in matching. Evaluation basically compares computed mappings to a *gold standard* [12]. To evaluate algorithms, one requires test cases: pairs of meta(models) and gold standards. In MDE, each approach defines its own test cases and methodology, therefore, it is difficult to establish a consensus about its real strengths and weaknesses. We refer to these issues as *lack of a common set of test cases* and *low evaluation efficiency*.

This thesis addresses the two issues mentioned above by means of the following contributions.

1.1 Contributions

Below we briefly highlight the four contributions of the thesis and the foundations on which such contributions are based.

A survey of model matching approaches We provide a broad survey of the recently emerged MDE matching systems. This contribution complements other surveys mostly done by the ontology/database community [13][14].

[1] It is possible by executing a prior step that translates metamodels into the Ecore format.

Matching heuristics independent of technical space and abstraction level We propose matching transformations that can be reused in M2-to-M2 or M1-to-M1 matching algorithms. To achieve reusability, we rely on two notions: *technical spaces* and *Domain Specific Languages (DSLs)*.

According to Kurtev et al. [15] a technical space is a broad notion denoting a technology, for example, MDE, EBNF [16], RDF/OWL[17][18]. Each technical space has its own metametamodel. The reuse of M2-to-M2 matching algorithms independently of technical spaces is possible by using projectors and DSLs. A projector translates metamodels, built in a concrete technical space, into a format that our matching transformations can process. We use the projectors available on the AmmA platform [19][15].

DSLs have gained importance due to their benefits in expressiveness, testing, etc., over General Purpose Languages (GPLs), e.g., Java [20]. We have designed a new DSL called the AtlanMod Matching Language (AML). AML notations hide types, therefore, it is possible to reuse matching transformations in diverse technical spaces and abstraction levels (i.e., metamodels and/or models). In addition, we have implemented a compiler that translates AML matching transformations into executable ATL code. AML transformations are substantially less verbose than their corresponding ATL versions.

AML aims at facilitating matching transformation development and algorithm configuration. Like existing approaches, AML enables a coarse-grained customization of matching algorithms, i.e., how to combine heuristics. Moreover, AML moves a step forward with respect to previous work: the language allows a fine-grained customization, i.e., AML provides constructs simplifying matching transformations themselves. Thus, users may get a quick intuition about what a matching transformation does, its interaction with other transformations, and its parameters. We have developed AML on top of the AmmA platform. Furthermore, we have contributed a library of linguistic/structure/instance-based matching transformations along with strategies. These matching transformations have been mostly inspired by the ontology community. The reason for that is to investigate the efficiency that heuristics, used in other technical spaces, have in MDE.

To demonstrate that our work goes beyond the MDE technical spaces (e.g., Ecore, MOF), we have applied our matching strategies to pairs of OWL ontologies. Like metamodels, ontologies are data representation formalisms. A difference between metamodels and ontologies is the application domain. Over the last decade, whereas the software engineering community has promoted metamodels, the Web, and AI communities have launched ontologies. An ontology is a body of knowledge describing some particular domain using a representation vocabulary [21]. For instance, ontologies have been used to represent Web resources and to make them more understandable by machines. We have preferred ontologies over other formalisms (e.g., database schemas) for two reasons. Firstly, ontologies can be translated into metamodels. A second and most important rationale is that the ontology community has a mature evaluation initiative called OAEI [22] which systematically evaluates ontology matching systems and publishes their results on the Web. The availability of these results facilitates the comparison of our approach to other systems.

Modeling artifacts to automate matching algorithm evaluation We obtain test cases from modeling repositories which are growing at a constant rate. There, we find

models and metamodels, and we derive gold standards from transformations. By using a *megamodel*, our approach executes matching algorithms over test cases in an automatic way. Bezivin et al. [23] propose the megamodel term to refer to kind-of map where all MDE artifacts and their relationships are represented.

Our contribution is to automatically build a megamodel representing the transformations stored in a given repository, each transformation corresponding to a matching test case. Both, the built megamodel and an additional script, guide the evaluation execution by indicating test cases, matching algorithms, and graphics. With respect to graphics, we have implemented transformations that render matching results in HTML and spreadsheet format. The automation offered by our approach may reduce the time spent during evaluations and increase the confidence on results.

As for the previous contribution, we show how to extend our evaluation approach beyond the MDE technical spaces. For example, we depict how the OAEI may be improved by using our modeling test cases, megamodels, and metric visualization means. There exist ways to close the gap between ontologies and modeling technologies. AML, for example, uses an AmmA-based projector named EMFTriple [24] to transform metamodels into ontologies. Moreover, we have developed transformations that translate our gold standards to a format well-known by the OAEI.

Three use cases based on matching Three use cases show how M2-to-M2 and M1-to-M1 matching algorithms complement other techniques in order to deal with MDE needs. In addition, the use cases underscore the reuse of matching heuristics independently of the abstraction level.

The first use case is about *co-evolution*, an interesting research topic in MDE. Just as any software artifact, metamodels are likely to evolve. Co-evolution is about adapting models to its evolving metamodel. Many approaches dealing with co-evolution have recently appeared. Most of them relying on traces of metamodel changes to derive adapting transformations. In contrast, we propose an M2-to-M2 matching algorithm that discovers the changes first, and then a HOT derives adapting transformations from them.

The second use case is called *pivot metamodel evaluation*. The goal is to evaluate whether a pivot metamodel has been correctly chosen. The interest of pivot metamodels is to reduce the effort of model transformation development. This use case combines M2-to-M2 matching algorithms and our matching evaluation approach.

Finally, the third use case is named *model synchronization*. Its purpose is to bring models in agreement with code. To this end, the use case employs, among other techniques, M1-to-M1 matching algorithms.

1.2 Outline

Below we list the thesis chapters. They contain a number of shortened forms whose meaning is in the list of abbreviations.

- Chapter 2 introduces the thesis context, i.e., MDE and DSLs as promoted by the software engineering community. It outlines criteria concerning the matching operation.

- Chapter 3 presents how several approaches tackle the matching operation. Moreover, the chapter compares the approaches with respect to the criteria defined in Chapter 2. At last, based on this comparison, the chapter describes in detail the issues tackled by the thesis.
- Chapter 4 depicts the phases we have followed to deliver the AML language: analysis, design, and implementation. The analysis phase covers the base concepts of matching. The design part shows how language notations overlap base concepts and implementation units. Finally, we summarize how AML has been implemented according to modeling techniques.
- Chapter 5 gives our approach for automatizing matching algorithms evaluation. The chapter provides a comparison of AML to other ontology/MDE matching systems. AML has been applied to modeling and ontology test cases.
- Chapter 6 reports how AML algorithms have been incorporated in MDE solutions to solve problems such as co-evolution, pivot metamodel evaluation, and model synchronization.
- Chapter 7 revisits the thesis contributions in detail, positions our approach with respect to the criteria established in Chapter 3, and draws future work.

1.3 Publications associated to the thesis

Chapter 4 and Chapter 5 are adapted versions of the following papers and poster:

1. A Domain Specific Language for Expressing Model Matching. In Actes des Journées sur l'IDM, 2009 [25].
2. Automatizing the Evaluation of Model Matching Systems. In Workshop on matching and meaning, part of the AISB convention, 2010 [26].
3. AML: A Domain Specific Language to Manage Software Evolution. FLFS Poster. Journées de l'ANR, 2010.

The results of the co-evolution use case of Chapter 6 have been published in:

1. Adaptation of Models to Evolving Metamodels. Research Report, INRIA, 2008 [27].
2. Managing Model Adaptation by Precise Detection of Metamodel Changes. In Proc. of ECMDA, 2009 [28].
3. A Comparison of Model Migration Tools. In Proc. of Models, 2010 [29].

Chapter 2

Context

2.1 Model-Driven Engineering

According to [30], MDE can be seen as a generalization of object oriented technology. The main concepts of object technology are *classes* and *instances* and the two associated relations *instanceOf* and *inheritsFrom*. An object is an instance of a class and a class could inherit from another class. For MDE, the first-class concept is *model*. A model represents a view of a system and is defined in the language of its metamodel. In other words, a model contains *elements* conforming to *concepts* and relationships expressed in its metamodel. The two basic relations are *representedBy* and *conformsTo*. A model represents a system and conforms to a metamodel. Metamodels, in turn, conforms to a metametamodel which is defined in terms of itself.

Concepts and elements can correspond to classes and instances, respectively. In a first glance that suggests only similarities between MDE and object technology but not variations. Looking deep into the definitions nonetheless reveals how MDE complements object technology. For instance, models enable representation of class-based implementations as well as other aspects of the systems.

In MDE, models are more than means of communication, they are precise enough to generate code from. The basic operation applied on models is model transformation. There exist many implementations of MDE such as MDA by OMG, Model Integrated Computing (MIC), Software Factories [31], etc. The subsequent sections describe MDA, the central MDE concepts in detail, and a model management platform.

2.1.1 Model-Driven Architecture

The word *models* used to be associated to UML models [32]. UML provides diagrams to represent not only a structural view of software (i.e., class diagrams) but also its behavior and interaction. UML is part of the Model-Driven Architecture (MDA) initiative made public by OMG. The goal of MDA is to solve the problems of *portability*, *productivity*, and *interoperability* happening in software industry. To achieve that, MDA proposes separation of software in business and platform models and composition of models by means of model transformations. Besides UML, MDA introduces other technologies such as MOF, XMI, OCL, etc. MOF [11] is a metametamodel indicating concepts as `Classes`

and relationships as `Associations` and `Attributes`. XMI [33], in turn, serializes models in XML format. At last, OCL [34] allows the definition of queries and constraints over models.

MDA proposes the separation of software into Platform Independent Models (PIMs) and Platform Specific Models (PSMs). A PIM considers only features of the problem domain. A PSM, in turn, takes into account implementation issues for the platform where the system will run [35]. A PIM is transformed into one or more PSMs. At last, PSMs are transformed into code. MDA proposes the following technologies:

- **UML** which has been introduced in Chapter 1.
- **MOF** is a meta-language used to define, among other languages, UML [11].
- **XMI** for serialization of MOF models in XML format [33].
- **OCL** to define model constraints [34].

2.1.2 Models, metamodels, metametamodels, and technical spaces

Favre [36] suggests MDA as a concrete incarnation of MDE implemented in the set of specification defined by OMG. Moreover, Kent [37] identifies various dimensions not covered by MDA, e.g., a software system involves not only an architecture but also a development process. Thus, MDE is much more than UML and MDA. MDE is a response to the lacks of MDA, below we present the concepts which MDE relies on. We will use these concepts in the remaining chapters.

A model represents a system by using a given notation and captures some characteristics of interest of that system. [38] gives the following formal definition of model:

Definition 1. *A directed multigraph* $G = (N_G, E_G, \Gamma_G)$ *consists of a set of nodes* N_G*, a set of edges* E_G*, and a function* $\Gamma_G : E_G \rightarrow N_G x N_G$*.*

Definition 2. *A model* M *is a triple* (G, ω, μ) *where:*

- $G = (N_G, E_G, \Gamma_G)$ *is a directed multigraph.*
- ω *is itself a model (called the reference model of* M*) associated to a multigraph* $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$*.*
- $\mu : N_G \cup E_G \rightarrow N_\omega$ *is a function associating elements (i.e., nodes and edges) of* G *to nodes* G_ω *(metaelements or types of the elements).*

The relation between a model and its reference model is called *conformance*. We denoted it as *conformsTo*, or simply (c2) (see Fig. 2.1). Def. 2 allows an infinite number of upper modeling levels. For practical purposes, MDE has suggested a three-level architecture shown in Fig. 2.2. The M3 level covers a metametamodel. The M2 level, in turn, includes metamodels. The M1 level embraces models (or terminal models). The system corresponds to the M0 level. The M0 is not part of the modeling world. Models at every level conform to a model belonging to the upper level. The metametamodel conforms to itself.

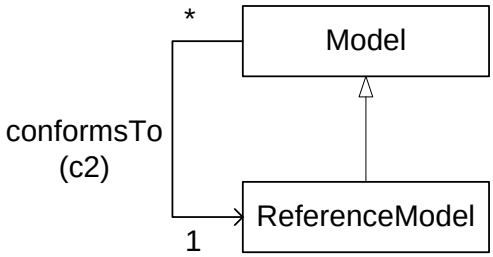


Figure 2.1: Definition of model and reference model

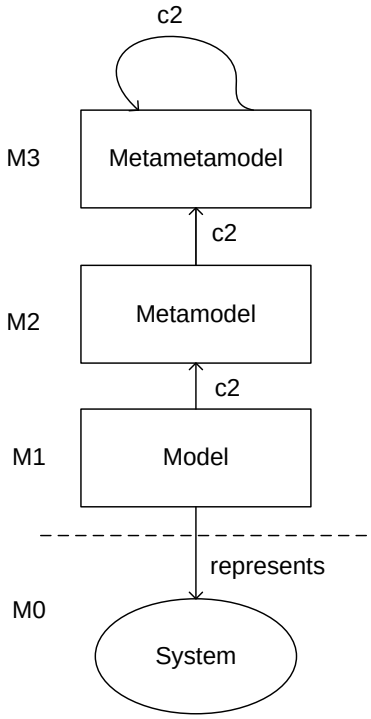


Figure 2.2: An architecture of three levels of abstraction

A technical space (denoted as TS) is a model management framework accompanied by a set of tools that operate on the models definable within the framework [15]. Each technical space has within a metametamodel. Below we list the metametamodels of diverse technical spaces. The list is not exhaustive, the selection of items is driven by their popularity and contributions to IT disciplines. In general, the technical spaces we list (except SQL-DDL) are based on two formats: XML and/or RDF [17]. One of XML strengths is its ability to describe strict hierarchies. RDF, in turn, is a standard for data exchange on the Web. RDF uses URIs to name the relationship between data sources as well as the two ends of the link (this is usually referred to as a *triple*). RDF statements (or triples) can be encoded in a number of different formats, whether XML based (e.g., RDF/XML) or not (Turtle, N-triples). The usage of URIs makes it very easy to seamlessly merge triple sets. Thus, RDF is ideal for the integration of possibly heterogeneous information on the Web.

- **SQL-DDL** is used to define schemas for relational databases. Relational schemas contain a set of tables. Tables have a set of columns. The different kinds of relationships between different tables are defined using foreign keys. SQL-DDL schemas have a text-based format.

- **XSD** is used to describe the structure of XML documents[1]. XSD schemas are based on XML.

- **OWL** is a language to define ontologies. OWL is based on RDF and XML. OWL adds extra vocabulary to RDF, this allows the description of more complex classes and properties, transitivity properties, or restrictions over properties and classes. An ontology differs from an XML Schema in that it is a knowledge representation. On top of it one can plug agents to reason and infer new knowledge [18].

- **MOF** as mentioned above this is an adopted OMG specification. MOF provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems [11]. A number of technologies standardized by OMG, including UML and XMI, use MOF.

- **Ecore** adapts MOF to the EMF. Ecore allows the specification of metamodels. MOF and Ecore have equivalent concepts and relationships (`EClasses` similar to `Classes`), a difference is that Ecore incorporates Java notions (e.g., `EAnnotation`). One of the main advantages of Ecore is its simplicity and the large number of tools developed on top of it.

- **KM3** is a language for representing metamodels [38]. The KM3 definition corresponds to the metametamodel. The main advantage of KM3 over other languages is its simplicity, i.e., lightweight textual metamodel definition. Metamodels expressed in KM3 may be easily converted to/from other notations like Ecore or MOF.

Fig. 2.3 illustrates the concepts mentioned above. It shows the corresponding multigraphs of a metametamodel, a metamodel, and a model. Each of them has nodes and

[1] http://www.w3.org/XML/Schema

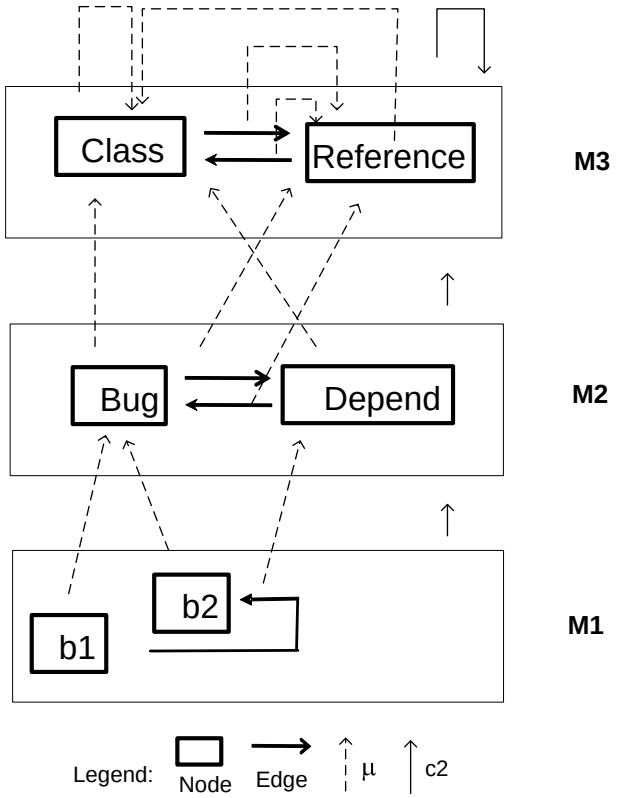


Figure 2.3: Example of the KM3 three-level modeling architecture

edges. The metametamodel is KM3, this example illustrates two of its central concepts, i.e., `Class` and `Reference`. The μ function indicates that:

- The metaelement of `Bug` and `Depend` is `Class`. The references between `Bug` and `Depend` have `Reference` like metaelement.
- The metaelement of `b1` and `b2` is `Bug`, `b1` and `b2` have a dependency whose metaelement is `Depend`.
- Finally, coming back to the M3 level, note that KM3 is defined in terms of itself. For example, the metaelement of `Class` and `Reference` is `Class`.

2.1.3 Model transformations

Model transformations bridge the gap between the models representing a system. A model transformation takes a set of models as input, visits the elements of these models and produces a set of models as output.

Fig. 2.4 illustrates the base schema of a model transformation. Let us consider a transformation from the input model *MA* into the output model *MB*. *MA* conforms to metamodel *MMA* (as indicated by the *c2* arrows). *MB* conforms to metamodel *MMB*.

Following the main principle of MDE, "everything is a model", one can consider a transformation such as model too. Thus, the model transformation *MT* conforms to the transformation metamodel *MMT*. *MMT* defines general-purpose and fixed operations

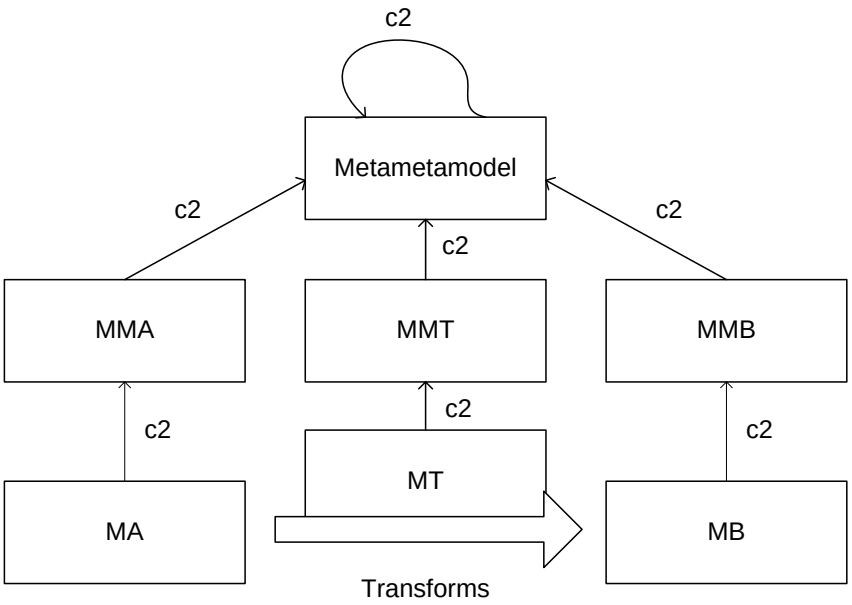


Figure 2.4: Base schema of a model transformation

which allow model manipulation. The model transformation essentially defines executable mappings between the concepts of *MMA* and *MMB*. All metamodels conform to the same metametamodel. Fig. 2.4 does not consider multiple input or output models, however, this schema can be extended to support multiple input and/or output models.

Vara describes different approaches to implement model transformations [39] (pag. 88). Below we summarize the approaches somehow related to the thesis:

- **Direct model manipulation.** Model transformations are developed by using GPLs (e.g., a Java API [40]). It is a very low-level approach for model transformation coding; expressions for navigation and creation of models are too complex.

- **XML-based.** Since models have a XMI format, XSLT (XML extensible Stylesheets Language Transformations) can be used to specify model transformations. XML-based approaches move a step forward with respect to direct manipulation approaches; one can navigate models by direct referencing of metamodel concepts. XSLT programs nonetheless remain complex and verbose.

- **Graph-based.** These approaches see models like pure graphs. A graph-based transformation takes as input an empty graph, and its rules build the output graph in a stepwise manner. The rules execution order can be explicitly specified (e.g., by means of a dataflow).

- **Declarative.** Provide high-level constructs to manipulate models. Graph-based and declarative approaches substantially improve user experience about model transformation development. The former often provides graphical interfaces to specify transformations, the latter mostly provides textual notations.

The Query/Views/Tranformations (QVT) Request for Proposal (RFP) [41], issued by OMG, sought a standard model transformation framework compatible with the MDA suite. The RFP pointed the need for three sublanguages:

- *Core* allows the specification of transformations as a set of mappings between metamodel concepts.
- *Relations* is as declarative as *Core*. A difference is that the *Relations* language has a graphical syntax.
- *Operational Mappings* extends *Core* and *Relations* with imperative constructs and OCL constructs.

Several formal replies were given to the RFP. Some of them are graph-based approaches (e.g., VIATRA [42], AGG [43]), others are declarative (e.g., ATL [7], Kermeta [44]). The OMG has adopted its own model transformation framework, i.e., a declarative language called QVT MOF 2.0 [45]. The use of most of implementations (included QVT MOF 2.0) is limited because of their youth [46]. Thus, for practical reasons a user might want to use languages with better tool support, for example ATL [47].

Typically an MDE development process involves not only a transformation but a set of transformations chained in a network (i.e., a transformation chain) [48]. To go from models to executable code, the chain often includes *Model-to-Model* and *Model-to-Text* transformations. We have introduced Model-to-Model transformations at the very beginning of this section, i.e., programs that convert a model (or a set of models) into another model (or set of models). Model-to-Text transformations convert a model element into a text-based definition fragment [49]. The languages mentioned in the previous paragraph are Model-to-Model. Some examples of Model-to-Text transformation languages are Acceleo [50], MOFScript [51], etc.

2.2 Domain Specific Languages

Many computer languages are *domain specific* rather than general purpose. Below the definition of DSL given in [52]:

> "A DSL provides notations and constructs tailored toward a particular application domain, they offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs."

Van Deursen et al. [53] mention four key characteristics of DLSs:

1. **Focused on a problem domain**, that is, a DSL is restricted to a specific area including particular *objects* and *operations* [54]. For example, a window-management DSL could include the terms *windows*, *pull-down menus*, *open windows*, etc.
2. **Usually small**, a DSL offers a restricted set of notations and abstractions.

3. **Declarative**, DSL notations capture and mechanize a significant portion of repetitive and mechanical tasks.

4. **End-user programming**, a DSL enables end-users to perform simple programming tasks.

Like classical software, a DSL implies the following development phases: *decision*, *analysis*, *design*, *implementation*, and *deployment*.

Decision Since a DSL development is expensive and requires considerable expertise, the decision phase determinates if a new DSL is actually relevant or not.

Analysis Its purpose is to identify the problem domain and to gather domain knowledge. The input can be technical documents, knowledge provided by domain experts, existing GPL code, etc. The output of domain analysis basically consists of terminology. Domain analysis can be done informally, however there exist well-known methodologies to guide this phase, e.g., FODA (Feature-Oriented Domain Analysis) [55], DSSA (Domain-Specific Software Architectures) [56], etc.

Design This step can be carried out in an informal or formal way. An informal design has within a DSL specification in natural language or/and a set of illustrative DSL programs. A formal design mostly includes concrete and abstract syntaxes, and semantics. Whereas there exist a common way to define syntaxes (i.e., grammar-based systems), there are many semantic specification frameworks but none has been widely established as a standard [15].

Implementation Mernik et al. characterize the following DSL implementation techniques [52]:

- **From scratch**
  - *Interpretation or compilation* are classical approaches to implement GPLs or DSLs. The structure of an interpreter is similar to that of a compiler. Compared to an interpret, a compiler spends more time analyzing and processing a program. However, the execution of such a program is often faster than interpret-resulting code [57]. The main advantage of building a compiler or interpreter is that the implementation of notations is fully tailored toward the DSL. The disadvantage is the high implementation cost.
- **Extending a base language**
  - *Embedded languages*, the idea is to build a library of functions by using the syntactic mechanisms of a base language. Therefore, DSL programs are built in terms of such functions. The benefit is reusing the base language compiler (or interpreter). The disadvantage is that the base language may restraint the new DSL expressiveness.

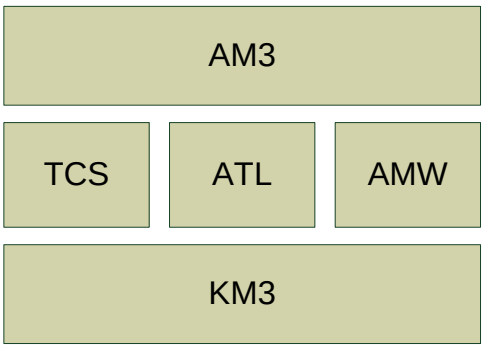


Figure 2.5: AmmA toolkit

– *Preprocessing*, this approach translates new DSL constructs into base language statements. The main advantage is a modest development effort. A disadvantage is that error reporting messages are in terms of base language concepts instead of DSL concepts.

Deployment makes a DSL available to end-users. Users write DSL programs and compile them.

Mernik et al. report DSL development toolkits mostly supporting the implementation phase. These toolkits generate tools from language specifications. Syntax-directed editor, pretty-printer, consistency checker, interpreter or compiler, and debugger are examples of generated tools. Language specifications can be developed in terms of other DSLs. This work focuses on the DSL implementation support offered by the AmmA toolkit.

The AmmA toolkit demonstrates the potential of MDE in DSLs: a metamodel and a set of transformations can (correspondingly) describe the abstract syntax and semantics of a DSL. In addition, projectors bridge MDE and EBNF technical spaces: they derive a model from a program expressed in the visual/graphical or textual concrete syntax of a DSL (and vice versa) [19].

2.3 The AtlanMod model management Architecture (AmmA)

The AmmA toolkit consists of DSLs supporting MDE tasks (e.g., metamodeling, model transformation) as well as DSL implementation. Fig. 2.5 shows the AmmA DSLs which are described in the next subsections.

2.3.1 Kernel MetaMetaModel

As mentioned in Section 2.1.2, KM3 allows the definition of metamodels [38]. Fig. 2.6 shows the basic concepts of the KM3 metametamodel. The `Package` class contains the rest of concepts. The `ModelElement` class denotes concepts that have a `name`. `Classifier` extends `ModelElement`. `DataType` and `Class`, in turn, specialize `Classifier`. `Class` consists of a set of `StructuralFeatures`. There are two kinds of structural features:

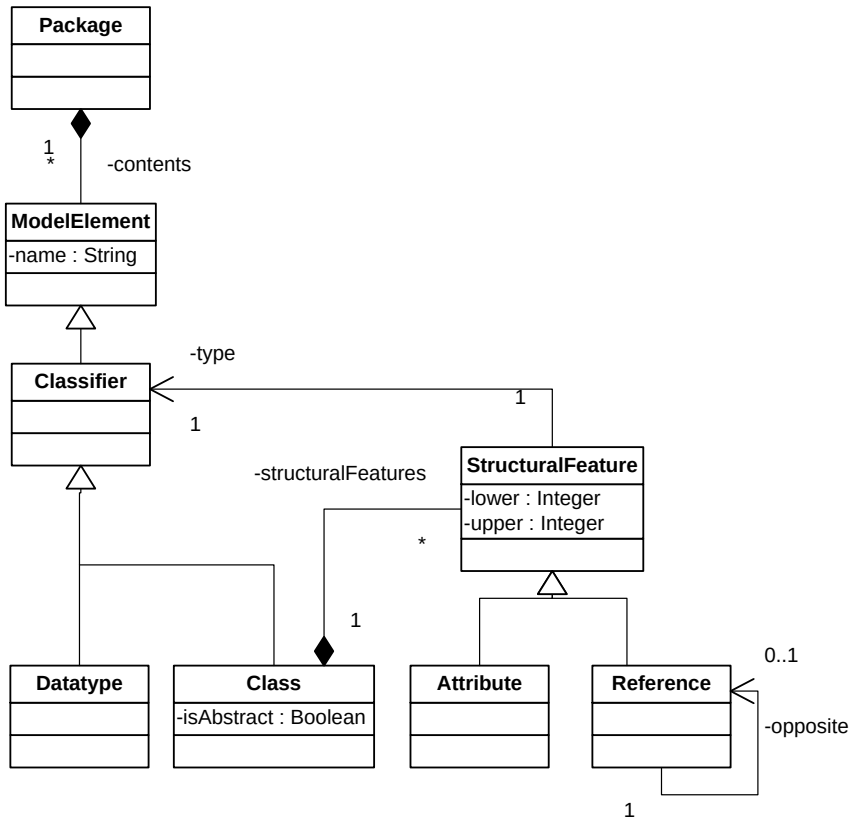


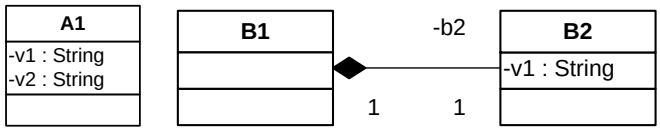
Figure 2.6: KM3 concepts

`Attribute` or `Reference`. `StructuralFeature` has `type` and `multiplicity` (`lower` and `upper` bound). `Reference` has `opposite` which enables the access to the owner and target of a reference.

Listing. 2.1 and Listing. 2.2 give the KM3 notation corresponding to the *MMA* and *MMB* metamodels illustrated in Fig. 2.7. The `A1` and `B2` classes contain the `v1` attribute referring to a primitive data type. The `B1` class, in turn, has the `b2` reference pointing to the `B2` class.

Listing 2.1: MMA metamodel in KM3 notation

```
1 package MMA {
2
3 class A1 {
4  attribute v1 : String;
5  attribute v2 : String;
```



(a) MMA metamodel (b) MMB metamodel

Figure 2.7: The MMA and MMB metamodels

```
6 }
7
8 }
```

Listing 2.2: MMB metamodel in KM3 notation

```
1 package MMB {
2
3 class B1 {
4  reference b2 : B2;
5 }
6
7 class B2 {
8  attribute v1 : String;
9 }
10
11 }
```

2.3.2 AtlanMod Transformation Language

Section 2.2 mentions how MDE can be productively used in DSLs. ATL is a DSL illustrating benefits in the other way around. ATL provides expressions (inspired by OCL) to navigate input models and to restrict the creation of output model elements. Such notations save the implementation of complex and verbose GPL code.

ATL allows the specification of declarative and imperative transformation rules in a textual manner. Let us present some ATL features by means of the *MMA2MMB* transformation. Its input and output metamodels are the *MMA* and *MMB* metamodels listed above.

Listing. 2.3 shows a *matched rule* (or declarative rule) named *A1toB1*. It consists of an `inPattern` (lines 2-3) and an `outPattern` (lines 4-10). The `inPattern` matches the *A1* type (line 3), and the `outPattern` indicates the type of the generated output model elements (lines 5-10), i.e., *B1* and *B2*. Types are specified as follows: `MetamodelName!Type`, for example, `MMA!A1`. An `outPattern` is composed of a set of `bindings`. A binding is the way to initialize output elements from matched input elements (line 6). The ATL virtual machine decides the execution order of declaratives rules.

Listing 2.3: ATL declarative rule

```
1 rule A1toB1 {
2  from
3   s : MMA!A1
4  to
5   t1 : MMB!B1 {
6    b2 <- t2
7   },
8   t2 : MMB!B2 {
9    v1 <- s.v1
10  }
11 }
```

Listing. 2.4 illustrates how to implement part of *A1toB1* by using an imperative rule, i.e., a *called rule* (line 10). It is explicitly called from a matched rule (line 6), like a procedure, and its body may be composed of a declarative `outPattern` (lines 12-14).

Listing 2.4: ATL imperative rule called from a declarative rule

```
1 rule A1toB1 {
```

```
 from
  s : MMA!A1
 to
  t1 : MMB!B1 {
   b1 <- thisModule.A1toB2(s)
  }
}

rule A1toB2 (s : MMA!A1) {
 to
  t2 : MMB!B2 {
   v1 <- s.v1
  }
 do {
  t2;
 }
}
```

ATL encourages declarative style instead of imperative. The latter should be only used when declarative constructs are not enough to support a particular case.

2.3.2.1 Higher-Order Transformations

In MDE, transformations leverage code generation. Transformations can be themselves generated and handled by model-driven development, exactly like traditional programs. An approach to generate transformations is the HOT operation [58][8][9]. A HOT is a model transformation such that its input and/or output models are themselves transformation models. The representation of a transformation as a model conforming to a transformation metamodel makes HOTs possible. Not all model transformation languages provide a transformation metamodel. In this work we will mainly refer to HOTs implemented with ATL.

2.3.3 AtlanMod Model Weaver

AMW provides a graphical interface to manipulate weaving models [59][2]. Whilst a model transformation defines executable mappings, a *weaving model* defines declarative ones. A weaving model captures the relationships (i.e., links) between (meta)model elements. Fig. 2.8 depicts a weaving model containing the relationships between *Ma* and *Mb*. Each relationship (i.e., r_1 or r_2) links a set of elements of *Ma* to a set of elements of *Mb*.

Weaving models conforms to a weaving core metamodel. Listing. 2.5 shows the main weaving concepts, i.e., `WModel` and `WLink`. `WModel` references to woven models. `WLink` has to be extended to define the kinds of links that may be created between woven models. `End` refers to an arbitrary numbers of elements belonging to the woven models.

Listing 2.5: Excerpt of the AMW core metamodel

```
 abstract class WElement {
 attribute name : String;
 attribute description : String;
 reference model : WModel oppositeOf ownedElement;
}

```

[2]This notion differs from the term used in Aspect Oriented Programming (AOP) [60]. Whereas the former refers to weaving between models, AOP weaves (executable) code by means of aspects.

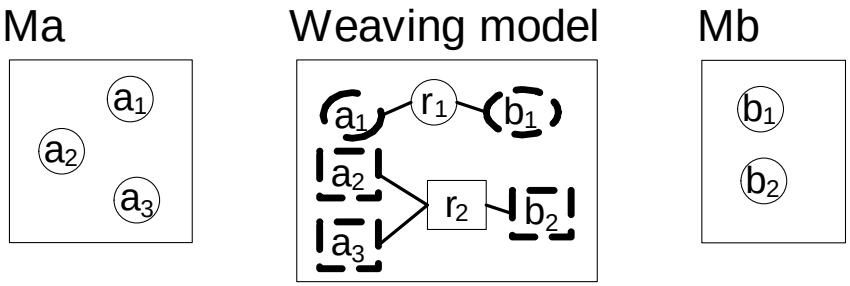


Figure 2.8: Weaving model

```
7  abstract class WModel extends WElement {
8   reference ownedElement[*] ordered container : WElement oppositeOf model;
9   reference wovenModel[1-*] container : WModelRef;
10 }
11
12 abstract class WRef extends WElement {
13  attribute ref : String;
14 }
15
16 abstract class WModelRef extends WRef {
17  reference ownedElementRef[*] container : WElementRef oppositeOf modelRef;
18 }
19
20 abstract class WElementRef extends WRef {
21  reference modelRef : WModelRef oppositeOf ownedElementRef;
22 }
23
24 abstract class WAssociation extends WElement {
25  reference associationEnd[1-*] container : WAssociationEnd oppositeOf association;
26 }
27
28 abstract class WAssociationEnd extends WElement {
29  reference association : WAssociation oppositeOf associationEnd;
30  reference relatedLink : WLink;
31 }
32
33 abstract class WLink extends WElement {
34  reference child[*] ordered container : WLink oppositeOf parent;
35  reference parent : WLink oppositeOf child;
36  reference end[1-*] container : WLinkEnd oppositeOf link;
37 }
38
39 abstract class WLinkEnd extends WElement {
40  reference link : WLink oppositeOf end;
41  reference element : WElementRef;
42 }
```

[2] shows how weaving models can address data integration. For the sake of further illustration, we comment a concrete use case. It focuses on interoperability of different kinds of sources (e.g., databases, files, tools). The use case presents how to use:

- **Metamodels** to represent data sources.
- **Weaving models** for capturing different kinds of semantic relationships required in data interoperability, e.g., concatenation of database schema properties, `name = firstName + '_' + lastName`).
- **HOTs** for producing transformations from weaving models.

Listing. 2.6 shows how `WLink` has been extended to support concatenation, i.e., the `Concatenation` class. `Source` refers to the set of properties to be concatenated (e.g., `firstName` and `lastName`). `Separator` represents the string used to join the properties (e.g., '_'). `Target` indicates to what property the concatenation result is assigned (e.g., `name`).

Listing 2.6: Excerpt of the AMW metamodel extension for data interoperability

```
class Concatenation extends WLink {
 reference source[*] ordered container : WLinkEnd;
 reference target container : WLinkEnd;
 attribute separator : String;
}
```

Listing. 2.7 gives an excerpt of the HOT used in the data interoperability use case. The HOT takes as input a weaving model and the referenced metamodels, and yields as output an ATL transformation. Listing. 2.7 focuses on the creation of ATL bindings from concatenation links.

Listing 2.7: Excerpt of a HOT for data interoperability

```
rule ConcatenationBinding {
 from
  amw : AMW!Concatenation
 to
  atl : ATL!Binding (
   propertyName <- amw.target.getReferredElement().name
  )
 do {
  atl.value <- thisModule.CreateConcat(amw, true);
 }
}
```

2.3.4 Textual Concrete Syntax

The TCS component is devoted to implementation of DSLs in the modeling context [19]. TCS enables the translation from text-based DSL programs to their corresponding model representation and vice versa. The start point of a DSL development with TCS is to develop a KM3 metamodel containing the domain concepts. Then, one specifies the concrete syntax by using TCS notations. In this step, one basically associates syntactic elements (e.g., keywords, special symbols) to metamodel concepts. From the KM3 metamodel and the concrete syntax, TCS generates three entities:

- An annotated grammar in ANTLR [16].
- An extractor that creates textual representation from models.
- And, an injector doing the opposite to the extractor, i.e., translation of models into textual programs.

2.3.5 AtlanMod MegaModel Management

The AM3 tool allows efficient management of all the MDE artifacts produced by a software development process (e.g., models, metamodels, metametamodels, model transformations,

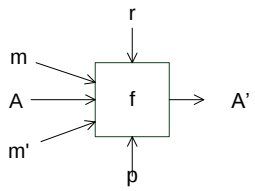


Figure 2.9: Matching algorithm (Adapted from [12])

weaving models, transformation chains) [23]. The main AM3 artifact is called *megamodel*. A megamodel is a specific kind of model whose elements represent models themselves as well as relationships between such models. For instance, a megamodel can represent the model transformations associated to a complex MDE system. Here, a megamodel has an understanding purpose, for example, one knows what are the models consumed and produced by a transformation and their interactions with other transformations. Megamodels conforms to a metamodel.

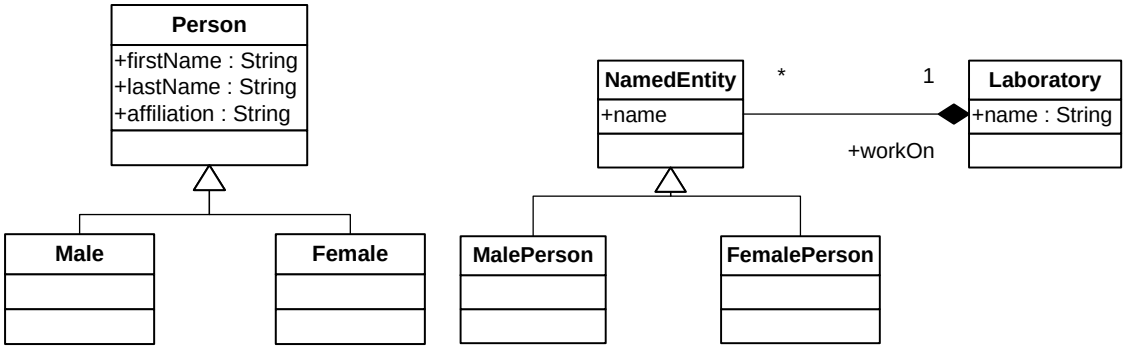
2.4 Model matching

Matching is the more longstanding solution to the heterogeneity problem. In a broad sense, the matching operation establishes mappings between two formalisms (e.g., metamodels, ontologies, database schemas) or formalism instances (e.g., models, individuals, databases). Here we refer to model matching, i.e., discovery of mappings between two metamodels or models. Since other formalisms can be translated into the modeling context, model matching is generic enough to cover ontology or schema matching. Manually finding of mappings is labor intensive, researchers have thoroughly studied how to (semi)automate the task. According to [61], the model matching operation can be separated in three phases: representation, calculation, and visualization.

The thesis focuses on *mapping calculation* and benefits from the work proposed in [2] to address the remaining matching phases. The calculation phase concerns algorithms able to establish mappings in a (semi)automatic way. As shown in Fig. 2.9, a matching algorithm f takes as input two models (m and m'), an initial mapping (A), parameters (p), and resources (r), and yields as output a new mapping (A'). Notice A can be empty or contain initial correspondences (either established by the user or pre-calculated in a previous matching process).

The following aspects characterize matching algorithms (the next sections will describe these aspects in detail):

- *Input* refers to what kinds of models a certain algorithm can match.
- *Output* alludes to the notation for expressing mappings.
- *Matching algorithm blocks* elaborates on how algorithms are implemented.
- *Evaluation* calls attention to the need for assessing the quality of matching results.



(a) MM1, families metamodel (b) MM2, persons metamodel

Figure 2.10: Families 2 Persons metamodels

Illustrating example

We have chosen an example simple but interesting enough to illustrate the aspects involved in the calculation phase. The example has been inspired from the *Families 2 Persons* use case[3]. Fig. 2.10 shows the metamodels involved in the example (*MM1* and *MM2*).

2.4.1 Input

This criterion classifies a model matching algorithm in regard to its input:

- **An M2-to-M2 matching algorithm** takes as input pairs of metamodels and is written in terms of metametamodel concepts. For example, to match KM3 metamodels, it is necessary to have an M2-to-M2 algorithm comparing the `Class` concept.

- **An M1-to-M1 matching algorithm** takes as input a given pair of models and involves the concepts of its corresponding pair of metamodels. For instance, to match models conforming to the Fig. 2.10 metamodels, an M1-to-M1 algorithm comparing the classes `Person` and `NamedEntity` is needed.

2.4.2 Output

Given two input models m and m', a matching algorithm yields as output a set of mappings (also referred to an alignment). A simple mapping (or correspondence) relates an element of m to an element of m'. A mapping has a similarity value which indicates its plausibility [62]. A similarity value can be discrete (i.e., 1 or 0, true or false) or continuous (i.e., infinite number of values between 0 and 1). Mappings can have an endogenous or exogenous notation. Endogenous means that the output has the same technical space of the input. Exogenous indicates the opposite.

In a mapping, one or more elements of the first model may be related with one or more elements of the second model, resulting in different cardinalities 1:1, m:1, 1:n, and

[3]http://www.eclipse.org/m2m/atl/basicExamples_Patterns/

m:n [62]. Note that one can decorate mappings to represent relations needed in an application domain, for example, `name = concat(firstName, lastName)`, it is a n:1 mapping decorated with *concat* to indicate a data integration operation.

2.4.3 Matching algorithm blocks

From this subsection, we will use the terms *matcher*, *matching technique* and *matching method* to refer to matching heuristic.

2.4.3.1 Individual matcher versus combining matcher

Rahm et al. [14] presents a taxonomy to classify schema matchers with regard to what kind of information is exploited:

- *Individual matcher* approaches compute mappings based on a single matching criterion.
  - *Schema-only based*
    - *Element-level* considers only an element at the finest level of granularity.
      - *Linguistic-based* uses labels and text to find similar elements.
      - *Constraint-based* determines the similarity of elements by regarding data types, ranges, uniqueness, optionality, relationship types, cardinalities, etc.
    - *Structure-level* takes into account combinations of elements that appear together in a structure.
  - *Instance-based* infers the similarity of elements taking into account the similarity of sets of instances. Instances can be compared as follows:
    - *Constraint-based* computes some statistics (e.g., mean, variance, etc.) about property values found in instances (e.g., size, weight). The hypothesis is that these measures should be the same for two equivalent properties.
    - *Linguistic-based* extracts keywords based on the relative frequencies of words and combination of words present in instances. Then, the matcher compares keywords and concludes what property is the better match for another.
- *Combining matcher* approaches integrate individual matchers. They can be done in two ways:
  - *Hybrid matchers* directly combine several matching approaches to determinate match candidates based on multiple criteria. Hybrid matchers often have a better performance than composites. Their drawback is the reduction of reuse capabilities.
  - *Composite matchers* combine the results of independently executed matchers, including hybrid matchers. Selection, execution order, and combination of matchers can be done either automatically or manually (by a human user).

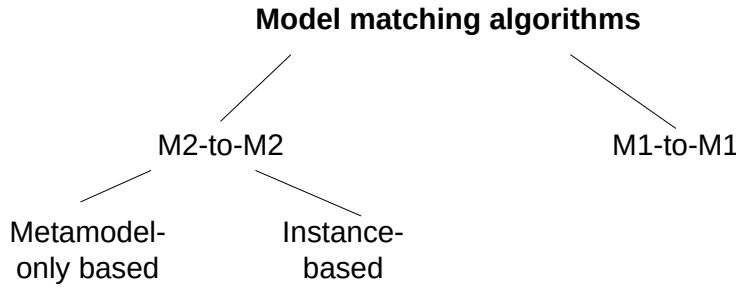


Figure 2.11: Classification of model matching algorithms

Some approaches plug machine learning methods on top of individual or combining matchers. The idea is to derive mappings from matching data (e.g., mappings, parameters). Such approaches operate in two phases: *training* and *matching*. In the training phase, data is manually provided by users, or collected from previous matching operations. The second phase matches ontologies or schemas using the gathered data. Examples of matching approaches using learning methods are: bayes learning [63], WHIRL learner [64], stacked generalization, neural networks [65][66][67], and decision trees [68].

[14] states that an individual matcher is unlikely to achieve as many good match candidates as a combining matcher. That is why the research community tends to contribute combining matchers [69]. Our algorithms belong to this category as well. By extrapolating Rahm's combining matchers into MDE, we itemize the classification given in Section 2.4.1 (see Fig. 2.11):

- **An M2-to-M2 algorithm** can be:
  - *Metamodel-only based*, takes as input a given pair of metamodels and discovers mappings by exploiting information available in the metamodels. This category corresponds to schema-only based matchers in Rahm's taxonomy.
  - *Instance-based*, its purpose is to match pairs of metamodels, to this end, the matcher judges models conforming to the input metamodels. This item is similar to Rahm's instance-based matchers.
  - *Metamodel-only based* and *instance-based* at once.
- **An M1-to-M1 algorithm** discovers mappings between pair of models. Rahm's taxonomy does not cover this category, however we have included it because M1-to-M1 matching is essential in MDE.

2.4.3.2 Blocks

Independently of category, M2-to-M2 or M1-to-M1, a model matching algorithm can involve the following blocks (take a look to Fig. 2.12):

- Normalize data representation.
- Search model elements.

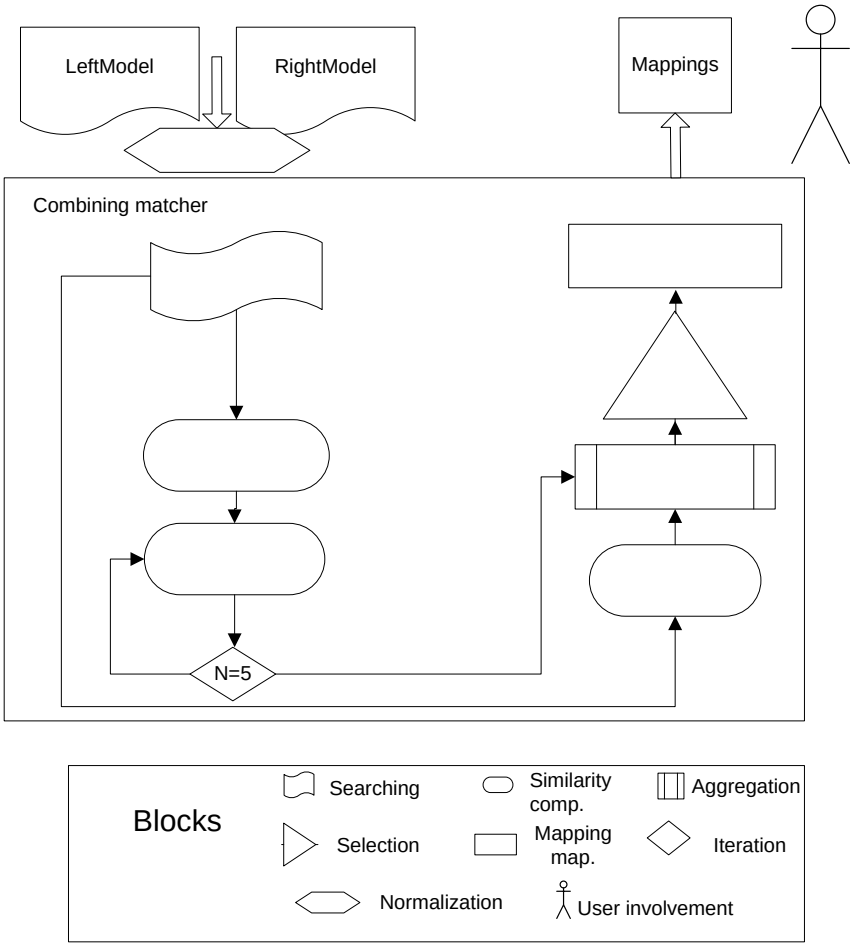


Figure 2.12: Blocks of a model matching algorithm

- Compute similarity values.
- Aggregate similarity values.
- Select mappings.
- Iterate for refining similarity values.
- Process mappings according to applications needs.
- Involve user.

There are different ways of implementing the blocks above, we present a compilation of them. The compilation is not extensive, its purpose is to show matching blocks variability. Please refer to [13][70] for a full compilation.

Normalization This block homogenizes data contained in models. There exist two kinds of normalization:

1. **Label-based.** The normalization reduces a text to a standard form based on its morphology (form of the words), syntax (grammatical structure of the text), or semantic (one aspect of the words, e.g., their synonyms). For example, let us consider the comment `it saves the annotation name`. The standard form could be: `it saves the name of annotation` (syntax) or `it saves the notation name` (semantic).

2. **Structure-based.** Besides the phase that bridges the gap between technical spaces, some approaches execute a structure-based normalization. The purpose is to make models conform to a generic format (typically a graph or a tree). Section 4.2.6 explains this in detail.

Searching This step chooses which mappings the subsequent matching blocks have to consider [70]. The most common methods for searching model elements are:

1. **Cartesian product.** Derives mappings by joining all elements of the first model with all elements of the second model.

2. **Fragment-based.** In turn, joins only parts of the models, for example, elements conforming to the same metamodel type.

Similarity computation This step computes similarity values by exploiting a kind of information.

Linguistic-based

- *String-based.* Labels are considered strings. The similarity is based on the structure of strings (i.e., the set of characters).

  - *String Equality.* If strings are identical, then similarity value is 1, otherwise is 0.
  - *Affix.* Evaluates if two strings have common affixes (i.e., prefixes and suffixes).
  - *SoundEx.* Computes the phonetic similarity between labels by taking into account their corresponding SoundEx index. SoundEx is an algorithm indexing labels by sound as pronounced in English [71] (pag. 392).
  - *N-gram.* Compares the number of common n-grams (i.e., sequences of n characters) between two strings. For example, the *di*-grams for `type` are: `ty`, `yp`, `pe`.
  - *Edit-distance.* Measures similarity values based on the edition operations that should be applied to a string to obtain another: more edition operations, lower similarity value. Some edit-distance implementations are proposed in [72] and [73].

  - *Path comparison.* Concatenates names of an element by taking into account its relationships with other elements (inheritance, association, or containment). For example, the `workOn` reference can be identified as `NamedEntity:workOn`, because the `NamedEntity` class contains `workOn`. Two elements are equivalent if their corresponding concatenations are equal.
  - *Token-based distance.* Considers a string as a set of words (a bag) and applies several similarity measures to it. For example, a common applied measure associates a vector to every bag, a word is a dimension, and the number of its occurrences in the bag is a position. Once vectors have been computed, usual metrics (e.g, Euclidean distance) calculate similarity.

- *Meaning-based.* Labels are considered as terms or texts with meaning. The similarity assessment is based on meanings and meaning relations.

  - *Extrinsic meaning.* Finds terms associated to labels and compare such terms. It looks for terms in external resources such as *Multi-lingual lexicons* and *thesauris*. A multi-lingual lexicon provides equivalent terms in several languages. For instance, given the English term `Person`, the lexicon provides `Persona` and `Personne`. That is, the corresponding terms of `Person` in Spanish or French. A thesauri provides sets of terms which are hypernym, hyponym[4], synonym, or antonym of a given term. For example, given `Person`, WordNet [75] provides the synonyms `Individual` and `Someone`.
  - *Intrinsic meaning.* Normalizes terms and compares them by applying string-based heuristics.

Constraint-based

- *Keys.* Compares XMI identifiers associated to model elements.

- *Datatype.* Compares compatibility between data types. The similarity is maximal when the data types are the same, lower when they are compatible (for instance, because integer can be casted into float they are compatible), and the lowest when they are not compatible. This method can not be used in isolation.

- *Domain.* Compares the domains of properties, for example the `age` property whose domain is [6 12].

- *Cardinalities.* Compares compatibility between cardinalities based on a table look-up. An example of such a table is given in [76].

[4] A hyponym is a word whose semantic field is included within that of another word, its hypernym. For example, scarlet, vermilion, and carmine are all hyponyms of red (their hypernym) [74].

Structure-level

- *Relational structure.* Computes similarity values based on the relations between model elements.
  - *Taxonomic structure.* Compares classes based on hierarchical relations: *superclassOf* or *subclassOf*. There are two variations:
    - *Indirect.* Computes similarity based on the taxonomic structure (i.e., hierarchy) of an external resource (e.g., WordNet). The method compares model elements to such a hierarchy.
    - *Direct.* Computes similarity based on the taxonomic structure of the metamodels themselves.
  - *Relation.* A concrete implementation is the *Similarity Flooding* (SF) algorithm [77]. SF assumes that two elements are similar when their adjacent elements are similar too. The SF algorithm executes two steps. The first step associates two mappings (*m1* and *m2*) if there is a semantic relationship between them. The second step propagates the similarity value from *m1* to *m2* because of the association. In our motivating example, the first step associates `(Person, NamedEntity)` and `(firstName, name)` because `Person` of *MM1* contains the `firstName` attribute and `NamedEntity` of *MM2* contains the `name` attribute. Then, the second step propagates the similarity value from `(Person, NamedEntity)` to `(firstName, name)`.

Instance-based These techniques can be combined in M2-to-M2 matching algorithms if models conforming to the input pair of metamodels are available. The techniques assume that two classes are similar if such classes share the same set of model elements.

- *Statistical approach.* Calculates statistics about the metamodel properties based on the models (maximun, minimun, mean, variance, existance of null, values, existence of decimals, scale, precision, grouping, and number of segments). The heuristic is practical when a metamodel property represents numerical values (e.g., real). For instance, the heuristic can deduce that `salary` and `compensation` are similar because the variations, computed from the model elements conforming to these properties, are similar.
- *Similarity-based extension comparison.* Two classes are similar if the model elements conforming to such classes are similar too.

Aggregation The aggregation step integrates the similarity scores produced by the previous block in order to form a unified score. Some ways of aggregating similarity values follow:

- **Max.** Selects the highest similarity value among all the values returned by similarity heuristics.
- **Min.** In turn, selects the lowest similarity value.

- **Weighted sum.** Multiplies each similarity value by a weight and aggregates the results into a single value.

Selection The previous steps may have created unwanted mappings (i.e., mappings with low similarities). The selection step chooses the mappings whose similarity values satisfy a criterion. Some selection step variants are:

- **MaxN.** Selects n mappings having the highest similarity values. For example, given the following set of mappings `(Person, NamedEntity, 0.09)`, `(Person, FemalePerson, 0.5)`, `(Male, NamedEntity, 0.18)`, and `(Male, MalePerson, 0.4)`, and $n = 2$, MaxN selects `(Person, FemalePerson, 0.5)`, and `(Male, MalePerson, 0.4)`.
- **Threshold-based.** Selects mappings whose similarity values are higher than a threshold.
- **Strengthening and weakening.** Defines functions taking similarity values and returning a value between 0 and 1. A function filters mappings whose similarity values are evaluated to 0.

Iteration The iteration block allows the execution of the same matching process n times given a condition. The output of an iteration can be the input of a next iteration.

1. **Manual.** It is the user who decides if the next iteration starts or not. The condition is to get a satisfactory matching result.
2. **Automatic.** The block automatically aborts the loop when a condition is satisfied. A condition can be not to exceed a maximal number of iterations.

Mapping manipulation In general, this step yields mappings needed in a given application domain. There are application domains more generic (e.g., merging, diff) than others (e.g., data integration, co-evolution). Below we give a brief description of some application domains:

1. **Merging.** Combines information from several models into a single model [78]. A way of merging is to match model elements that describe the same concepts in the diverse models. One creates a new model from the matching elements. Such a model is an integrated view of the concepts.
2. **Diff.** Returns a sequence of edit actions needed to get one model from another [78]. The diff problem has been thoroughly studied in many contexts. For example, the Unix diff compares two text files [79]. It reports a minimal list of line-based changes. This list may be used to bring either file into agreement with the other (i.e., patching the files). Another example is the SiDiff tool which computes a diff between two (graph-based) diagrams[5]. The differences between the graphs are computed

[5] http://pi.informatik.uni-siegen.de/Projekte/sidiff/

according to their structure. The tools may display the differences by using different colors. Model matching can leverage the diff operation; the matching step figures out the equivalences between models first, then diff creates the corresponding delta from the previous result.

3. **Integration.** It is technique of accessing information available in different models in a uniform way. The interoperability between models is achieved through the execution of transformations. The matching operation is of help here; once the users validate the correspondences identified by means of matching, they may derive integration transformations.

4. **Co-evolution.** Brings a model into agreement with a new metamodel (which is an evolved version of its former metamodel). A solution is to perform the diff operation to discover the changes introduced in the new metamodel version, and to derive an migrating transformation.

User involvement Users can be involved in a matching process as follows:

1. by providing initial mappings, mismatches, or synonyms.

2. by combining heuristics.

3. by providing feedback to the process in order for it to adapt the results.

4. by indicating threshold, weights, or constraints.

Tools are necessary to facilitate user involvement, for example, a graphical (referred as *GUI*) or textual (referred as *TUI*) user interface to manipulate mappings or combine heuristics.

2.4.4 Evaluation

A crucial issue in matching is to evaluate the results. Fig. 2.13 illustrates a classical matching evaluation. This aggregates the c block to Fig. 2.9. The c block calculates matching metrics (M) by comparing R to each A', where R is a reference mapping (or *gold standard*).

The most prominent metrics to assess matching accuracy are precision, recall, and fscore from information retrieval field [13].

Precision (P), recall (R), and fscore (F) are defined in terms of R and A'.

Precision measures the ratio of found mappings that are actually correct.

$$Precision(R, A') = \frac{R \cap A'}{|A'|} \tag{2.1}$$

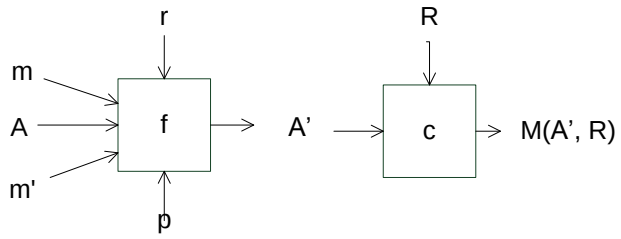


Figure 2.13: Matching algorithm evaluation (adapted from [12])

Recall measures the ratio of correct mappings found in comparison to the total number of correct existing mappings.

$$Recall(R, A') = \frac{|R \cap A'|}{|R|} \tag{2.2}$$

Fscore represents the harmonic mean of precision and recall. For instance, a precision and recall of 0.5 leads to a fscore of 0.5 indicating that half of all correct mappings were found and half of all found mappings are correct.

$$Fscore(R, A') = \frac{2 * Recall(R, A') * Precision(R, A')}{Recall(R, A') + Precision(R, A')} \tag{2.3}$$

Test cases quality gives credibility to evaluation. Test cases quality depends on the features of m, m', and A', e.g., size, variety of domains represented by the models. Reference mappings can be defined from scratch (e.g., by a panel of experts) or derived from existing software artifacts (e.g., from computed mappings which have been refined by experts).

2.5 Summary

This chapter has shown the context of the thesis, i.e., MDE, DSLs, and matching. We revisited the central MDE concepts. Model matching is an operation computing mappings between models. We have determined the scope of the thesis: *mapping calculation*, and we have described its main aspects.

Chapter 3

A survey of matching approaches and problem statement

In this chapter we present how a set of combining matching approaches supports the calculation phase, in particular, the aspects introduced in Section 2.4. We have classified the approaches by research community: ontologies (and database schemas), and models. Whereas our list of schema/ontology approaches is not exhaustive, we mention all approaches we found in the context of models[1]. Based on what these approaches do, we have defined issues on matching calculation. These issues draw the thesis scope.

3.1 Ontology-based and schema-based approaches

3.1.1 Coma++

Coma++ is an ontology (and schema) matching tool[2]. It provides a extensible library of Java-based matching algorithms, a framework for combining obtained results, and a platform for the evaluation of the effectiveness of the different matchers. The matching operation is described as a workflow that can be graphically edited and processed. Users can control the workflow execution in a stepwise manner and change execution parameters. Also, they can approve matches and mismatches. Coma++ provides two kinds of matchers: hybrid and composite. A Coma++ composite matcher may consist of:

- *Schema-only based matchers* (provided by Do [62]).
  - *Name* considers element names.
  - *Comment* compares element comments.
  - *Type* matches elements based on their data type.
  - *Path* computes similarities between two elements by comparing their complete name paths.

[1]The reason is that there already exist quite good surveys about the former approaches that readers may want to consider [70][13].

[2]http://dbs.uni-leipzig.de/Research/coma.html

  - *Children, Leaves, Parents,* or *Siblings* derive the similarity between elements from the similarities between neighbor elements, e.g., Children.
  - *Statistics* uses the Euclidean distance function to compute the similarity between structural statistics, which are determined for single schema elements using a feature vector uniformly capturing the number of children, parents, leaves, etc.

- *Instance-based matcher* (contributed by Engmann et al. [80]).
  - *Constraint-based* describes patterns that the instance data must satisfy. The pattern can be expressed in terms or letters, numbers or special characters.
  - *Content-based* determines the similarity of two elements by executing a pairwise comparison of instance values using a similarity function (typically a string similarity function). The performance is lower than the performance of the constraint-based matcher.

3.1.2 Semap

Semap [81] is a system that finds semantic relationships (i.e., not only simple mappings but also *has-a*, *is-a*, *associates* relationships) between schemas. The Semap architecture has three components:

- **Schema matcher**. Consists of a set of basic matchers. Every basic matcher looks at a particular model aspect:
  1. *Label* evaluates the syntactic similarity of labels (names).
  2. *Sense* compares the meanings that WordNet [75] gives to the labels.
  3. *Type* determines the similarity of schema elements based on data types.
  4. *Data instances* considers the format and distribution of information stored at instance level.
  5. *Structure* implements the Similarity Flooding algorithm.

  The matcher assigns similarity values and saves traces about how the mappings have been generated (this traces are used by the mapping assembler to discover generic relationships that schema matchers do not find).

- **Match selector**. Searches for the best global matchings. It applies a set of domain constraints (they are establish by the user) and a statistical model to select a subset of candidate matches. User interaction can greatly improve prediction accuracy. Semap applies active learning to identify the points in selecting the matches where user interaction is maximally useful.

- **Mapping assembler**. Selects an optimal set of mappings, identifies the relationships implicit in the selected matches, and assembles these matches together to form a final, generic semantic set of mappings. It identifies the relationships by using rules.

3.1.3 Learning Source Descriptions (LSD)

LSD [82] finds mappings between schemas. The process is as follows:

- **Train learners.** The user provides a few semantic mappings. LSD computes weights for each learner (i.e., matching heuristic) by performing least-squares linear regression on the dataset. The weights depends on what accuracy the learners provide. A learner has a high accuracy, when its results correspond to user-provided mappings. LSD trusts learners which have the highest accuracy. Note that the learners are applied to schema fragments.

- **Match.** Having the weights for each learner, LSD applies the learners to the whole schemas. [82] reports the use of three learners: *Name*, *Content*, and *Naive Bayes*. *Name* computes similarity between schema elements using the edit distance of their names. *Content* uses edit distance algorithms to find out similarity between schema data instances, and then propagates such a similarity to the corresponding schema elements. *Naive Bayes*, in turn, discovers similarity between schema elements by taking into account the tokens composing their names.

- **Exploit constraints.** Domain constraints are applied on the probable mappings to filter them. Finally, user feedback improves matching accuracy.

3.1.4 MAFRA

MAFRA [83] is a system for bringing instances of a source ontology into agreement with a target ontology. The idea is to write rules for matching the instances in terms of ontologies (concepts and properties). The output mappings conform to the target ontology. This ontology has two main concepts: *SemanticBridge* (which represents a mapping) and *Service* (which represents matching techniques). The inclusion of the latter concept is precisely its main drawback, the system does not separate aspects of mapping from services to obtain them. It makes computed mappings difficult to translate into other formats, i.e., difficult to be reused.

3.1.5 APFEL

APFEL [68] combines matching learning algorithms, user validation, schema-only based and instance-based methods. APFEL aligns ontologies by following four steps:

1. Apply a naive strategy that gives an initial set of alignments. The strategy combines several heuristics. Each of them has a weight and judges an ontology feature.

2. Users select correct alignments.

3. Based on such alignments, a machine learning algorithm suggests weights and heuristics giving the best results.

4. APFEL proposes a new strategy tuned with the weights and heuristics suggested above. This strategy can be further improved by executing the process iteratively.

APFEL authors pursued 7 strategies to two scenarios (i.e., string-based, feature similarity combinations and aggregation, three versions of the decision tree learner, neural networks, and support vector machine). They found that APFEL is as competitive as other ontology matching systems. However, APFEL highly depends on chosen machine learning algorithms and training data.

3.1.6 GeromeSuite

GeromeSuite allows the definition of composite matchers [84]. It provides three kinds of matching blocks: similarity, selection, and aggregation. GeromeSuite furnishes import and export operators for SQL-DDL, XML schema, and OWL. The m and m' inputs have to conform to the Gerome internal representation. GeromeSuite gives a GUI for configuring strategies. GeromeSuite matching operators are implemented in Java. GeromeSuite provides three kinds of mappings:

1. *Informal*: simple mappings.

2. *Intensional*: complex mappings describing a relation between sets of elements (i.e., disjointness, equality, and subset).

3. *Extensional*: complex mappings representing a query for data translation.

3.1.7 An API for ontology alignment

This approach proposes a format for expressing alignments in RDF and a Java API to manipulate it [85]. In particular, the API provides baseline matching algorithms and functionalities for refinement/evaluation of alignments and transformation generation. The Ontology Alignment Evaluation Initiative (OAEI) [12] uses the API evaluation functionalities.

3.2 Model-based approaches

This category groups approaches that have adopted the term *model* to refer to contemporary technical spaces such as MOF or Ecore.

3.2.1 Kompose

This is a framework to compose models conforming to the same metamodel [86]. The composition implies matching and merging. Whereas the merging algorithm is generic, one needs to define a new matching algorithm for each metamodel to be composed. The algorithm specifies the metamodel types to be matched, and associates to each concrete type an implementation of the operator *equals*. This operator determinates whether two model elements, conforming to a given type, are equal or not. Kompose is based on Kermeta [44].

3.2.2 Epsilon Comparison Language (ECL)

ECL is a DSL that enables users to specify model matching algorithms [5]. The main ECL abstraction is *MatchRule* which specifies two parameters (specifying the types of matching elements), the *guard part* (that limits the applicability of the rule), the *compare part* (that compares the pairs of elements and decides if they match or not), and the *do part* (performing additional required actions). The result of comparing two models with ECL is a set of mappings with boolean values that indicate if the referred elements have been found to be mappings or not. ECL has been implemented atop the Epsilon platform[3] and therefore it inherits its syntax and features (e.g., rule inheritance, invoking native Java code).

3.2.3 EMF Compare

EMF Compare reports differences between two pairs of models conforming to the same metamodel[4]. The tool performs two steps, i.e., matching and diff. Matching strategies are hard-coded into the tool in Java. According to [87], EMF Compare compares model attributes by judging data types, labels, and IDs. The two strengths of EMF Compare are its Graphical User Interface and performance.

3.2.4 Generic and Useful Model Matcher (Gumm)

Gumm is a tool that aligns models [88]. The tool represents models by means of labeled (and directed) graphs. Gumm provides an API to create/manipulate graphs. An available algorithm is the *Similarity Flooding* [77]. It seems an early Gumm version has been used for computing mappings and deriving model transformations [3].

3.2.5 SmartMatcher

SmartMatcher uses matching techniques for transformation generation [4]. Its main feature is to train candidate mappings by using data instances. For example, to generate a transformation whose input and output metamodels are MMA and MMB, SmartMatch proceeds as follows:

1. Build training models by hand, e.g., MA and MB, conforming to MMA and MMB, respectively.

2. Match MMA and MMB by using third-party ontology systems, e.g., Coma++ [62].

3. Generate a transformation MT from the obtained mappings.

4. Execute MT.

[3] www.eclipse.org/gmt/epsilon/
[4] http://wiki.eclipse.org/index.php/EMF_Compare

5. Compare MB' (yielded by MT) and MB. The approach compares attributes values, and model structure (i.e., references). A function, called *fitness*, verifies if the elements of MB' and MB are equal. If the answer is positive, then the mappings from which the transformation has been generated is accepted.

6. Train incorrect mappings by repeating the process from the step 1.

All SmartMatcher functionalities are implemented in Java. SmartMatcher needs the definition of training models, it may imply more work than mapping discovery from scratch.

3.2.6 MatchBox

Like SmartMatcher, MatchBox [89] computes M2-to-M2 mappings. MatchBox is developed on top of the SAP Auto Mapping Core (AMC) which is based on Coma++ [62]. MatchBox executes the following steps:

1. Translate metamodels into a tree data representation. To build such a tree, metamodel properties are flattened in a concrete way.

2. Apply single matchers (i.e., *name*, *name path*, *children*, *sibling*, *parent*, and *leaf*).

3. Aggregate and combine their results.

3.3 Comparison of approaches

The next 7 tables position revisited approaches with respect to the criteria established at the very beginning of this chapter. The tables include the AMW tool referring to Marcos Didonet del Fabro's approach [90] which has been described in Chapter 1. A discussion follows each table. The rows represent the criteria. The columns mention the approaches arranged by community. The possible cell values follow:

- An 'x' indicates that a given approach supports a certain criterion.

- An empty space indicates the contrary.

- Text or numbers corresponding to the options or descriptions given in the criteria.

- A question mark (?) means that the approach documentation does not provide enough information to evaluate the criterion.

Appendix E contains 4 tables positioning our approach with respect to the same criteria.

3.3.1 Input

Criteria		Schema/Ontology -based approaches							Model-based approaches						
		Coma++	Semap	LSD	MAFRA	APFEL	GeromeSuite	INRIA Alignment API	Kompose	ECL	EMF Compare	Gumm	SmartMatcher	MatchBox	AMW
M2-to-M2	SQL-DDL						✔								
	XSD	✔	✔	✔			✔								
	OWL	✔				✔	✔	✔							
	MOF (XMI)														
	Ecore										✔		✔	✔	✔
	Others						Gerome					Graph		Tree	
M1-to-M1					✔				✔	✔	✔				

Table 3.1: Comparing related work with respect to the input criterion

Table. 3.1 depicts that most of the approaches provide M2-to-M2 matching algorithms. In particular, GeromeSuite covers the largest spectrum of technical spaces. GeromeSuite provides heuristics written in terms of its own metametamodel (called Gerome), and operators to translate SQL-DDL, XML Schema, or OWL models into Gerome. The operators are implemented declaratively using a rule-based approach (i.e., SWI-prolog).

On the other hand, only 4 approaches (MAFRA, ECL, Kompose, and EMF Compare) provide M1-to-M1 matching algorithms. In the case of MAFRA, Kompose, and ECL, given a new pair of models, developers have to implement a new algorithm. EMF Compare has a default algorithm that matches models conforming to the same metamodel. The algorithm mostly relies on linguistic-based similarity heuristics. When the metamodel, to which models conform to, does not have the *name* attribute, the algorithm gives low accuracy results. Recently, EMF Compare came out with a new feature which allows plugging customizable algorithms. Developers have to implement these algorithms with Java and Ecore.

3.3.2 Output

Criteria		Schema/Ontology based approach							Model-based approaches						
		Coma++	Semap	LSD	MAFRA	APFEL	GeromeSuite	Alignment API	Kompose	ECL	EMF Compare	Gumm	SmartMatcher	MatchBox	AMW
Similarity value	Discrete								✔	✔					
	Continuous	✔	✔	✔	✔	✔	✔	✔			✔	✔	✔	✔	✔
Notation	Endogenous	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Exogeneous														
Cardinality	1:1	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	1:n	✔			✔	✔	✔	✔			?				✔
	m:1	✔			✔	✔	✔	✔			?				✔
	m:n	✔				✔	✔	✔			?				✔
App. domain relationships			✔								✔				

Table 3.2: Comparing related work with respect to the output criterion

According to Table. 3.2, Kompose and ECL enable the specification of comparison algorithms that render discrete similarity values (0 or 1, equal or not equal). The rest of approaches instead apply the notion of probabilistic mappings (i.e., continuous similarity

values), which is closer to real world mappings. The revisited approaches produce endogenous outputs. For instance, if the approach matches ontologies, the matching result conforms to an ontology as well. Coma++, APFEL, the Alignment API, GeromeSuite, and AMW support all kinds of mapping cardinalities. At last, Semap and EMF Compare mark mappings with types needed in different applications domains, data integration and diff, respectively.

3.3.3 Matching algorithm blocks

Table. 3.3 and Table. 3.4 respectively compare the schema/ontology-based and model-based approaches in regard to the matching algorithm criteria. Below we discuss each block.

Criteria		Schema/Ontology -based approaches						
		Coma++	Semap	LSD	MAFRA	APFEL	GeromeSuite	Alignment API
Normalization	Label-based	?	?	?		?	?	✔
	Structure-based	✔	✔	✔			✔	
Searching	Cartesian product	✔	?	?		?	?	
	Fragment-based	✔	?	?	✔	?	?	✔
Similarity computation		✔	✔	✔	✔	✔	✔	✔
Aggregation		✔	✔	✔		✔	✔	✔
Selection		✔	✔	✔		✔	✔	✔
Iteration	Manual	✔	✔	✔				
	Automatic					✔		
Mapping manipulation		Merging, Diff	Mapping construction				Merging, Data integration	Merging, Data integration
User Involvement	Initial mappings	✔	✔	✔		✔		?
	Initial parameters, e.g., thresholds, weights, constraints	✔		✔		✔		✔
	Additional inputs, e.g, mismatches, synonyms	✔						✔
	Combination of heuristics	GUI		?		TUI	GUI	
	Mapping refinement	GUI	?	?	GUI	TUI		TUI

Table 3.3: Comparing schema/ontology-based approach with respect to the matching building blocks criterion

Criteria		Model-based approaches						
		Kompose	ECL	EMF Compare	Gumm	SmartMatcher	MatchBox	AMW
Normalization	Label-based			?	?	?	?	✔
	Structure-based				✔		✔	
Searching	Cartesian product				✔			
	Fragment-based	✔	✔	✔	✔	✔	✔	✔
Similarity computation		✔	✔	✔	✔	✔	✔	✔
Aggregation				✔	✔	✔	✔	✔
Selection				✔	✔	✔	✔	✔
Iteration	Manual							
	Automatic					✔		
Mapping manipulation		Merging		Diff	Data integration	Data integration		Data integration
User Involvement	Initial mappings					✔		
	Initial parameters, e.g., thresholds, weights, constraints							✔
	Additional inputs, e.g, mismatches, synonyms							
	Combination of heuristics							GUI
	Mapping refinement			GUI		?		GUI

Table 3.4: Comparing model-based approach with respect to the matching building blocks criterion

3.3.3.1 Normalization

The approaches documentation does not explicitly indicate what kind of label-based normalization is applied. However, the Alignment API and AMW code shows a morphological normalization, in particular, a case normalization. It converts each alphabetic character in the strings into its lower case counterpart [13]. Some XSD and OWL-based approaches (i.e., Coma++, Semap, LSD, and GeromeSuite) execute a structure-based normalization. The model-based approaches (except Gumm and MatchBox) do not execute this extra step, they directly operate on metamodels conforming to a standard metamodeling format (e.g., Ecore).

3.3.3.2 Searching

All the revisited approaches execute the searching step. However, from the documentation, it is not possible to determinate the kind of searching applied by Semap, LSD, APFEL, and GeromeSuite. Coma++ and Gumm provide both Cartesian product and fragment based. They spell out that a Cartesian product impacts matching algorithm performance. Coma++ and MatchBox select the elements with respect to a tree structure (e.g., Children, Leaves, etc.), Gumm exploits a graph structure. Alignment API hardcodes what kinds of elements are of interest, e.g., `Class`, `Property`, `Individuals`. MAFRA, ECL, Kompose, and AMW, in turn, choose elements by specifying metamodel types in an explicit way.

3.3.3.3 Similarity computation

Table. 3.5 classifies the approaches with respect to Rahm's taxonomy. About the type of combining matcher, the number of composite approaches corresponds more or less to the hybrids' one. Coma++ is the only approach that provides composite and hybrid matchers.

In addition, Table. 3.5 shows the number of individual matchers that can be combined in a composite matcher. Coma++ provides the largest library of individual matchers. GeromeSuite spells out to provide several element-level and structural level matchers. However, the authors do not exactly precise what are their functionalities. APFEL uses techniques taken from the original PROMPT tool [91]. SmartMatcher and MatchBox reuse (or have been inspired by) the Coma++ techniques.

Based on the approaches documentation, it is difficult to list the individual matchers that have been combined in hybrid matchers. The Alignment API furnishes 4 hybrid algorithms combining various element-level (above all String-based) and structure-level techniques. [5] cites an ECL algorithm matching models that represent trees. The algorithm embeds linguistic-based and structure-level comparison. According to [87], it seems EMF Compare implements 1 structure-level and 3 element-level techniques.

Criteria		Schema/Ontology based approaches							Model-based approaches						
		Coma++	Semap	LSD	MAFRA	APFEL	GeromeSuite	Alignment API	Kompose	ECL	EMF Compare	Gumm	SmartMatcher	MatchBox	AMW
Type of combining matcher	Composite	✓	✓	✓		✓	✓						✓	✓	✓
	Hybrid	✓			✓			✓	✓	✓	✓	✓			
Type of individual matcher	Element-based	3	3	1		Reuses PROMPT tool	?						Reuses Coma++	Inspired by Coma++	3
	Structure-based	6	1				?								1
	Instance-based	2	1	2			?								

Table 3.5: Comparing related work with respect to the similarity criterion

3.3.3.4 Aggregation and selection

In general, the revisited approaches use a *weighted sum* aggregation. Besides this technique, Coma++ reports 3 additional aggregation techniques, i.e., *Max*, *Min*, and a special case of *weighted sum* (see [62] pag. 69).

The approaches involving the selection step often use thresholding. In addition, LSD filters mappings with domain constraints. The high granularity of constraints suggests that their definition is a time-consuming task.

The tough part of aggregation and selection is to define weights and thresholds. Although this is not the focus of our work, we want to mention eTuner [69], an automatic learning approach for finding the best knobs (e.g., thresholds, formula coefficients). Lee et al. experiment eTuner on four matching systems, among them Coma++ and LSD.

3.3.3.5 Iteration

Five out of fourteen cited approaches draw the iteration as a matching process block. The iteration is mostly manual. For example, for each iteration, Coma++ enables the user to select the matchers to combine, and to accept or reject candidate mappings. Semap, in turn, prompts the user to disambiguate cases where its feedback is maximally useful. Semap iterates on the matching techniques identifying simple mappings, but not on the techniques extracting complex relationships. A new LSD iteration starts if the user is not happy with the results. Moreover, he or she can specify new domain constraints or manually match elements.

In contrast to the previously mentioned approaches (which ask the user for mapping error correction), SmartMatcher and APFEL provide an automatic iteration. SmartMatcher derives transformations from the computed mappings, and adjusts the matching algorithm by comparing the transformation output to an output built by the user. SmartMatcher adapts the mappings based on the identified differences, and starts a new iteration. APFEL stops a loop when no new alignments are proposed, or if a predefined number of iterations has been reached.

3.3.3.6 Mapping manipulation

The mapping manipulation block transforms simple mappings into complex mappings in order to fulfill an application domain request. Most of approaches use GPLs to encode the manipulation logic.

Coma++ and EMF Compare implement the Merge and Diff operators with Java. SeMap uses a rule-based language. The authors define rules that translate simple mappings into more semantically richer ones. To merge models, Kompose requires the implementation of a matching algorithm. Merge and Match operators are implemented with Kermeta [44].

Gumm and SmartMatcher state the use of mapping for generating transformations, however we have not found what transformation languages they support and how they perform the generation.

The Alignment API provides the notion of *visitor*. Each visitor implements the translation of mappings into a specific format, e.g., HTML, OWL axioms, XSLT, or SWRL. HTML simply displays mappings in a more friendly way than RDF. OWL axioms merge the concepts of two aligned ontologies. A SWRL or XSLT file performs data migration. GeromeSuite translates mappings into query languages such as XQuery and SQL. AMW, in turn, generates ATL transformations by using ATL HOTs.

3.3.3.7 User involvement

We have implicitly discussed this criterion throughout the previous sections. As mentioned in the iteration paragraph, Coma++, Semap, LSD, APFEL, and SmartMatcher allow the user to provide initial mappings as input. Coma++, LSD, APFEL, the Alignment API, GeromeSuite, and AMW allow indicating parameters without going to the kernel of the matching system but using an interface. Coma++, GeromeSuite, and AMW enable the combination of matching techniques by using a GUI. APFEL, in turn, provides a TUI. The Alignment API command-line interface has an option enabling users to choice a given matching algorithm. Users nonetheless have to go into Java code if they wish to understand how matching techniques have been combined. Finally, Coma++, GeromeSuite, MAFRA, APFEL, the Alignment API, EMF Compare, and AMW contribute TUIs or GUIs for mapping refinement.

3.3.4 Evaluation

Table. 3.6 and Table. 3.6 correspondingly compare the schema/ontology-based and model-based approach with respect to the evaluation criteria.

Criteria		Schema/Ontology -based approaches						
		Coma++	Semap	LSD	MAFRA	APFEL	GeromeSuite	Alignment API
Dataset (Metamodels)	Number of pairs	16	5	4	?	2	?	1
	Size	20 - 850	5 - 25	20 - 70	?	400-2100	?	?
	Domains	Purchase orders, E-business	Synthetic, Real state, Course Info	Real State, Time schedule, Faculting listing	?	Bibliographic metadata, Travel websites about Russia,	?	Bibliography
	Exec. Time (min.)	7-10	?	?	?	?	?	?
	Fscore	0.7 - 0.9	0.7 - 1.0	0.7 - 0.9	?	0.3 - 0.6	?	0.8
Determination of reference alignments	Experts	✔	✔	✔	✔	✔	✔	✔
	Existing software artifacts							

Table 3.6: Comparing schema/ontology-based approaches with respect to the evaluation criterion

Criteria		Model-based approaches						
		Kompose	ECL	EMF Compare	Gumm	SmartMatcher	MatchBox	AMW
Dataset (Metamodels)	Number of pairs	?	?	?	3	1	23	2
	Size	?	?	?		60	50-360	6300
	Domains	?	?	?	Ecore-Kemeta, Ecore-UML, Ecore-MinJava	UML 1.04 - UML 2.0	The ATL Zoo	Scade, Autosar
	Exec. Time (min.)	?	?	?	?	?	?	?
	Fscore	?	?	?	0.3 - 0.8	0.9	0.5	?
Determination of reference alignments	Experts	✔	✔	✔	✔	✔	✔	✔
	Existing software artifacts						✔	

Table 3.7: Comparing model-based approaches with respect to the evaluation criterion

MAFRA, Kompose, ECL, GeromeSuite, and EMF Compare do not give details about the datasets on which algorithms have been applied. The remaining systems evaluate their algorithms over small or large test cases. The approaches mostly cite validations over pairs of metamodels but not over models. In particular, Coma++ and MatchBox have the more extensive benchmarks in terms of pairs and metamodel size. In the schema/ontology context, the domains of tested pairs of metamodels overlap. Besides the purpose of evaluating matching algorithms over well-known benchmarks, a reason behind the overlapping may be the reference alignment cost. To reduce such a cost (or workload), the schema/ontology approaches have used common test cases (i.e., e-business, real state, course information, bibliographic metadata). Coma++, Semap, and LSD present the highest fscores. All the revisited approaches use the classical accuracy measures, i.e., precision, recall, and fscore.

Most of approaches use test cases where experts manually determinate reference alignments. Only MatchBox benefits from existing software artifacts (i.e., model transformations) to automatically derive gold standards.

3.4 Problem statement

As indicated in Section 2.4, the thesis focuses on the calculation phase, and benefits from the work proposed in [2] to represent and visualize mappings. Because no matching algorithm exists that is 100% accurate in calculating all mappings between all pairs of models, it is necessary to provide means to configurate algorithms that yield precise results as much as possible.

The previous sections mention how previous work addresses the calculation phase with respect to the criteria defined in Section 2.4. From the comparison, we have figured the following requirements or research issues:

1. The revisited approaches often provide M2-to-M2 algorithms coupled to a given metametamodel (e.g., Ecore, OWL). There is a need for M2-to-M2 matching algorithms covering a largest spectrum of metametamodels at once.

2. The comparison associated to the matching algorithm blocks criterion reveals that heuristics are mostly reused in M2-to-M2 matching algorithms. Is it possible to straightforwardly reuse these heuristics in M1-to-M1 algorithms too?

3. Table. 3.7 has more question symbols than other tables. This shows the lack of a systematic matching evaluation, above all, in the MDE context.

Two issues *reusability of matching heuristics* and *evaluation of customized algorithms* cover the items mentioned above. Let us elaborate on them.

3.4.1 Issues on reusability of matching heuristics

Coupling of matching heuristics to reference model Depending on the kind of a model matching algorithm, M2-to-M2 or M1-to-M1, its heuristics are written in terms of metametamodels or metamodels, respectively. Even though some heuristics compare very standard model features (e.g., `name`), these may be no longer applicable when the inputs conform to other reference model. To explain the problem in detail, let us recall two related works that implement M2-to-M2 matching algorithms with GPLs: the Alignment API [85] and AMW [2].

Fig. 3.1 shows an excerpt of a class diagram for the Alignment API. To implement a new algorithm, one creates a new Java class (e.g., `NameEqAlignment`) extending the `Similarity` interface. One implements a method for each type of element one wants to compare, e.g., `Class`, `Property`, `Individuals`. Suppose one wants to match KM3 metamodels using the Alignment API. Besides a prior stage translating KM3 metamodels into OWL, one would need to modify the `Similarity` interface and every dependent class (`NameEqAlignment`, `EditDistNameAlignment`). One would have to add methods comparing other types of elements relevant to KM3, e.g., `Package`. Thus, for each new metametamodel one may have to modify the API. As a result, one gets an API that tends to stray from the planned structure.

We have observed the same problem as one develops matching algorithms with general purpose model transformation languages (e.g., ATL [7]). Listing. 3.1 shows the excerpt

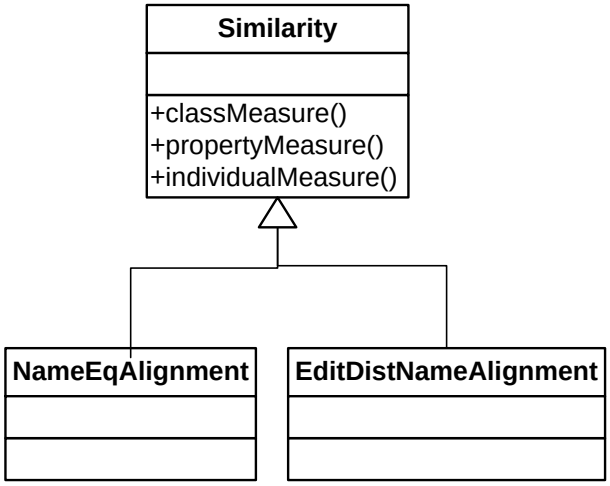


Figure 3.1: Excerpt of the Alignment API class diagram [85]

of an ATL matching transformation proposed in [2]. The rule matches two Ecore classes. Lines 3-5 establish the matching pattern, i.e., metamodel types (i.e., `EClass`) and condition. Line 7 expresses the output pattern. Since the matching pattern declares Ecore types, one can not directly use the rule to match (for example) KM3 metamodels. A way of reusing the rule is to perform a prior translation (for example from KM3 to Ecore). However, a translation program might fail or not exist at all.

Previous paragraphs illustrate the difficulty of reusing heuristic in M2-to-M2 matching algorithms. It is also the case for M1-to-M1 algorithms. An *ad-hoc* solution is to write the code for each new model representation. As a result, one gets multiple heuristics encompassing the same matching logic.

Listing 3.1: ATL matching transformation excerpt

```
1 rule Similarity {
2  from
3   r : RightMetamodel!EClass in RightModel
4   l : LeftMetamodel!EClass in LeftModel
5   (condition) -- for example, r.name = l.name
6  to
7   e : EqualMM!Equal
8 }
```

We have observed further issues about matching algorithm development. For example, when one executes an algorithm written with the Alignment API, one does not know what is exactly going on into the algorithm at the first glance; how do heuristics interact to deliver mappings?, what heuristic is taking what parameter?

On the other hand, looking further into matching transformations shows:

1. Repetitive code devoted to preserve information.

2. Verbosity of existing way of combining matching transformations (e.g., Ant tasks[5]).

There is a need for constructs that facilitate reasoning about matching process and factorize portions of code.

[5]http://wiki.eclipse.org/index.php/AM3_Ant_Tasks

3.4.2 Issues on matching algorithms evaluation

In the context of ontology (or schema) matching, there are two common ways of getting test cases for performing evaluation. Some approaches develop user studies where experts define reference alignment between pairs of models [92][62]. Other works use test cases defined in third-party evaluation initiatives (such as the OAEI [22]). Every year since 2004, the OAEI develops a campaign that involves three phases: preparation, execution, and evaluation. During the preparation phase, the OAEI defines a set of test cases to compare matching systems and algorithms on the same basis. A given test case includes a pair of ontologies (coming from different domains) and their corresponding reference alignment. In the execution phase, the participants use their algorithms to match test cases. Finally, the OAEI organizers check the results obtained in the evaluation phase.

In MDE, there is no evaluation initiative yet. Each matching approach establishes its own test cases and evaluation frameworks. Thus, we have observed the following issues:

Not enough test cases. Whereas it is relatively easy to find out pairs of meta(models), the availability of reference alignments is restricted. Most of the time one asks experts for defining reference alignments from scratch. Note this may be an expensive task.

Low evaluation efficiency. Evaluation is a time-consuming, tedious, and error-prone task.

Although ontology community has made efforts to standardize matching algorithm evaluation, i.e., definition of test cases, methodologies, and tools. We believe that the mentioned issues are relevant to ontology community too, in particular, the second one. For example, the OAEI reports not to have enough time to systematically validate all the matching results ([93], page 5).

3.5 Summary

This chapter surveyed matching approaches proposed in diverse communities: databases systems, ontologies, and models. It gave comparative tables that position the approaches with respect to the criteria defined in Section 2.4, i.e., input, output, matching algorithm, and evaluation. The tables showed that the approaches have made substantial progress on all the criteria. However, they are mostly restricted either to the schema/ontology context or to the modeling context. In addition, most of the approaches provide M2-to-M2 matching algorithms, a few ones support M1-to-M1 matching algorithms. Finally, ontology community has moved a step forward with respect to MDE community in the matter of evaluation. Ontology community has defined common dataset, methodologies, and tools facilitating evaluation. In contrast, each MDE matching approach uses their own datasets and *ad-hoc* methodologies. Thus, it is difficult to known about the real advantages and disadvantages of these approaches. These issues draw the thesis scope. Therefore, the thesis investigates how MDE can contribute to matching independent of technical context (i.e., matching heuristics able to take as input OWL, Ecore metamodels, etc.) and abstraction level (heuristics applicable to pairs of metamodels or models). We

have called that the capability of reusing matching heuristics. Moreover, the thesis tackles the deficiencies on (meta)model matching evaluation.

Chapter 4

The AtlanMod Matching Language

To address the issue on matching heuristic reusability, we propose a DSL-based solution instead of a GPL-based solution. According to [14], the matching operation has been investigated from the early 1980s. We promote the use of a DSL which gathers all the expertise gained over the last 30 years. The DSL has to capture a significant portion of the repetitive tasks that a expert needs to perform in order to produce an executable matching algorithm.

A first approximation to such a DSL is the AtlanMod Matching Language (AML) which is based on the MDE paradigm [25]. The development of AML has covered all the phases going from decision to deployment. Section 3.4 presents the motivations to deciding in favor of AML. In regard to the deployment, we have followed guidelines to make AML available for use in Eclipse.org. This chapter describes the remaining phases of the AML development, i.e., analysis, design, and implementation.

Section 2.2 mentions variants to carry out analysis, design, and implementation of DSLs. To develop AML, we have followed an informal analysis, a formal design (i.e., concrete and abstract syntaxes have been specified), and a hybrid implementation technique (i.e., embedding language and preprocessing).

4.1 Analysis: AML base concepts

Unlike hybrid matchers, whose combination of matching techniques is hard-coded, composite matchers allow the selection of constituent techniques [14]. Because composite matchers maximize reusability, we devote AML to composite matcher development. Below we summarize the terminology of composite matchers in a more or less abstract form.

Definition 3. *A composite matcher is an incremental process that takes two models $LeftModel$ and $RightModel$, along with optional domain knowledge (e.g., dictionaries, parameters, reference metamodels, previously computed mappings, user inputs, etc.), and produces mappings between the models. The matcher executes a heuristic in each step. Each heuristic takes mappings as input and gives mappings as output.*

Definition 4. *A mapping links elements of $LeftModel$ to elements of $RightModel$. A mapping has a similarity value (between 0 and 1) that represents how similar the linked elements are.*

Each heuristic involved in a composite matcher conforms to one of the following categories:

- **Creation** establishes mappings between the elements of *LeftModel*) and the elements of *RightModel* when these elements satisfy a condition. This category correspond to the searching block.
- **Similarity** computes a similarity value for each mapping prepared by the search heuristics. A function establishes the similarity values by comparing particular model aspects: labels, structures, and/or data instances (the latter has sense when the algorithm is M2-to-M2). In general, a given comparison is direct and/or indirect. Direct means only comparison between model elements. The indirect comparison separately examines model elements in relation to a third element from a domain knowledge resource (e.g., a reference metamodel).
- **Aggregation** combines similarity values by means of an expression. An expression often involves:
  - n, the number of heuristics providing mappings.
  - $\sigma(a_i, b_i)$ similarity value computed by the heuristic i
  - w_i weight or importance of the heuristic i, where $\sum_{i=1}^{n} w_i = 1$
- **Selection** selects mappings whose similarity values satisfy a condition.
- **User-defined** is devoted to mapping manipulation or to functionality beyond the heuristics mentioned above.

The listed heuristics correspond to some blocks illustrated in Fig. 2.12, a composite matcher may need the rest of blocks (i.e., normalization, iteration, and user involvement) as well.

4.2 Design: notations overlapping AML base concepts

4.2.1 Overview

Given a composite matcher $S1$, the AML functional components operate as follows (see Fig. 4.1):

- The **compiler** takes the $S1$ strategy and generates a chain of matching transformations (`Type`, `SF`, etc.) written in ATL and launched by an Ant script, each transformation:
  - Instruments a heuristic.
  - Takes as input a set of models: *LeftModel* and *RightModel*, an initial *equal* (mapping) model, and a set of additional models (e.g., a parameter model).

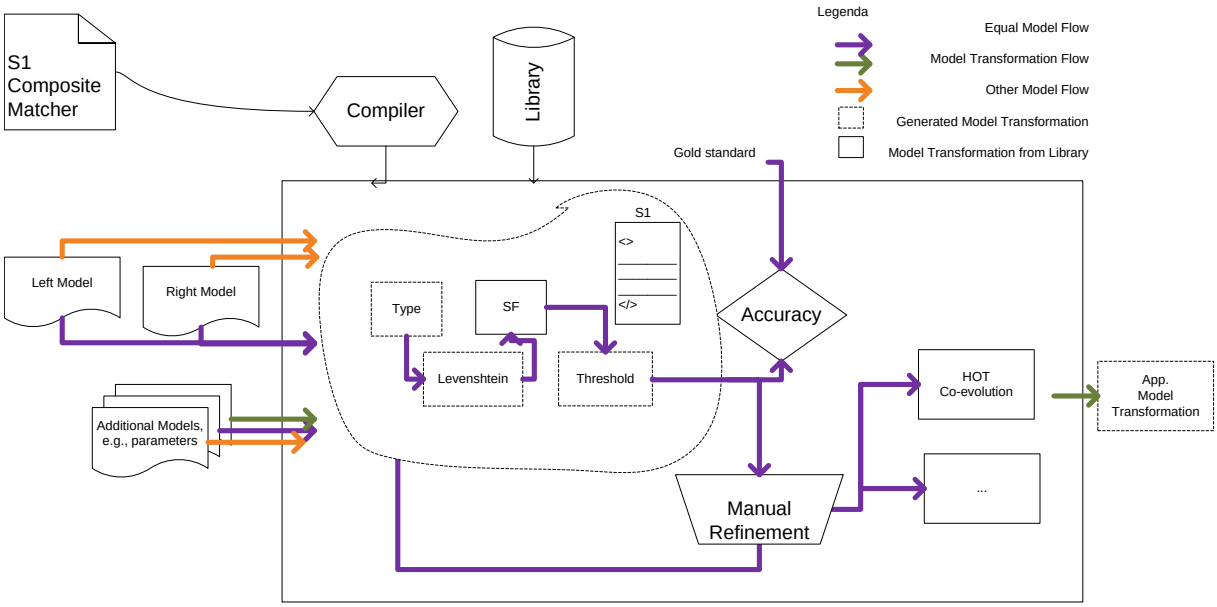


Figure 4.1: AML functional components

  - And yields an *equal* model.

- *S*1 can import heuristics which are available in a **library**.
- A component enables the **accuracy assessment** of *S1* by comparing a computed *equal* model to a gold standard which also an *equal* model.
- A component allows **manually refinement** of *equal* models.
- HOTs take computed *equal* models and generate ATL transformations. Each HOT addresses an application domain, e.g., co-evolution.

The next sections describe in detail the artifacts manipulated by the functional AML components.

4.2.2 Parameter model

A composite matcher can take as input parameter models which are means for customizing. Thresholds or dictionaries (for example) can be indicated as parameters. Parameter models conform to the parameter metamodel given in Listing. 4.1. The main metamodel concepts are `ParameterList` and `Parameter`, a parameter can be a number or a string. Other kinds of parameters can be indicated by extending the `Parameter` concept.

Listing 4.1: Excerpt of the parameter metamodel

```
1 class ParameterList {
2  reference parameters[*] container : Parameter oppositeOf list;
3 }
4 abstract class Parameter {
5  reference list : ParameterList oppositeOf parameters;
6  attribute name : String;
```

```
7  }
8  abstract class NumericParameter extends Parameter {}
9  class RealParameter extends NumericParameter {
10  attribute value : Double;
11 }
12 class IntegerParameter extends NumericParameter {
13  attribute value : Integer;
14 }
15 class StringParameter extends Parameter {
16  attribute value : String;
17 }
```

4.2.3 Equal (mapping) model

Fig. 4.1 depicts how *equal* models interact with matching transformations (purple lines). Transformations refine *equal* models in a stepwise fashion.

An *equal* model conforms to the *equal* metamodel[1] illustrated in Listing 4.2. Its main concept is `Equal` which describes a simple mapping. `Equal` refers to an element of *LeftModel* and an element of *RightModel* by means of the `left` and `right` references, respectively. `Equal` has a similarity value.

Listing 4.2: Excerpt of the equal (mapping) metamodel

```
1  class Equal extends WLink {
2   attribute similarity : Double;
3   reference left container : LeftElement;
4   reference right container : RightElement;
5  }
6  abstract class ReferredElement extends WLinkEnd {}
7  class LeftElement extends ReferredElement {}
8  class RightElement extends ReferredElement {}
9  class Association extends WAssociation {}
```

To represent a mapping with cardinality different to 1:1, for example m:1, we create a set of `Equal` elements referencing the same `RightElement` and differing `LeftElements`. Then, we associate such `Equal` elements by using an `Association`. Moreover, we can extend `Equal` or `WLink` to describe further complex mappings.

4.2.4 AML composite matcher

The compiler takes as input files expressing composite matchers in the textual concrete syntax of the AML language. This section presents the syntax of the language based on M2-to-M2 matching examples. Appendix A and Appendix B give the abstract and concrete syntaxes of AML.

4.2.4.1 Overall structure of composite matcher definition

Composite matcher definition form *strategies*. A strategy contains an *import* section, a set of *matching methods* (going from *creation* to *selection*), a *models block*, and a *modelsFlow block*.

[1] This metamodel extends concepts of the core weaving metamodel [2].

Listing 4.3: Overall structure of an AML matcher

```
1 strategy S1 {
2  imports SimilarityFlooding;
3  create TypeClass () {...}
4  ...
5  sel Threshold () {...}
6  models {...}
7  modelsFlow {...}
8 }
```

A strategy starts with the keyword `strategy` followed by the name of the matcher (e.g., `S1`). The keyword `import` declares AML matchers (e.g., `SimilarityFlooding`) intended to be fully invoked from the *modelsFlow block*. The next paragraphs explain the rest of AML notations.

4.2.4.2 Matching methods

Matching method is the basic construct in AML. It overlaps the notion of heuristic given in Section 4.1, therefore, a matching method may specify creation, similarity, aggregation, selection or user-defined functionality.

A matching method starts with the keyword `create`, `sim`, `aggr`, `sel`, or `uses`, followed by a name, a list of models[2], a list of ATL and Java libraries to be used in the method, and a body. In general, the body can include an *inpattern* and a set of variables. However, the body contains specificities associated to each kind of matching method.

Create method establishes correspondences between elements of *LeftModel* and `RightModel` given a condition. The condition begins by the keyword `when`. To refer to *left* and *right* elements, one can use the constructs `thisLeft` and `thisRight`, respectively.

Listing. 4.4 shows a `create` method called `TypeClass`. It creates correspondences between M2-to-M2 elements having the same type, i.e., `Class`.

Listing 4.4: Type AML method

```
1 create TypeClass () {
2  when
3    thisLeft.isClass and
4    thisRight.isClass
5 }
```

If a creation method is devoted to M1-to-M1 matching, then an *equal pattern* is necessary. An `equal pattern` allows the declaration of types via the keywords `leftType` and `rightType`. Listing. 6.8 (line 5) gives a concrete example.

Sim method manipulates the similarity values of correspondences. OCL expressions indicate how to obtain such values. OCL expression is followed by the keyword `is`.

Listing. 4.5 illustrates the `sim` method `Levenshtein`. Similarity values are computed by the helper `simStrings` contained in the ATL library `Strings`. The helper internally calls an edit-distance function provided by the SimMetrics Java API [94]. The keywords `ATLLibrary` and `JavaLibrary` indicate the helper and Java API used by `Levenshtein`.

[2]Note that this list excludes the mapping model manipulated by the method

Listing 4.5: Levenshtein AML method

```
1 sim Levenshtein ()
2 ATLLibraries{
3  (name='Strings', path='../AMLLibrary/ATL/Helpers')
4 }
5 JavaLibraries{
6  (name='match.SimmetricsSimilarity', path='../AMLLibrary/Jars/simmetrics.jar')
7 }
8 {
9  is thisLeft.name.simStrings(thisRight.name)
10 }
```

Listing. 4.6 depicts a more complex `sim` method. This method illustrates two aspects: list of input models and variables. Firstly, line 1 declares the input model list that contains a mapping model (i.e., `prop`). Note that `prop` is additional to the mapping model the method is supposed to manipulate. This mapping model remains implicit in the method declaration but has to specified in the `modelsFlow block`. Secondly, Lines 2-5 indicate a variable followed by the keyword `using`.

Listing 4.6: SF AML method

```
1 sim SF (prop : EqualModel(m1:Metametamodel, m2:Metametamodel))  {
2  using {
3   propEdges : Sequence(OclAny) = thisModule.propMap.get(thisEqual.xmiIDs_Equal);
4  }
5  is
6   if propEdges.oclIsUndefined() then
7    thisSim
8   else
9    if propEdges.isEmpty() then
10    thisSim
11   else
12    thisSim
13    +
14    propEdges
15    ->collect(e | e.propagation * thisModule.mapEqual.get(e.outgoingLink)
16    ->first().similarity)
17    ->sum()
18   endif
19  endif
20 }
```

Sel method chooses correspondences that satisfy a condition. The condition starts with the keyword `when`. The condition often involves the expression `thisSim` that refers to similarity values.

Listing. 4.7 shows a method that select mappings with a similarity value higher than a given threshold. Line 2 specifies this condition.

Listing 4.7: Threshold

```
1 sel Threshold () {
2  when thisSim > 0.7
3 }
```

Aggr method indicates a function of aggregation of similarity values. The function is an OCL expression (often) including the following constructs: `Summation`, `thisSim`, and `thisWeight`.

Listing. 4.8 illustrates an `aggr` method. It computes a weighted sum of similarity values of mapping models. The method needs relative weights associated to input mapping models. Weights and mapping models are indicated in the method invocation (see line 7). The method declaration, in turn, shows how the `Summation` expression adds the results of the multiplication of similarity values to weights, denoted as `thisSim thisWeight` (lines 1-3).

Listing 4.8: Weighted Sum

```
1 aggr WeightedSum () {
2  is Summation(thisSim * thisWeight)
3 }
4
5 modelsFlow {
6 ...
7 weighted1 = WeightedSum[0.5:lev, 0.5:outSF]
8 ...
9 }
```

User-defined method has a signature but not a body. The reason is that user-defined functionality is implemented by means of an external ATL transformation. Listing. 4.9 depicts an user-defined method.

Listing 4.9: Propagation

```
1 uses Propagation[IN : EqualModel(m1:Metametamodel, m2:Metametamodel)]()
```

4.2.4.3 Models block

This section specifies the models taken as input by a composite matcher. Three kinds of models are possible: `equal` (or mapping) model, `weaving` model, and `input` model.

An `input` model declaration is composed of a name and a metamodel, for example, `m1 : '%EMF'`. As show in Listing. 4.10, `equal` and `weaving` model declarations are more elaborated. To specify an *equal* model, one uses the keyword `EqualModel` followed by the declaration of *right* and *left* input models (line 2). A weaving model declaration, in turn, starts with the keywork `WeavingModel` following by an AMW core extension and a list of woven input models. A difference between an *equal* model and a weaving model is that the former links two models and the latter links n models.

Listing 4.10: Excerpt of a models block

```
1 models {
2 map : EqualModel(m1:'%EMF', m2:'%EMF')
3 inst : WeavingModel(Trace)(m1model : m1, m2model : m2)
4 ...
5 }
```

4.2.4.4 ModelsFlow block

This block allows us to declare how all kinds of models interact with matching methods. It consists of matching method invocations.

An invocation is comprised of an output mapping model, a method name, a list of mapping models, and an optional list of additional models. Our example respectively illustrates all these parts: `instances`, `ClassMappingByData`, `(inst)`, and `[tp]`.

Listing 4.11: Matching method invocation

```
1 instances = ClassMappingByData[tp](inst)
```

In our example, parenthesis contain the list of additional models. They can be mapping, weaving, or input models. This list has to overlap the list of models established by the method signature.

Brackets, in turn, contain the list of mapping models that a method has to manipulate. This list is quite flexible because one can directly refer to a mapping model or to a full method invocation. The list of mapping models for an `aggr` method differs from others. Firstly, it contains more than one mapping model. Secondly, it associates a weight to each mapping model. Listing. 4.12 shows the `WeightedSum` method taking as input the mapping models `lev` and `outSF`, and their corresponding weights, i.e., 0.5–0.5.

Listing 4.12: Aggr method invocation

```
1 weighted1 = WeightedSum[0.5:lev, 0.5:outSF]
```

As a final point, we want to spell out that it is possible to invoke an entire composite matcher from a `modelsFlow block`. One refers to the matcher by using its name. The lists of mapping models and additional models contain no elements. The compiler infers them if the `modelsFlow block` elements overlap the matcher definitions.

For illustration purposes, Listing. 4.13 shows the `SimilarityFlooding` matcher, and Listing. 4.14 the way of calling it from $S1$ (line 10).

Listing 4.13: Similarity Flooding as an AML algorithm

```
1 strategy SimilarityFlooding {
2 models {...}
3 modelsFlow {
4  filtered = Threshold[inSF]
5  prop = Propagation[filtered]
6  sf = SF[filtered](prop)
7  outSF = Normalization[sf]
8 }
9 }
```

Listing 4.14: The S1 strategy calling the similarity flooding algorithm in a single line

```
1 strategy S1 {
2 imports SimilarityFlooding;
3 models {...}
4 modelsFlow {
5  tp = TypeClass[map]
6  typeRef = TypeReference[map]
7  typeAtt = TypeAttribute[map]
8  merged = Merge[1.0:tp, 1.0:typeRef, 1.0:typeAtt]
9  inSF = Levenshtein[merged]
10  outSF = SimilarityFlooding[]
11  instances = ClassMappingByData[tp](inst)
12  filInst = ThresholdBySample[instances]
13  weighted1 = WeightedSum[0.5:lev, 0.5:outSF]
14  thres2 = Threshold[weighted1]
15  weighted2 = WeightedSum[0.5:thres2, 0.5:filInst]
16  result = BothMaxSim[weighted2]
17 }
```

4.2.4.5 Other AML constructs

Besides `thisLeft`, `thisRight`, `thisSim`, `Summation` and `thisWeight`, AML provides the following notations:

- `thisEqual` and `thisEqualModel`. Whereas the former refers to the correspondences contained by a mapping model, the latter alludes to a full mapping model. Listing. 4.18 gives examples of these notations.
- `thisInstances` recovers M1-to-M1 mappings whose linked elements conform to the metaelements linked by an M2-to-M2 mapping model. See Listing. 4.17 for further explanations.

Let us present the main differences between AML methods and their corresponding ATL transformations:

1. AML constructs hide source and target patterns that respectively specify: a) types of conformance of matching models, and b) mapping metamodel concepts. We can refer to them using the constructs `thisLeft`, `thisRight`, and `thisEqual`. The developer uses `thisLeft` to refer to elements of $LeftModel$, `thisRight` to relate to elements of $RightModel$, and `thisEqual` to refer to mapping elements.
2. Exceptions to the 1.a item follow: 1) M1-to-M1 matching algorithms require creation methods including source patterns, 2) if besides mapping models a rule takes as input additional models then the rule requires source pattern.
3. In the AML versions only remain conditions and functions modifying similarity values.
4. AML provides notations that factorize code, e.g., `Summation`, `thisWeight`, etc.

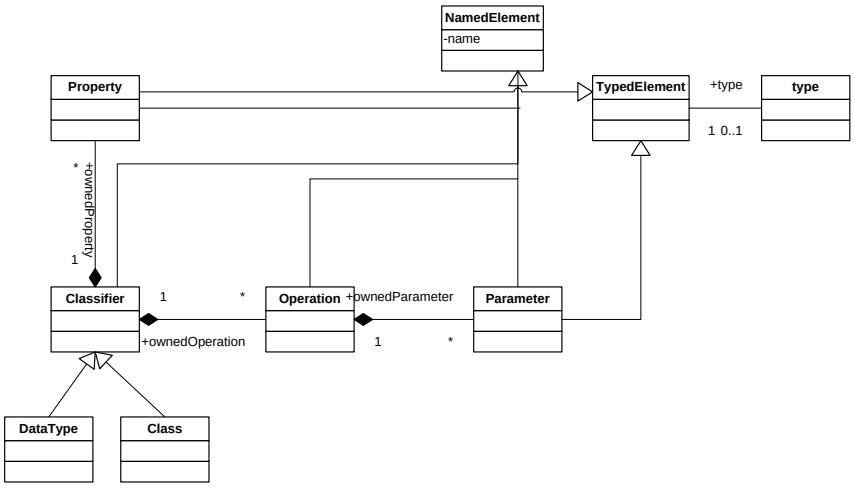
At last point, Table. 4.1 illustrates how the analysis concepts given in Section 4.1 overlap implementation units and syntax defined in the design phase.

Concept	Notation	Implementation unit
Composite matcher, i.e., matching strategy, matching algorithm	`modelsFlow`	Transformation chain, i.e., Ant script
Heuristic, technique	`method (create, sim, aggr, sel, uses)`	ATL Transformation
Mapping	`thisEqualModel`	Equal model
Correspondence	`thisEqual`	Equal element

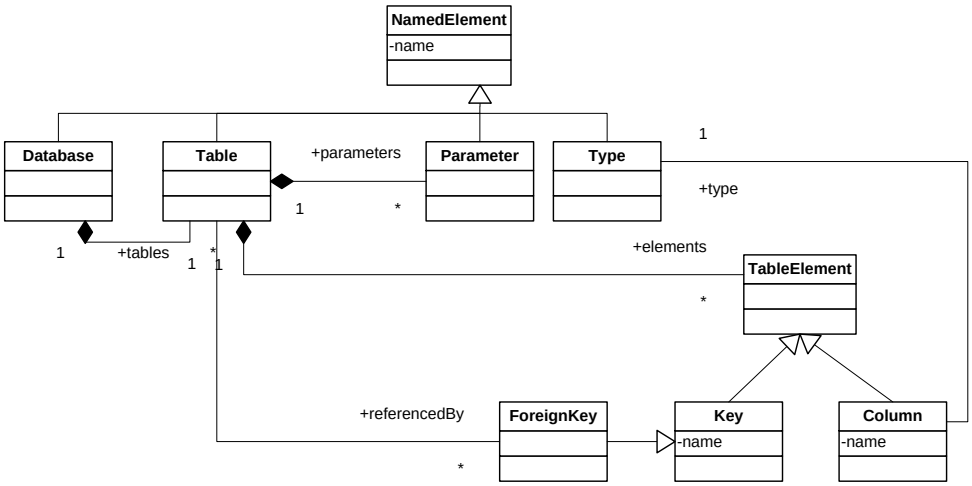
Table 4.1: Overlapping between analysis concepts, notations, and implementation units of AML

4.2.5 An AML M2-to-M2 matching algorithm

This subsection presents how to use the main constructs of AML by means of a full composite matcher. The matcher invokes methods that inspect diverse metamodel aspects, i.e., labels, structure, and data instances. Given the methods, we have configured a `modelsFlow block` that produces the more accurate correspondences for an illustrating pair of metamodels, i.e., (UML class diagram, SQL-DDL). Fig. 4.2(a) and Fig. 4.2(b) represent the concepts of each metamodel: `Class`, `Table`, etc.



(a) UML class diagram metamodel



(b) SQL-DDL metamodel

Figure 4.2: Input metamodels for an M2-to-M2 algorithm

4.2.5.1 Models block

Listing. 4.15 shows the *S1* `models block` which takes two models as input; `map` and `inst`. `Map` is an empty M2-to-M2 mapping model and `inst` is an M1-to-M1 mapping model. `Map` refers to the illustrating pair of metamodels. `Inst` refers to models conforming to the metamodels; *LeftModel* and *RightModel* conform to UML class diagram and SQL-DDL, correspondingly. Fig. 4.3 gives the `inst` mapping model displayed in the AMW GUI. Its *LeftModel* and *RightModel* represent the domain of online shopping. *LeftModel* contains (for instance) the `Catalog` and `Product` elements conforming to `Class`. *RightModel* has within `category` and `item` conforming to `Table`. The red rectangle points an M1-to-M1 mapping linking `Item` and `item`. The `inst` mapping model has been computed by the AMW traceability use case[3]. This computation is out of the example scope. We focus now on M2-to-M2 mapping discovery by using, among other information, M1-to-M1 mappings.

Listing 4.15: Models section of an illustrating strategy

```
strategy S1 {
models {
 map : EqualModel(m1:'%EMF', m2:'%EMF')
 inst : WeavingModel(Trace)(m1model : m1, m2model : m2)
}
...
}
```

4.2.5.2 ModelsFlow block

Listing. 4.16 illustrates the *S1* of `modelsFlow block`. Every method (except `TypeClass`, `TypeReference`, and `TypeAttribute`) consumes mapping models produced during the strategy execution.

4.2.5.3 Matching methods

Below we discuss the methods used in *S1* which have not been presented in Section 4.2.4. Each matching method has an associated code listing. Given the illustrating pair of metamodels, we depict the output mapping models of some methods by means of figures. For the sake of readability these figures contain a few correspondences. Each figure shows *LeftModel* and *RightModel* as well as the mappings between their elements. Dotted lines represent mappings and their respective similarity values.

Listing 4.16: ModelsFlow section of an illustrating strategy

```
modelsFlow {
 tp = TypeClass[map2]
 typeRef = TypeReference[map2]
 typeAtt = TypeAttribute[map2]
 merged = Merge[1.0:tp, 1.0:typeRef, 1.0:typeAtt]
 inSF = Levenshtein[merged]
 filtered = Threshold[inSF]
 prop = Propagation[filtered]
 sf = SF[lev](prop)
 outSF = Normalization[sf]
```

[3]http://www.eclipse.org/gmt/amw/usecases/traceability/

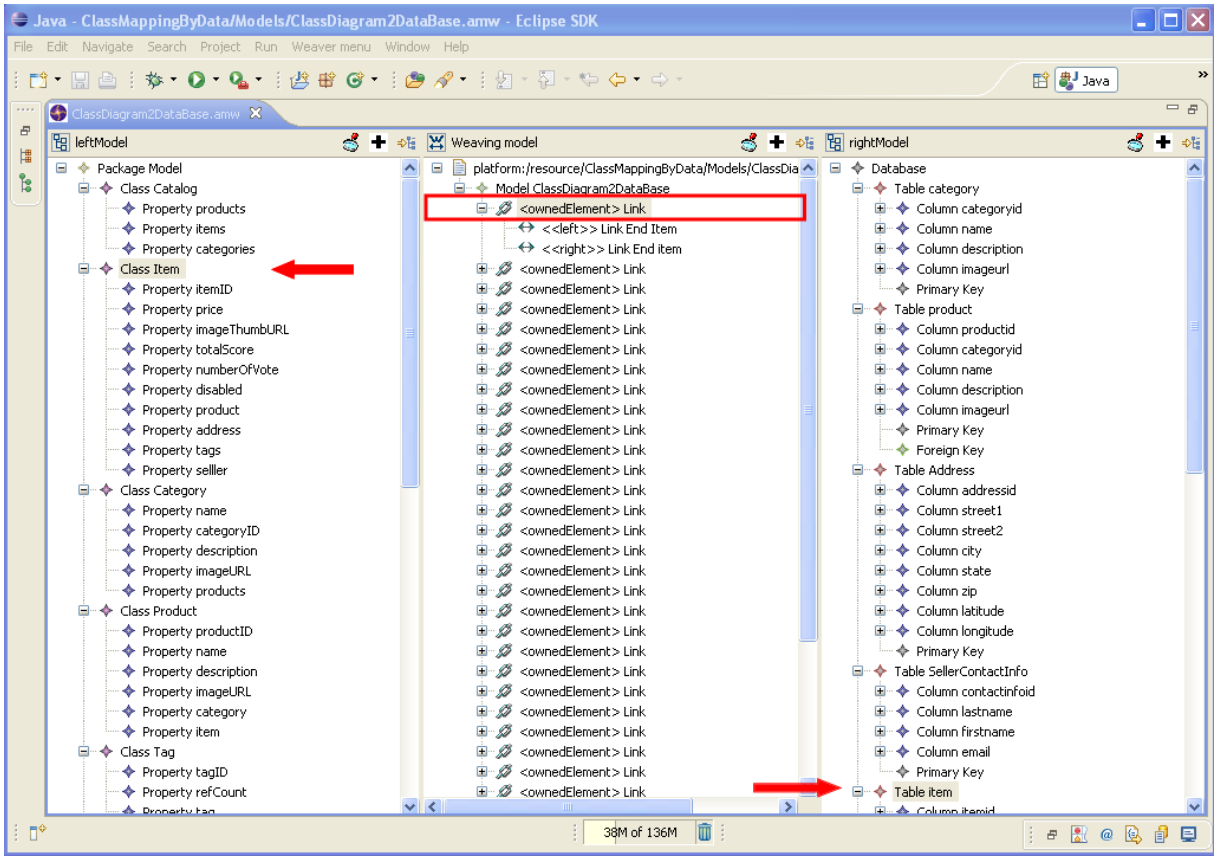


Figure 4.3: Input weaving model

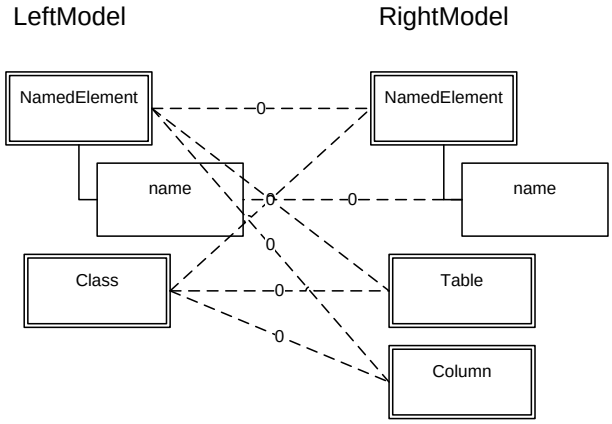


Figure 4.4: Merge output mapping model

```
11  instances = ClassMappingByData[tp](inst)
12  filInst = ThresholdBySample[instances]
13  weighted1 = WeightedSum[0.5:lev, 0.5:outSF]
14  thres2 = Threshold[weighted1]
15  weighted2 = WeightedSum[0.5:thres2, 0.5:filInst]
16  result = BothMaxSim[weighted2]
17 }
```

`TypeClass`, `TypeReference`, and `TypeAttribute` create a correspondence for each pair of model elements having the same type, i.e., `Class`, `Reference`, or `Attribute`. Listing. 4.4 shows how `TypeClass` looks like. The `Merge` method puts together the mappings returned by the other heuristics. We have instrumented it by means of an aggregation construct. Fig. 4.4 shows the output mapping model of `Merge`. Note that the correspondences similarity value is 0.

SimilarityFlooding (Listing. 4.16, lines 7-10) propagates previously computed similarity values. It is inspired by the *Similarity Flooding* algorithm [77]. We have implemented this algorithm by means of three AML heuristics (`Threshold`, `SF`, and `Normalization`) and an external ATL transformation (`Propagation`). Below we describe each of them:

1. `Threshold`, its purpose is to filter mappings before the propagation. We will give more details about `Threshold` later on.

2. `Propagation` creates an association (i.e., a `PropagationEdge`) for each pair of mappings (*m1* and *m2*) whose linked elements are related. For example, `Propagation` associates the `(DataType, Database)` mapping to `(name, name)` because `DataType` contains `name`, and `Database` contains `name` as well.

3. `SF` propagates a similarity value from *m1* to *m2* as indicated by the `PropagationEdges`.

4. `Normalization` makes similarity values conform to the range [0,1]. In the example, `SF` propagates the similarity values given by `Levenshtein`. Fig. 4.5 provides the `Normalization` output mapping model. The red line indicates the propagation from `(name, name)` to `(DataType, Database)`.

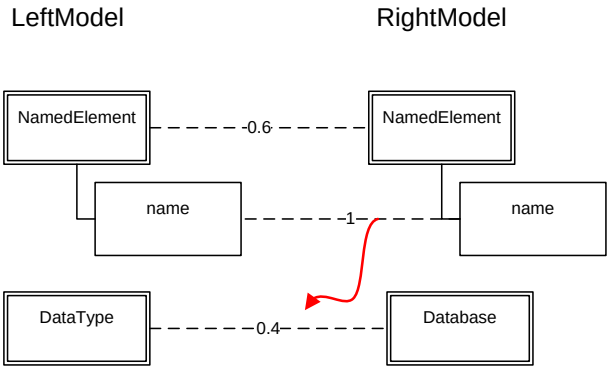


Figure 4.5: Normalization output mapping model

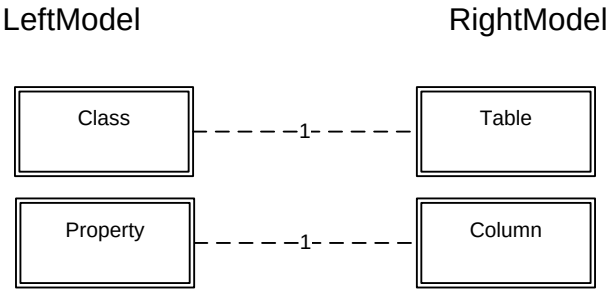


Figure 4.6: ThresholdBySample output mapping model

`ClassMappingByData` (Listing. 4.17) propagates similarity values from M1-to-M1 mappings to M2-to-M2 mappings. We use the `thisInstances` primitive (line 3) to recover, for each M2-to-M2 mapping, e.g., linking the concepts a and b, the M1-to-M1 mappings whose linked elements conform to a and b. `ClassMappingByData` assigns 1 to an M2-to-M2 mapping if there exists at least a corresponding M1-to-M1 mapping, otherwise it assigns 0. The `ThresholdBySample` method filters the M2-to-M2 mappings satisfying the latter case. Fig. 4.6 gives the `ThresholdBySample` output mapping model; only two correspondences remain. The rationale is that the M1-to-M1 mapping model (i.e., `inst`) only links elements conforming to (`Class`, `Table`) and (`Property`, `Column`).

Listing 4.17: Instances

```
1 sim ClassMappingByData (mapModel : WeavingModel(Trace)(leftModel : m1,rightModel : m2))
2 {
3     using {
4         mappingsModel : Trace!Link = Trace!Link.allInstancesFrom('mapModel');
5     }
6     is if thisInstances(mappingsModel)->notEmpty() then
7      1
8      else
9        0
10       endif
11 }
```

`BothMaxSim` (Listing. 4.18) selects a correspondence (a, b) if its similarity value is the highest among the values of other correspondences linking either a or b. We have implemented this heuristic by means of two hashmaps: `equalMaxSimByLeft` and `equalMaxSimByRight`. In these hashmaps, $LeftModel$ and $RightModel$ elements are *keys* and

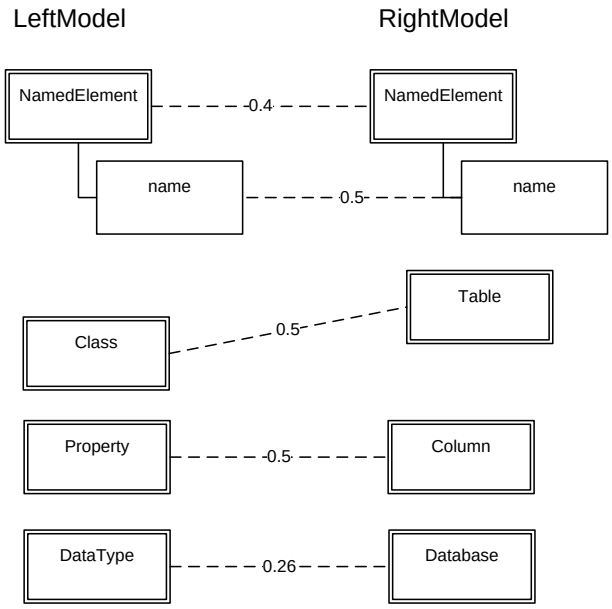


Figure 4.7: BothMaxSim output mapping model

correspondences with the highest similarity scores are *values*. `BothMaxSim` has been inspired by [62]. Fig. 4.7 illustrates the `BothMaxSim` output mapping model. This result contains a good number of correct correspondences. However, the algorithm introduces a false negative `(DataType, Database)`.

Listing 4.18: BothMaxSim

```
1 sel BothMaxSim ()
2 {
3  when
4  thisEqualModel.equalMaxSimByLeft.get(thisLeft).includes(thisEqual)
5  and
6  thisEqualModel.equalMaxSimByRight.get(thisRight).includes(thisEqual)
7 }
```

This example has shown how AML matching algorithms calculate mappings. In general, AML matching algorithms populate and prune mapping models in a stepwise manner. The next subsection discusses an important decision taken during the AML design.

4.2.6 Compilation strategy: graphs versus models

We want to elaborate on the compilation of `create` methods because it is related to the first issue addressed by our thesis. As stated in Section 4.2, `create` methods use the `thisLeft` and `thisRight` constructs to hide (at design time) metamodel types. Thus, a given AML `create` rule may remain useful to match many pairs of models. Even though these constructs solve the type declaration problem at design time, another solution is required at compilation time, that is, when AML `create` rules have to be translated into executable ATL transformations. We have experimented two solutions to compile this kind of rules:

1. **Graph-based solution** translates a `create` rule into an ATL transformation written in terms of a graph metamodel. In other words, the solution translates the `thisLeft` and `thisRight` constructs into a source pattern involving graph metamodel concepts, e.g., `Node`, `Edge`, etc. Besides ATL transformation generation, the solution implies two additional steps: *pre-translation* and *translation*. The pre-translation step generates ATL code devoted to make models to be conforming to the graph metamodel. The translation step executes such a code.

2. **Model-based solution** aims at keeping the matching models as they are. The solution varies its modus operandi with respect to the kind of matching algorithm. Thus, if one has an M2-to-M2 algorithm, the compiler automatically translates a `create` rule into an ATL rule whose source pattern has a metametamodel type; `EModelElement` for Ecore metamodels or `ModelElement` for KM3 metamodels. On the other hand, if one wants an M1-to-M1 matching algorithm, then one has to develop a `create` method for each desired pair of metamodel types. The compiler translates each rule into an ATL transformation containing the indicated types.

Let us discuss the implications of each solution. Suppose the user wants to developed an AML `create` rule called *MR*.

Graph-based solution

1. Source pattern specification is not necessary.

2. The *MR* rule is compiled once and can be reused many times.

3. If a `create` condition is not specified, the ATL engine performs a Cartesian product between the elements of $LeftModel$ and $RightModel$ conforming to `Node`.

4. The pre-translation and translation steps have to be performed for each new pair *(LeftModel, RightModel)* taken by MR.

5. If one wants an external transformation to interact with the generated ATL transformation, the former has to be written in terms of the graph metamodel.

Model-based solution Its implications depends on the kind of matching algorithm. Thus, the 3 first hints of the graph-based solution apply to the M2-to-M2 matching algorithms too. With regard to M1-to-M1 matching algorithms, the 3 first hints vary as follows:

1. The user has to develop `create` rules specifying source patterns.

2. The *MR* rule is compiled every time its source pattern changes.

3. Since a source pattern is specified, the ATL engine performs a targeted Cartesian product. One can define a `create` condition to further constraint the Cartesian product. Model-based creation conditions look simpler than graph-based ones.

The 2 last hints of the graph-based solution have nothing to do with the model-based solution; Prior (pre)translation of matching models is not necessary and external transformations are written in terms of the metamodels of $LeftModel$ and $RightModel$.

We have tested the performance of two matching algorithms which have been generated by the compilation strategies mentioned above. The algorithms have matched models going from 4 to 250 elements. Some runtimes are:

- The pre-translation and translation steps took 0.19 (s) for small models and 5 (s) for large models.
- The generation of the ATL matching transformations took the same time.
- The graph-based solution generated low performance ATL matching transformations. For example, this solution generated a linguistic-based similarity transformation which took 755 (s) to match large models. In contrast, the model-based solution generated a corresponding transformation with a runtime of 75 (s).

Based on the numerical results and observations over the AML and ATL code, we have selected the model-based solution. In a nutshell, its advantages over the graph-based solution are:

- The compilation of AML rules is less expensive than the pre-translation and translation steps.
- This solution increases the performance of generated matching algorithms.
- The `create` conditions and external transformations are easier to develop and to understand.

We see only a disadvantage in the model-based solution. It comes out if the user wants an M1-to-M1 matching algorithm; he/she has to develop `create` rules indicating metamodel types.

Given the design specification of AML, the next section presents the language from the implementation point of view.

4.3 Implementation on top of the AmmA suite

4.3.1 Architecture

The previous section has presented AML from a functional point of view. Here we describe how the language has been implemented on top of the AmmA suite (i.e., ATL, TCS, KM3, and AMW), EMF, and the Eclipse platform. Fig. 4.8 shows the AML tool components (white blocks), a component description follows.

4.3.1.1 Wizard

The wizard component enables the creation of AML projects, its extends an Eclipse wizard. Fig. 4.9 shows a screenshot of the AML project wizard.

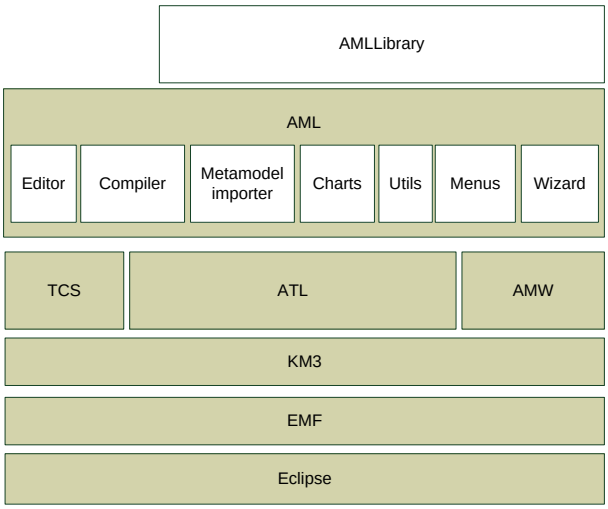


Figure 4.8: AML tool components

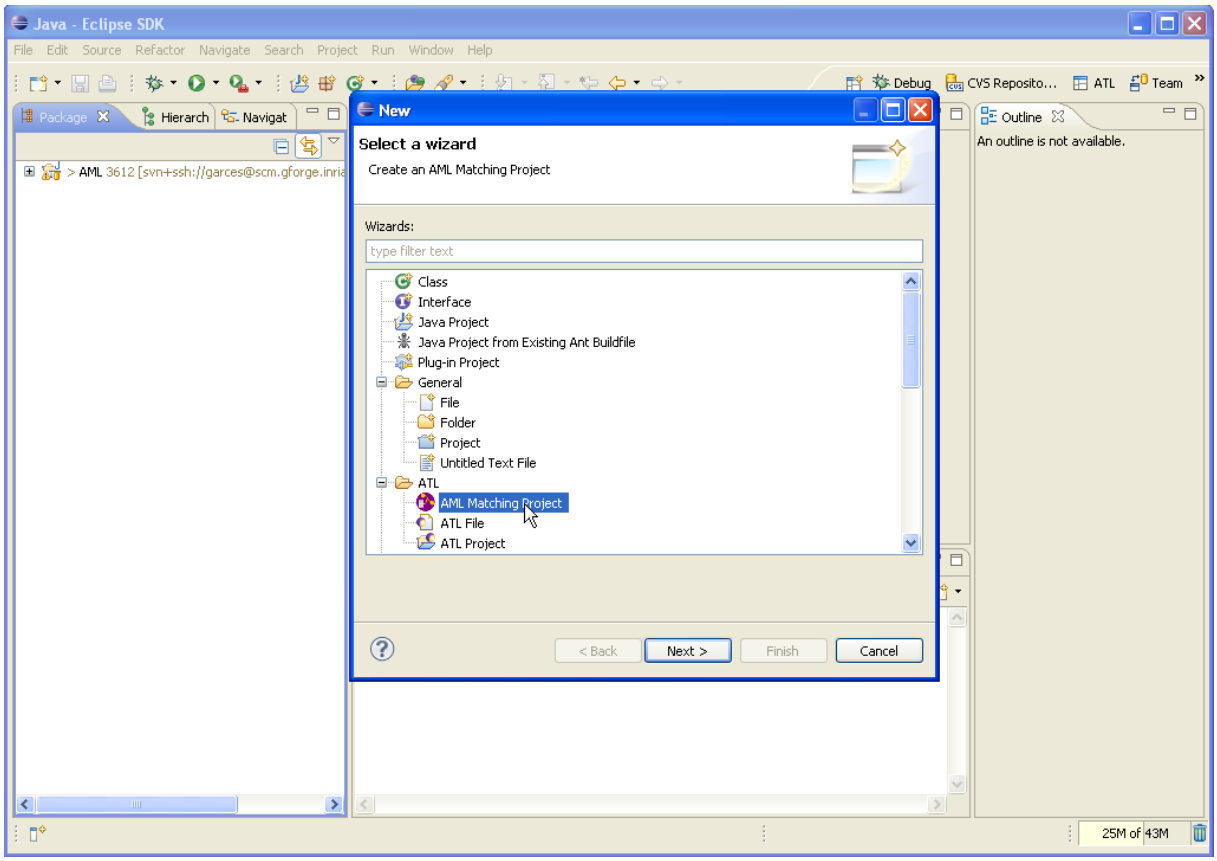


Figure 4.9: AML project wizard

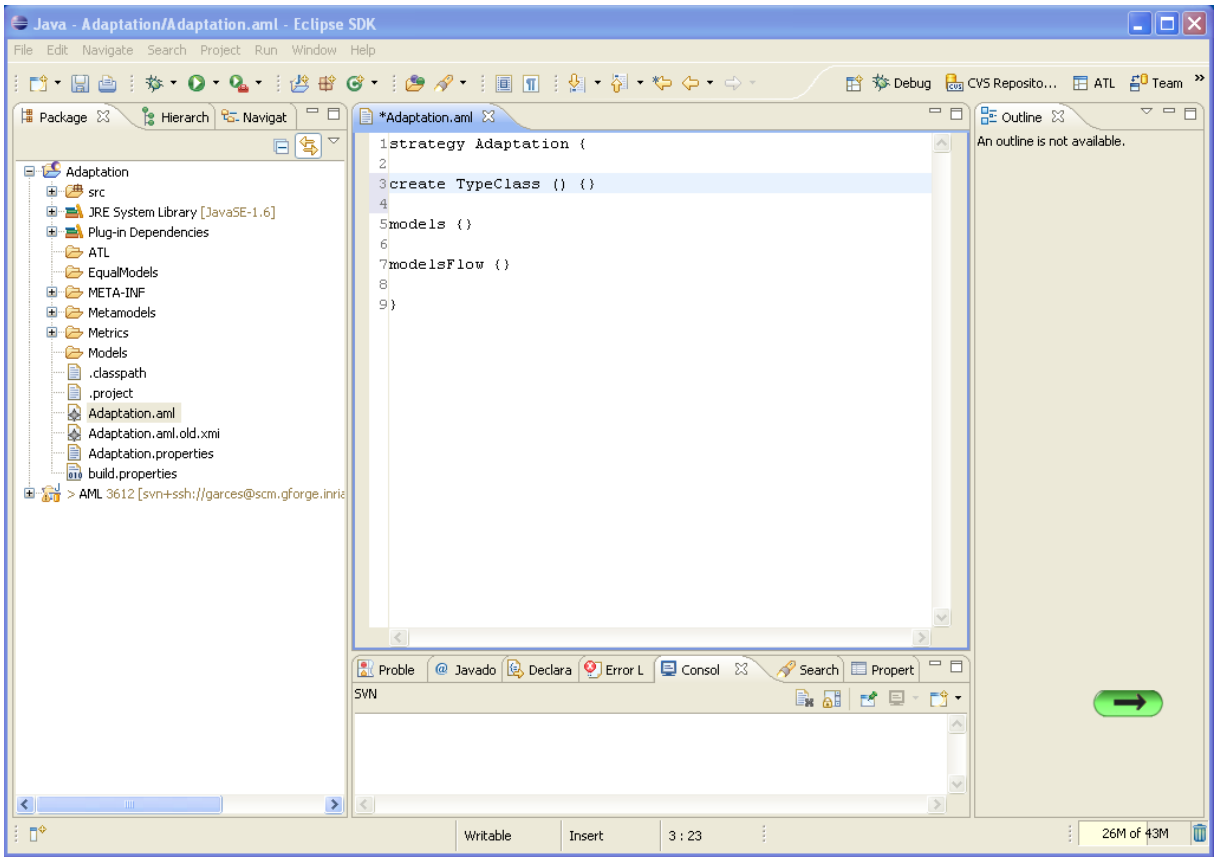


Figure 4.10: AML editor

4.3.1.2 Editor

As its name indicates, this component allows the edition of AML programs. AML annotates the programs by adding markers to indicate compilation errors. Moreover, AML reports such errors in the Eclipse Problems view. We have implemented the AML concrete syntax by using TCS [19]. TCS generates a Java parser that converts an AML program from text to model format and vice versa. The parser detects syntactical errors and an ATL transformation detects semantical ones.

Fig. 4.10 shows an empty AML program in the editor. Note that the keywords are not highlighted (e.g., `modelsFlow`). Even tough TCS generates Java code that highlights key words, the publicly available AML version does not include this particular functionality. The reason is that the generated Java code is not compatible with ATL 3.0 (the version below of AML).

4.3.1.3 Compiler

This component takes a given AML program and performs the following tasks:

1. Merge imported code to the AML program declarations. One can import AML matching rules or full strategies.

2. Generate an ATL matching transformation for each AML matching rule.

3. Translate the `modelsFlow` section into an Ant script.

4. Generate a properties file responsible for the Ant script parameterization. That is, the file indicates full paths associated to input models of the AML program.

We have devoted a HOT to each mentioned task. In particular, the second task needs Java code because the associated HOT yields one model containing all the ATL transformations together. The Java code splits the model in a set of small models (one for each ATL matching transformation). Then, the ATL and XML extractors (available in the AmmA suite) generate the ATL modules and Ant scripts in a textual format.

4.3.1.4 Metamodel importer

This component brings metamodels into agreement with Ecore or KM3. The component internally consists of ATL transformations between a pivot metamodel (i.e., Ecore or KM3) and other technical spaces (i.e., MOF, UML). These transformations have been contributed by the m2m community [95]. Notably, the metamodel importer invokes the AmmA-based EMFTriple tool [24] to translate OWL ontologies into Ecore metamodels and vice versa. If the users want a technical space not currently supported by the component, they need to develop a transformation. If there exist a transformation between the new technical space (e.g., SQL-DDL) and one of the technical spaces currently supported (e.g., OWL), users may have translation for free; instead of writing the transformation SQL-DDL to Ecore, they need to execute a chain of existing transformations, e.g., SQL-DDL to OWL and OWL to Ecore. Finally, if there is no transformation or transformation chain, one uses the extension points of the metamodel importer component (see Section 4.3.2.3).

4.3.1.5 Menus

This component provides three functionalities:

1. Create empty mapping models (often required by AML programs).

2. Create AMW properties files needed to display mapping models in the AMW GUI.

3. Compute matching metrics.

We have implemented the functionalities mentioned above by using Java. Moreover, the third functionality requires ATL transformations, among them those contributed by Eric Vépa in the `Table2TabularHTML` use case[4]. Fig. 4.11 shows the first menu functionality which is available as a mapping model is selected.

4.3.1.6 Charts

The chart component allows drawing charts from matching results. The first AML version offers line charts, bar charts, and area charts. The chart component consists of a set of

[4]http://www.eclipse.org/m2m/atl/atlTransformations/

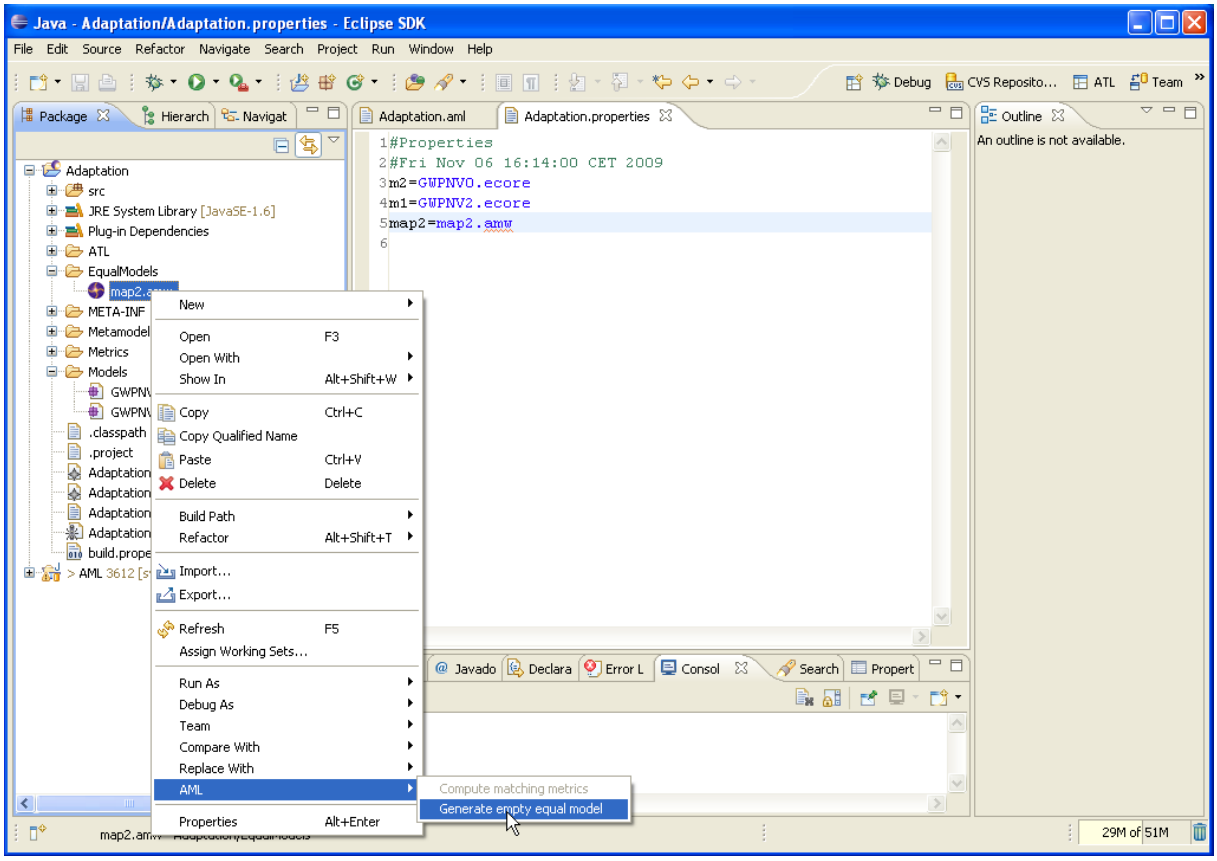


Figure 4.11: AML menus

ATL transformations taking matching results as input and generating spreadsheet files as output. Once the component has generated a spreadsheet (containing only numerical data), the user has to draw the desired chart by means of (for example) Excel wizards.

4.3.1.7 The AML library

This component is actually an Eclipse project containing artifacts usable for building new matching algorithms:

- The *match* package contains Java code called from AML algorithms.
- The *AML* folder includes the `AMLBasis` module that declares AML methods. By default the compiler links such methods to every AML algorithm. As a consequence, developers can invoke them from a `modelsFlow` section without any additional declaration.
- The *ATL* folder has within ATL code arranged in three folders:
  - *EcoreMetametamodel/KM3Metametamodel* have transformations matching metamodels, their functionality go beyond the AML methods. The separation of transformations in two folders is to indicate that the transformations match either Ecore or KM3 metamodels. Note that the *KM3Metametamodel* folder is a mirror of the *EcoreMetametamodel* folder.
  - The *Helper* folder contains ATL helpers invoked from AML methods. The helpers mostly factorize matching functionality or decouple the access to metametamodel properties.
  - The *HOT* folder includes HOTs that translate mappings into ATL transformations for a concrete application domain (e.g., co-evolution).

4.3.2 Extension points

4.3.2.1 Compiler

Developers can extend the compiler to generate code different to ATL and Ant from AML programs. They have to extend the `AmlCompiler` class, and implement new HOTs.

4.3.2.2 The AML Library

The library provides extension points related to each kind of contained artifact:

Java To add Java functionality that can be called from an AML method, developers need to create a class extending the `LibExtension` interface, and then indicate its use in the method `JavaLibraries` section.

AML Besides the `AMLBasis` module, developers may want to have more AML libraries. If they want the compiler to link such libraries, it is necessary to modify the `AmlBuildVisitor#getLibraries` method.

ATL The addition of further external ATL M2-to-M2 transformations is possible. Developers just need to implement the Ecore transformations, and execute the `RefactorATL-TransformationEcoretoKM3` transformation. The latter automatically generate the mirror KM3 transformations.

If a helper is needed for an AML method, developers simply need to create an ATL library (storing the helper), and indicate its use in the method `ATLLibraries` section.

In addition to co-evolution transformations, it is certain that other kinds of transformation (associated to other application domains) can be derived from mappings. To do that, it is needed:

1. develop an ATL transformation translating simple mappings into complex. An example is the `ConceptualLink` transformation used in the co-evolution use case.

2. implement a HOT encoding ATL patterns for each kind of complex mapping. The HOT has to superimpose *HOT_match.atl*.

3. modify the *MatchingMethod-HOT.properties* which relates the output files of steps 1 and 2.

4.3.2.3 Metamodel importer

Besides the pivot transformations suggested in Section 4.3.1.4, a way to match metamodels not conforming to KM3 or Ecore, for example OWL, is the following:

- to implement `create` methods indicating the types of interest, e.g., `Class`, `Individuals`, `Relation` in the case of OWL.

- to build matching algorithms using the new `create` methods.

4.3.3 The AML tool in numbers

Fig. 4.12 and Fig. 4.13[5] show the AML source code from two different points of view: 1) what languages have been used to implement the code?, and 2) what has been the effort invested in each component? Whilst the use of Java is moderate, i.e., 18%, Fig. 4.12 shows an extensive use of the AmmA languages, i.e., 82%. In particular, the ATL source code corresponds to 66% of the total. Fig. 4.12 shows the percentage of AML source code, i.e., 3%. It corresponds to the library of heuristics and the algorithms described Section 4.4.4 and Section 5.5, respectively. Instead of showing a poor use of AML, this percentage reveals that AML factorizes a considerable portion of ATL code. While we have implemented more than 20 heuristics in 273 lines with AML, we have implemented 10 user-defined heuristics in 3516 ATL lines[6]. As explained in Section 4.2.4, the difference between an AML heuristic and its corresponding ATL transformation is that the latter needs additional rules to work on. AML keeps such rules implicit.

[5]The pie does not depict the percentage of the metamodel importer because the component uses transformations mostly contributed by the m2m community.

[6]This value includes the heuristics described in Section 4.4.4 and the transformations of the co-evolution use case Section 6.1.

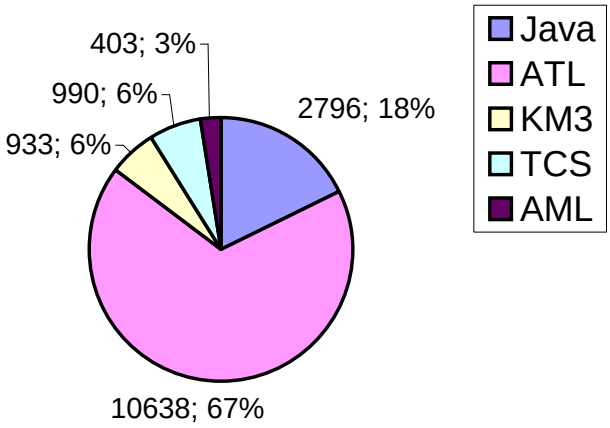


Figure 4.12: Distribution of AML source code (languages point of view)

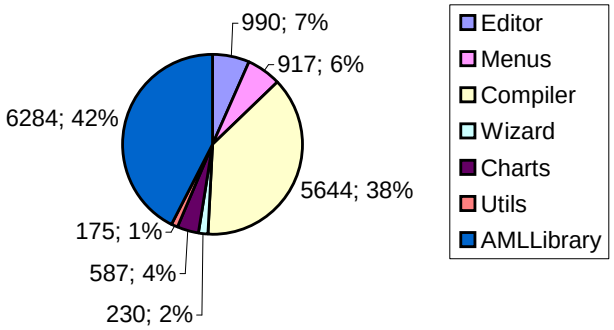


Figure 4.13: Distribution of AML source code (components point of view)

Fig. 4.13 depicts that most of the efforts have been devoted to implement the compiler and the AML Library components. Especially, we have dedicated 4310 lines to the compiler. We believe that the effort worth because users may develop model transformation-based matching algorithms with a lowest effort than before.

4.4 AML library

Table. 4.2 lists the implemented AML matching heuristics. The library contains 24 heuristics embedding creation, similarity, selection, aggregation, and user-defined logic. All of them can be used in M2-to-M2 matching algorithms. Linguistic-based, selection, and aggregation heuristics can be used in M1-to-M1 matching algorithms. Some heuristics listed in Table. 4.2 have been described in Section 4.2.5, the rest is explained here.

Creation	Similarity				Selection	Aggregation
	Linguistic-based	Constraint-based	Instance-based	Structure-based		
TypeClass	Wordnet	TypeElement	ClassMappingByData	Statistics	ThresholdMaxSim	WeightedAverage
TypeReference	MSR	Multiplicity	AttributeValues	SimilarityFlooding	BothMaxSim	Merge
TypeAttribute	Levenshtein		SetLinks		Threshold	
TypeDatatype	Name					
TypeStrF						
TypeEnumeration						
TypeEnumLiteral						
CreationByFullNameAndType and CreationAddedDeleted						
8	4	2	3	2	3	2

Table 4.2: AML matching heuristic library

4.4.1 Creation heuristics

TypeStrF, **TypeEnumeration**, and **TypeEnumLiteral** create mappings between two metamodel elements conforming to a given metametamodel type, i.e., `StructuralFeature`, `Enumeration`, or `EnumLiteral`.

Listing 4.19: TypeStrF, TypeEnumeration, and TypeEnumLiteral

```
create TypeStrF ()
{
 when
    thisLeft.isStrFeature and
    thisRight.isStrFeature
}

create TypeEnumeration ()
{
 when
    thisLeft.isEnumeration and
    thisRight.isEnumeration
}

create TypeEnumLiteral ()
{
 when
    thisLeft.isEnumLiteral and
    thisRight.isEnumLiteral
}
```

4.4.2 Similarity heuristics

4.4.2.1 Linguistic-based heuristics

Name implements the string equality heuristic described in Section 2.4.3.2.

Listing 4.20: Name

```
sim Name ()
{
 is
 if thisLeft.name = thisRight.name then
  1.0
 else
  0
 endif
}
```

Measures of Semantic Relatedness (MSR) We have developed AML heuristics exploiting MSR. This is a computational mean for extracting relatedness between any two labels based on a large text corpora, e.g., Google or Wikipedia [96]. An effort to centralize and unify MSR technology is a publicly available Web server[7]. To request the server for a specific relateness measure, it is necessary to make a http request passing the following parameters:

[7]http://cwl-projects.cogsci.rpi.edu/msr/

- `msr` - the name of the text corpora one would like to use.
- `terms` - list of terms to be compared to terms2.
- `terms2` - if terms2 is not specified, terms2 = terms.

The http result is a page containing a link to a progress text file. This file may be requested at any time at the provided address. It will tell both, the current progress of the batch, and the address of a (partial or completed) spreadsheet file. The spreadsheet file has 3 columns: term1, term2, and relateness score.

Our MSR implementation has two matching transformations: an user-defined one (called `RequestMSR`) and a similarity method (named `MSR`). `RequestMSR` sends the http request. The user has to recover the spreadsheet file from the address stored in the progress text file. Then, he/she has to execute the AML `MSR` similarity method using $LeftModel$, $RightModel$ and spreadsheet files as input. Listing. 4.21 presents the code of `RequestMSR` and `MSR`, we explain their functionality below.

Listing 4.21: RequestMSR and MSR

```
1 uses RequestMSR[equalM : EqualModel(leftM:Metametamodel, rightM:Metametamodel)](paramM :
    ↪ParameterMM)
2 JavaLibraries {
3  (name = 'match.MSRSimilarity', path='')
4 }
5
6 sim MSR (MSRExcel : SpreadsheetMLSimplified, paramM : ParameterMM)
7 ATLLibraries {(name = 'SpreadsheetMSR', path='../AMLLibrary/ATL/Helpers/SpreadsheetMSR')}
8 {
9  is
10  thisModule.mapExcelResult.get(
11   thisLeft.name.leftProperTerm.buildTerm(
12    thisRight.name.rightProperTerm
13   )
14  )
15 }
```

`RequestMSR` builds `terms` and `terms2` containing the labels of $LeftModel$ and $RightModel$. `RequestMSR` sends the lists to the `MSRSimilarity` Java class. `MSRSimilarity`, in turn, creates and copies the http request in the console. Then, the user has to copy and send the request to the MSR server. Notice `RequestMSR` takes as input the `paramM` model which indicates the selected `msr` (e.g. Google) and normalization parameters. The normalization consists of tokenizing labels, and filtering distractor tokens. Thus, `paramM` specifies a distractor list and tokenizers suitable for $LeftModel$ and $RightModel$. We have implemented tokenizers that break strings into tokens. Each tokenizer specifies a delimiter character that serves to separate the string:

1. `HyphenTokenizer`, a hyphen (-).
2. `UnderScoreTokenizer`, an underscore (_).
3. `UpperCaseTokenizer`, an uppercase character [A-Z].

To create a new tokenizer it is necessary to implement the `Tokenizer` interface. Section 6.2.3.1 presents an example of `paramM`, this model indicates the tokenizers and distractors used to match a concrete pair of models.

Once we have the spreadsheet file returned by the MSR server, a transformation translates it into XMI[8]. The AML `MSR` similarity method takes the XMI file, the `paramM` model, *LeftModel* and *RightModel* as input. For each pair of model elements, the method searches the corresponding similarity value in the XMI file. The method applies tokenizers and distractors again.

WordNet uses the Java API for WordNet Searching (JAWS)[9]. To compare two labels, the AML heuristic asks JAWS for retrieving their corresponding synsets from the WordNet database. A synset is a set of synonyms considered semantically equivalent. Having the synsets, the AML heuristic calculates a Jaccard distance [98]. Like in `MSR`, we normalize the labels prior to the comparison.

Listing 4.22: WordNet

```
sim WordNet (paramM : ParameterMM)
ATLLibraries {(name ='ProperTerm',path='../AMLLibrary/ATL/Helpers/')}
JavaLibraries {(name = 'match.JWISimilarity,match.ProperTermSimilarity',
        path='../AMLLibrary/Jars/jwi.jar'
        )
       }
{
 is
 if thisLeft.name = thisRight.name then
  1.0
 else
   ''.jwiSimilarity(thisLeft.name.properTerm, thisRight.name.properTerm)
 endif
}
```

4.4.2.2 Constraint-based heuristics

TypeElement compares the types of two properties by means of the `isEqualTo` helper. The helper verifies if the method input mapping model contains an equivalence between the compared types.

Listing 4.23: TypeElement

```
sim TypeElement ()
{
 is
 if thisEqualModel.isEqualTo(thisLeft.type, thisRight.type) then
  1
 else
  0
 endif
}
```

Multiplicity compares the multiplicity of properties. It has been inspired by the *cardinalities* heuristic described in Section 2.4.3.2.

Listing 4.24: Multiplicity

```
sim Multiplicity ()
```

[8] We have used the transformation proposed in [97].
[9] http://lyle.smu.edu/ tspell/jaws/index.html

```
{
 is
 thisModule.multTable.get(
  Tuple {
   left = Tuple {lower = thisLeft.lower, upper = thisLeft.upper},
   right = Tuple {lower = thisRight.lower, upper= thisRight.upper}
  }
 )
}
```

4.4.2.3 Structure-level heuristics

Statistics has been inspired by [62]. This computes the Euclidean Distance between two vectors that contain statistical data about classes, i.e, number of superclasses, attributes, and siblings.

Listing 4.25: Statistics

```
sim Statistics ()
ATLLibraries {
 (name = 'Vectors', path = '../AMLLibrary/ATL/Helpers'),
 (name = 'Math', path = '../AMLLibrary/ATL/Helpers')
}
{
 is
  thisModule.distance(
   Sequence{thisLeft.ParentsStatistic,
    thisLeft.ChildrenStatistic,
    thisLeft.SiblingsStatistic},
   Sequence{thisRight.ParentsStatistic,
    thisRight.ChildrenStatistic,
    thisRight.SiblingsStatistic}
  )
}
```

4.4.2.4 Instance-based heuristics

AttributeValues compares attributes that have the same primitive type, e.g., string, integer. The similarity of two attributes depends on how similar their corresponding instances are. We compare attribute instances as simple labels.

Listing 4.26: AttributeValues

```
sim AttributeValues (left : m1,right : m2)
ATLLibraries{(name='Strings', path='../AMLLibrary/ATL/Helpers')}
JavaLibraries{(name='match.SimmetricsSimilarity',
        path='../AMLLibrary/Jars/simmetrics.jar'
       )
       }
{
 is
 if thisLeft.isAttribute and thisRight.isAttribute then
  if thisEqual.model.isEqualTo(thisLeft.type, thisRight.type) then
   -- aggregation of similarity of instances
   thisLeft.owner.allInstancesFrom('left')
    ->iterate(instClass1; acc1 : Real = 0.0 |
     acc1 + thisRight.owner.allInstancesFrom('right')
      ->iterate( instClass2; acc2 : Real = 0.0 |
        if instClass1.refGetValue(thisLeft.name).oclIsUndefined() or
           instClass2.refGetValue(thisRight.name).oclIsUndefined() then
           0
        else
```

```
20            if instClass1.refGetValue(thisLeft.name).toString()
21              =
22              instClass2.refGetValue(thisRight.name).toString() then
23             1
24            else
25             0
26            endif
27           endif
28         )
29       )
30      else
31       0
32      endif
33    else
34     0
35    endif
36  }
```

4.4.3 Selection heuristics

ThresholdMaxSim selects a mapping when its similarity satisfies the range of tolerance $[Threshold - Delta, Threshold]$. We have borrowed `ThresholdMaxSim` (along with deltas and thresholds) to Do [62]. According to [62] (pag. 114) the best delta and threshold are 0.008 and 0.5, respectively.

Listing 4.27: ThresholdMaxSim

```
1 sel ThresholdMaxDelta ()
2 {
3  when
4  thisSim > 0.5
5  and
6  thisSim >= thisEqualModel.mapRangeByLeft.get(thisLeft).maxD
7  and
8  thisSim <= thisEqualModel.mapRangeByLeft.get(thisLeft).max
9 }
```

4.4.4 User-defined heuristics

We have implemented the heuristics `CreationByFullNameAndType-CreationAddedDeleted` and `SetLinks-SetLinksFiltering` like external ATL transformations. The former pair of heuristics embeds creation logic, and the latter instance-based similarity functionality. In particular, `CreationByFullNameAndType-CreationAddedDeleted` are useful to match large metamodels.

CreationByFullNameAndType creates mappings when two elements has the same metametamodel type and full name. A full name is a string similar to the one built by the *path comparison* heuristic (see Section 2.4.3.2). For example, the full name of `transition` reference is `PetriNet|Net|transition` because the `PetriNet` package contains the `Net` class, and this class contains the `transition` reference. The heuristic marks the elements not satisfying the condition like `Added` ($LeftModel$) or `Deleted` ($RightModel$).

CreationAddedDeleted performs a Cartesian Product between the `Added` and `Deleted` elements computed by `CreationByFullNameAndType`.

SetLinks-SetLinksFiltering have been inspired from [99]. Their purpose is to test the intersection of instance sets (e.g., A and B). [99] defines 5 kinds of intersection: equal (A ? B = A = B), contains (A ? B = A), contained in (A ? B = B), disjoint (A ? B = 0), and overlaps.

4.5 Summary

This chapter presented AML points of view going from its abstract concepts to its implementation details. Below we position AML with respect to the first issue described in Section 3.4 and some additional features that a DSL has to satisfy.

- **Reusability of heuristics.** Whereas most of related work uses GPLs for matching techniques development, we use a DSL. We have developed 24 heuristics, 100% of these can be used in M2-to-M2 matching algorithms, and 37% can be used in M1-to-M1 matching algorithms. The AML contructs allow a loosely coupling of code to reference model. This, in turn, promotes reuse of heuristics.

- **Declarativity.** AML considerably reduces the number of lines of ATL (or Ant) code to be written. For each matching heuristic (`create`, `sim`, `aggr`, and `sel` methods) included in an algorithm, AML saves the codification of 80 lines of ATL code, and 10 lines of Ant code.

- **End-user experience.** The m2m newsgroup already shows some activity around AML. Given the user inquiries posted at the time we wrote the thesis, we can not judge whether AML improves end-user experience or not. What we can say is that AML is a first approximation of DSL to facilitate matching techniques reuse. Since AML is an open-source tool, developers can download it and share their user experience in newsgroups. User feedback will give us more elements about AML usability.

- So far, we have widely applied an MDE toolkit (AmmA) to AML development. Some lessons learned from the experience follow:

  - AmmA is a powerful suite to implement domain specific languages. Metamodels and transformations facilitate the separation of concerns; metamodels express the concrete syntax of languages, and HOTs bridge the gap between DSL programs and executable code.
  - When regarding HOTs, we believe that it is hard to manipulate them. There is a need for a DSL that facilitates HOT development.

Chapter 5

Automatic Evaluation of Model Matching Algorithms

This chapter gives our approach addressing the evaluation issues raised in Section 3.4.2. Firstly, we describe an approach automatizing the evaluation of model matching algorithms. Just as the OAEI framework [22], our approach covers the preparation, execution, and evaluation phases. The features of our approach follow: extraction of test cases from model repositories, execution of AML matching algorithms, and use of megamodels to automate the whole evaluation process. Moreover, the chapter states how the approach can be profitable to ontology community. Secondly, we present the approach validation.

5.1 Approach overview

Fig. 5.1 shows the artifacts manipulated by our approach along the evaluation phases. The artifacts are stored in a model repository.

- **Preparation.** The approach constitutes a given test case using two metamodels, m and m', and a transformation $tm2m'$ written in terms of these models. A reference alignment R is extracted from $tm2m'$ (block e in Fig. 5.1).
- **Execution.** We develop a matching algorithm f with AML.
- **Evaluation.** The approach executes f which computes a candidate alignment A' from m, m', A, r, and p. Finally, the block c compares A' to R and derives matching metrics $M(A', R)$.

Our approach uses a megamodel to constitute many test cases, launch a set of matching algorithms, and compute metrics. The next sections describe our approach in detail.

5.2 Preparation: getting test cases from model repositories

In this phase, our approach prepares a set of test cases. Each test case consists of a pair of metamodels and a reference alignment.

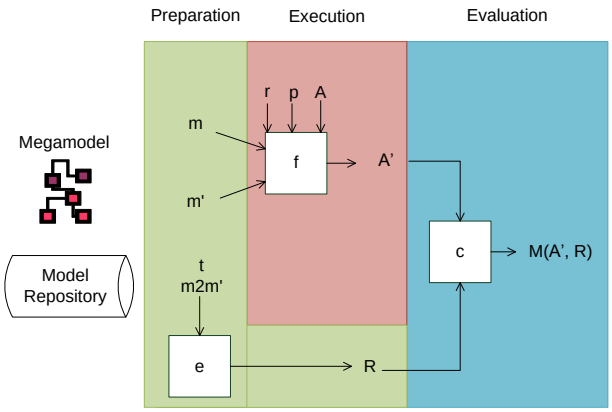


Figure 5.1: A approach to automate matching system evaluation

5.2.1 Discovering test cases

The megamodel is the cornerstone of our approach. Before working with the megamodel it has to be populated. Since a model repository is continuously growing, we have implemented a strategy to automatically populate the megamodel. The strategy consists of the following steps:

1. A program parses a set of files containing metadata. The metadata describes, for example, in terms of what metamodels a transformation is written. The output of the parsing is a text file.

2. A textual syntax tool translates the text file into a megamodel [100].

3. Because the megamodel refers to more models than the ones we are interested in, a program refines the megamodel to keep only the records related to transformations and their corresponding metamodels.

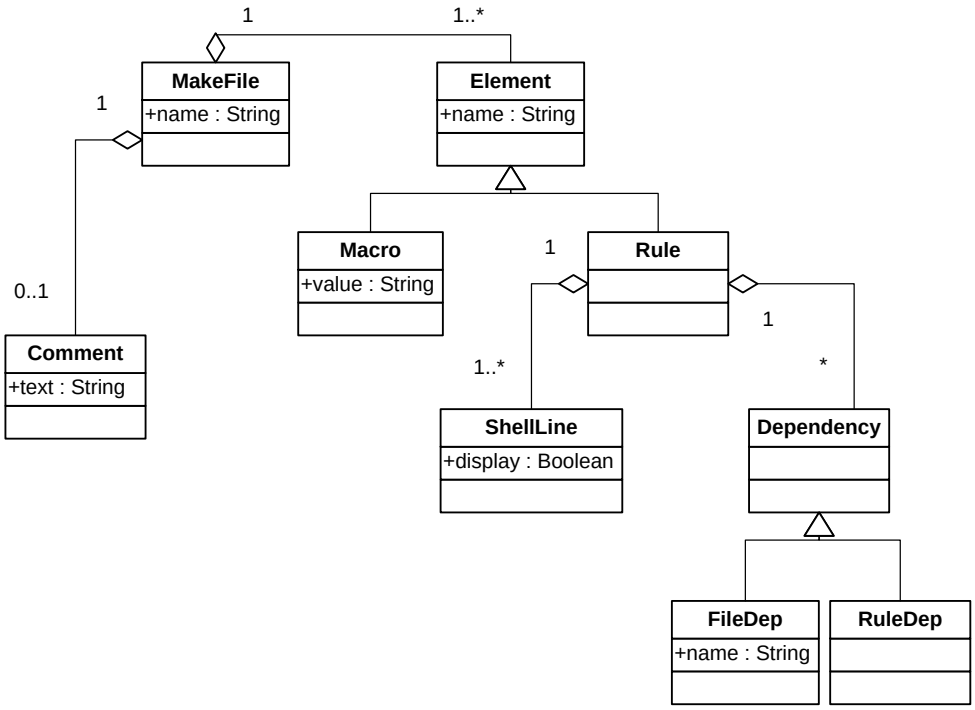
Fig. 5.2 shows the metamodels of an extracted test case. For instance, the Ant metamodel contains the class `Project`, the attribute `message`, and the reference `default`.

5.2.2 Extracting reference alignments from transformations

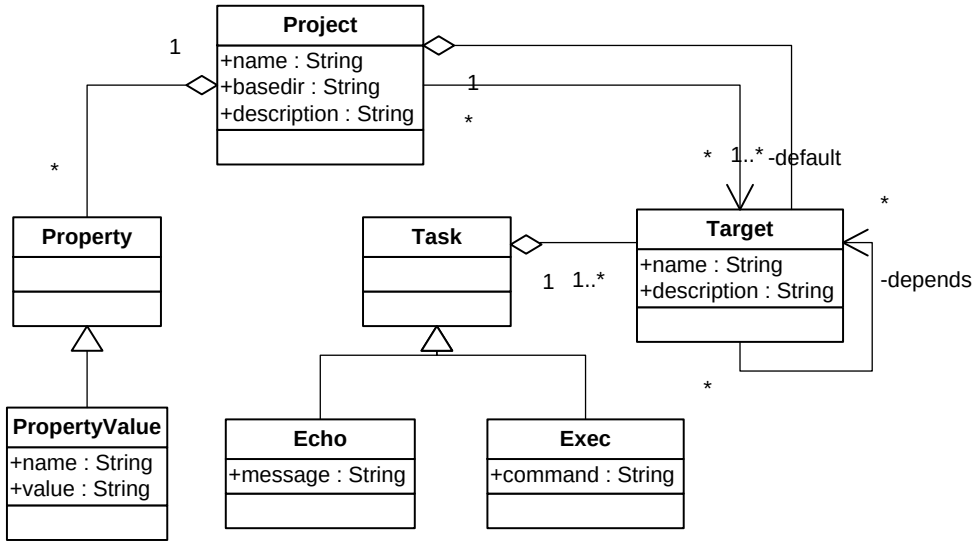
Having the megamodel, the approach extracts reference alignments from transformations. A transformation may be available in various flavors, i.e., being developed by programmers geographically distributed, or using model transformation languages different each other. Thus, we should define the extraction scope by answering the questions: 1) in what language are the available transformations developed?, and 2) what patterns are embedded in them?

For example, we have found the following patterns in ATL transformations (Listing. 5.1 shows a concrete example):

- Each transformation contains a set of rules (lines 1-9).



(a) Make metamodel



(b) Ant metamodel

Figure 5.2: Metamodels of a test case

- Each rule consists of a pair of patterns, i.e., `inPattern` and `outPattern`. The `inPattern` matches a class of m (line 3), and the `outPattern` a class (or set of classes) of m' (line 5).

- Each `outPattern` is composed of a set of bindings. A binding initializes a rightClass property using a structural feature of leftClass. We have characterized four types of bindings:

  1. `rightStructuralFeature` ← a `leftStructuralFeature` (line 6).
  2. `rightStructuralFeature` ← an OCL [34] iteration expression which includes a `leftStructuralFeature` (line 7).
  3. `rightStructuralFeature` ← an operation (i.e., a helper expression) involving a `leftStructuralFeature`.
  4. `rightStructuralFeature` ← a variable referring to another `rightClass` created in the `outPattern`.

Note that the more we figure patterns out, the better is the quality of reference alignments. The quality can be further increased by extracting alignments not only from one transformation but also from a set of transformations $tm2m'$ if they are available.

Listing 5.1: Excerpt of the transformation Make2Ant

```
1 rule Makefile2Project{
2  from
3   m : Make!Makefile
4  to
5   a : Ant!Project(
6    name       <- m.name,
7    targets    <- m.elements -> select(c | c.oclIsKindOf(Make!Rule))
8   )
9 }
```

There are ways to bridge the gap between modeling artifacts and ontologies. Tools like EMFTriple [24] can translate metamodels to ontologies and vice versa. Moreover, a transformation can translate our alignments to the format established by the Alignment API [85]).

5.3 Execution: implementing and testing matching algorithms with AML

The execution phase typically involves implementation, testing, and deployment[1] of matching algorithms. In our approach, the execution only embraces development and testing. We postpone the deployment to the evaluation phase. The previous chapter has already presented the development and testing facilities provided by AML.

[1] We call deployment the stage that executes algorithms in order to produce the final matching results.

5.4 Evaluation: deploying and assessing AML algorithms

Based on a megamodel, our approach automatically deploys algorithms and computes metrics. The megamodel indicates the algorithms to be executed, the required inputs, and the reference alignments. Our approach computes the matching metrics introduced in Section 2.4.4, i.e., precision, recall, and fscore. Moreover, this approach updates the megamodel with records associating A', R, and $M(A', R)$.

The use of DSLs and megamodels facilitates the evaluation phase. Since there is an unique programming interface, launching the algorithms become easier. By using the megamodel, our approach can execute algorithms over a large set of test cases and to obtain results by itself. This promotes a more confident evaluation of performance and accuracy.

The ontology community has contributed several tools to compute sophisticated matching metrics (e.g., PrecEvaluator [12]). We can benefit from these tools by translating our alignments to their formats.

5.5 Validation

This section recapitulates experiments whose objectives are:

1. Show that our approach can automatically extract a large set of matching test cases from a model repository. Such test cases, named modeling dataset, are useful to evaluate not only AML strategies but also other matching systems.

2. Demonstrate that AML can be used to customize matching strategies including diverse kinds of heuristics that exploit, for example, linguistic/structural information and sample instances.

3. Validate that AML strategies can be applied to modeling datasets but also to ontology datasets.

4. Compare the quality of AML strategies with respect to other matching systems (e.g., MatchBox [6], Aflood [101], Aroma [102]).

The next four subsections address the mentioned objectives. We ran the cited experiments on a Debian GNU/Linux machine with a 3 GHz Intel Xeon processor and 70GB of RAM.

5.5.1 Modeling dataset

The approach described in Section 5.2 has built a megamodel referring to 68 ATL projects available on the "ATL Transformation Zoo". An ATL project consists of one or more model transformations and a set of models and metamodels. In our approach, each ATL project corresponds to a matching test case. The ATL Zoo is an open-source model repository contributed by the m2m Eclipse community [95].

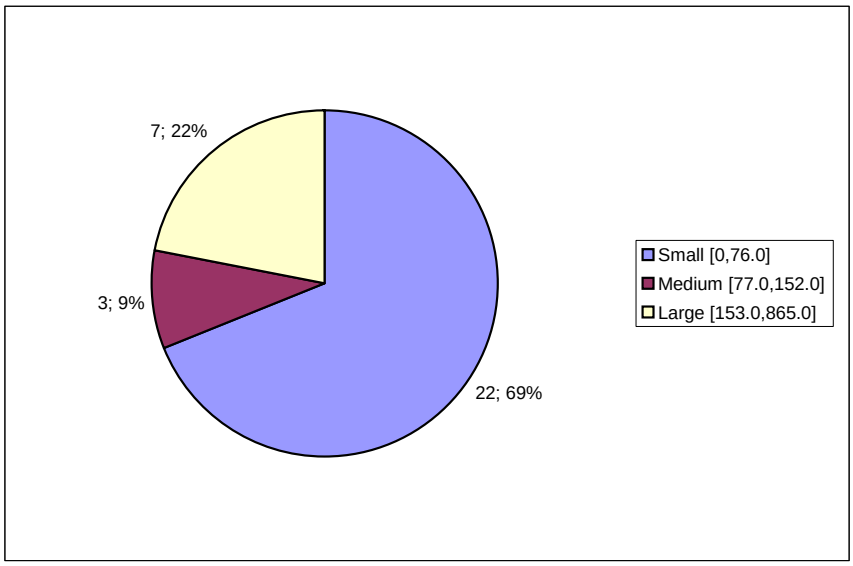


Figure 5.3: Number of metamodels involved in small, medium, and large series

Our experimentation only exploits 50% of the ATL Zoo (that currently has 103 projects), the reason is that some projects lack metadata necessary to sucessfully extract test cases from them. However, compared to MatchBox [6], that extracts alignments from the transformations stored in the same model repository, our approach is more robust; whereas MatchBox refers to 15 test cases (see [6], pag. 2287, par. 2), our approach considers 68 [2].

Fig. 5.3 shows the number of metamodels involved in the extracted test cases, i.e., 32. Some metamodels are used in more than 1 test case. We observed that these metamodels represent more a solution than a problem. For example, the metamodels describe a concrete technical space (e.g., XML, OWL, UML, SVG, etc.) or a tool (e.g., ATL, CodeClone). Few metamodels describe synthetic problem spaces (e.g., Families). Here we find a difference with respect to ontologies which mostly describe a problem space, e.g., conference, anatomy, bibliographic.

We have classified each metamodel in a series (small, medium, or large). Fig. 5.3 depicts an interval for each series. The interval establishes the number of elements (i.e., `Class`, `Attribute`, and `Relations`) that a metamodel has to have in order to belong to a given series. Most of metamodels belong to the small series.

Having a precise definition of reference mappings is often impossible [92]. For example, when experts manually establish reference mappings they have their own intentions. Thus, their alignments can differ from other experts' ones. In our case, two aspects can impact the validity of reference mappings automatically extracted from transformations:

- **Complexity** means that a given transformation implements imperative patterns (rules with only an `outPattern`). In contrast, our mechanism relies on declarative patterns (as described in Section 5.2.2, rules with `inPattern` and `outPattern`). Examples are the `Measure2Table` and `UML2Measure` transformations which mostly

[2]Test cases available at http://docatlanmod.emn.fr/AML/TestCases/testcases.zip

contain imperative code. The complexity of these transformations explains why the extracted reference alignments contain very few correspondences.

- **Incompleteness** means that a transformation lacks items. The justification is that developers write transformations in terms of the data instances they expect to have in source and target models. If the data instances are not relevant, developers do not written rules associating certain metamodel concepts.

Although these aspects might question the validity of our approach, we believe that it remains applicable. Our reference alignments have in average 19 correspondences. Only 1 out of 68 transformations gave an empty reference alignment. Given our large dataset, having experts to manually check correspondences validity may be expensive. An alternative would compare our correspondences to the ones extracted by MatchBox.

5.5.2 Diversified matching strategies

We have tested 6 AML algorithms, i.e., 4 metamodel-only based and 2 instance-based. The algorithms combine heuristics exploiting linguistic, structure, and data instances. Although we have contributed a library of 24 heuristics, we have experimented in detail only 10. We have chosen some heuristics because they have given accurate results on previous work [62][77][3]. The next paragraphs describe the accuracy and performance obtained by our algorithms when applied to the extracted modeling test cases.

5.5.2.1 Metamodel-only based algorithms

We have applied to the modeling dataset the following metamodel-only based algorithms: `WordNet_SF_Both`, `Lev_SF_Thres`, `WordNet_SF_Thres`, `Lev_SF_Both`). Each algorithm combines 7 heuristics. The algorithms keep the same creation, aggregation, and structure-level similarity methods and vary linguistic-based and selection ones. Table. 5.1 shows both, constant and variant methods, and Listing. 5.2 and Listing. 5.3 present how the heuristics interact to deliver the mappings.

	Similarity			
Creation	**Linguistic-based**	**Structure-based**	**Selection**	**Aggregation**
CreationByFullNameAndType	Levenshtein	SimilarityFlooding	ThresMaxSim(0.5) + Delta(0.008)	WeigthedSum
CreationAddedDeleted	WordNet		BothMaxSim	Merge

Table 5.1: Heuristics combined in metamodel-only based algorithms

Listing 5.2: Lev_SF_Thres

```
1 modelsFlow {
2
3  tp = CreationByFullNameAndType[map]
4  adddel = CreationAddedDeleted[tp]
5  inSF = Levenshtein[adddel]
6  outSF = SimilarityFlooding[]
7  tmpresult = WeightedAverage[0.5:inSF, 0.5:outSF]
8  result = ThresholdMaxSim[tmpresult]
9  merge = Merge[1.0:tp, 1.0:result]
10
11 }
```

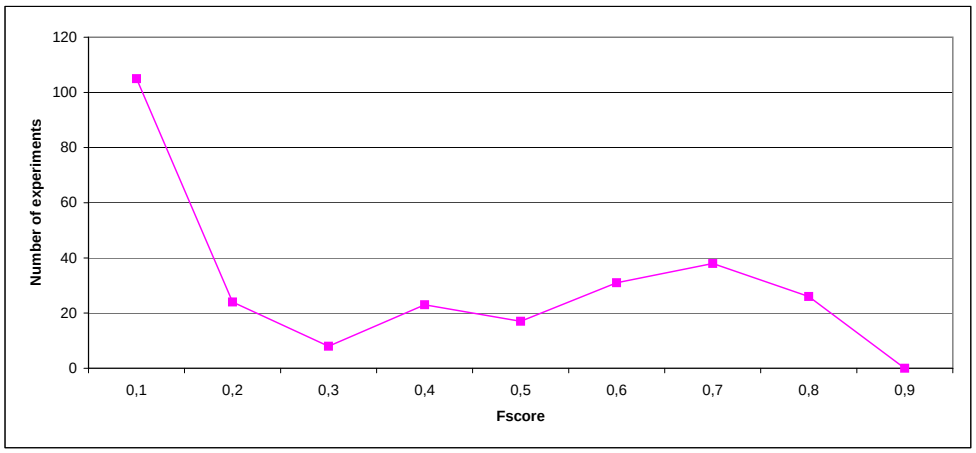


Figure 5.4: Quality distribution of experiments

Listing 5.3: WordNet_SF_Both

```
modelsFlow {

 tp = CreationByFullNameAndType[map]
 adddel = CreationAddedDeleted[tp]
 inSF = WordNet[adddel](param)
 outSF = SimilarityFlooding[]
 tmpresult = WeightedSum[0.5:inSF, 0.5:outSF]
 result = BothMaxSim[tmpresult]
 merge = Merge[1.0:tp, 1.0:result]

}
```

Fig. 5.4 shows the distribution of the 272 (4 algorithms tested on 68 test cases) performed experiments with respect to different fscore ranges. The line chart indicates that the algorithms achieved a low fscore (i.e., <0.1) in around 100 experiments and a higher fscore in 172.

Fig. 5.5, in turn, depicts the distribution of experiments with regarding the fscore of each strategy. All the strategies have two common behaviors. Firstly, they are represented in fscores ranging below 0.8. Secondly, whilst the strategies give a good fscore (>=0.5) to 35% of experiments, they obtained a low fscore (< 0.5) for the rest.

In addition to the average fscore, Fig. 5.6 shows the average recall and precision of our algorithms. The average fscore tends to 0.3, the `Lev_SF_Both` fscore is gently up this value. The Both-based strategies give the highest recalls (i.e., 0.6 and 0.7), and the Threshold-based strategies give the best precision (i.e., almost 0.3). Looking at the `Lev_SF_Both` results in detail shows that the fscores obtained in the small test cases (going to 0.8) are higher than the large test cases' ones (going to 0.5). The rationale is that most of extracted gold standards, associated to large test cases, contained a few correspondences. If computed correspondences exceed gold standards then precision (and fscore) goes down. Section 5.5.5 explains why our approach sometimes extracts poor gold standards. This mostly happens when model transformations are complex or involve large metamodels.

Table. 5.2 lists the time required for our algorithms for matching the modeling dataset. The fastest algorithm was `Lev_SF_Thres`. In general, the runtime values mentioned in

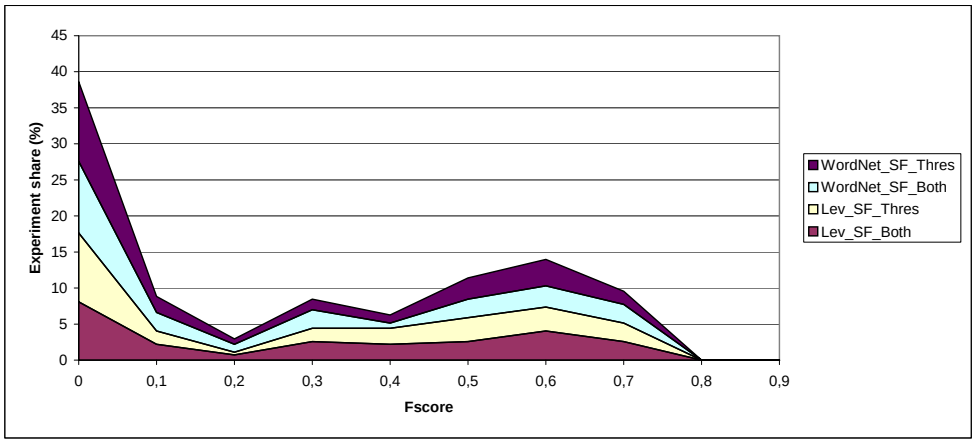


Figure 5.5: Experiment distribution for metamodel-based only strategies

Figure 5.6: Matching metrics of metamodel-only based algorithms

the chapter include not only the matching process but also the creation of empty input mapping models, extraction of reference alignments, and metrics computation.

Algorithm	Runtime
Lev_SF_Both	275 min. 57 sec.
WordNet_SF_Thres	122 min. 42 sec.
WordNet_SF_Both	282 min. 27 sec.
Lev_SF_Thres	114 min. 48 sec.

Table 5.2: Runtime metamodel-only based algorithms - modeling dataset

5.5.2.2 Instance-based algorithms

We have applied to the modeling test cases the following instance-based algorithms: `Sets` and `Sets_Lev_SF_Both`. Listing. 5.4 and Listing. 5.5 present the heuristics combined by each algorithm. Unlike the metamodel-only based algorithms, `Sets` and `Sets_Lev_SF_Both` do not invoke the full Similarity Flooding algorithm but two of its heuristics, i.e., `Propagation` and `SF`.

`Sets_Lev_SF_Both` combines `Sets` and `Lev_SF_Both`. Thus, the algorithm executes SF and Propagation twice; the purpose is to propagate instance-based and linguistic similarity. Line 8 of Listing. 5.5 presents the weights assigned to each similarity method, Levenshtein has the highest weight. We have combined `Lev_SF_Both` to `Sets`, because (according to the results) `Lev_SF_Both` was the more accurate metamodel-only based algorithm.

Listing 5.4: Sets

```
1 modelsFlow {
2
3  s = SetLinks[map](m1model, m2model)
4  f = SetLinksFiltering[s]
5  prop = Propagation[f]
6  sf = SF[f](prop)
7  outSets = Threshold[sf]
8
9 }
```

Listing 5.5: Sets_Lev_SF_Thres

```
1 modelsFlow {
2
3  tp = CreationByFullNameAndType[map]
4  adddel = CreationAddedDeleted[tp]
5  inSF = Levenshtein[adddel]
6  outSF = SimilarityFlooding[]
7  outSets = Sets[]
8  tmpresult = WeightedSum[0.4:inSF, 0.3:outSets, 0.3:outSF]
9  result = BothMaxSim[tmpresult]
10 merge = Merge[1.0:tp, 1.0:result]
11
12 }
```

We have experimented the instance-based algorithms over a subset of the modeling dataset described in Section 5.5.1, that is, 22 out of 68. Even though the megamodel has all the information about the inputs needed for the instance-based algorithms, the execution of the instance-based algorithms failed in some test cases. The reason is that such test cases involve models not stored in the repository. For example, some ATL Zoo

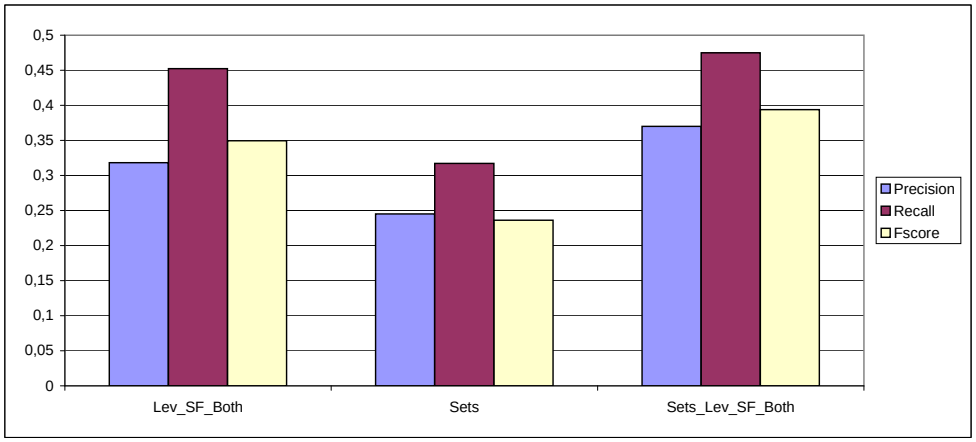


Figure 5.7: Matching metrics of instance-based algorithms + Lev_SF_Both

contributors do not post output models which are intended to be generated by transformations. Because the generation of missing models is an extra workload, we have applied the instance-based algorithms to only 22 test cases. Note that it remains to be a good dataset.

In addition to the matching metrics of `Sets` and `Sets_Lev_SF_Both`, Fig. 5.7 shows the metrics of `Lev_SF_Both` when executed over the same test cases. Fig. 5.7 illustrates that the use of linguistic information and data instances increases matching algorithm accuracy. Fscore goes up from 0.35 to 0.4 when the algorithm combines `Sets` results to `Lev_SF_Both`. Finally, the execution of instance-based algorithms took less than 2 min. `Sets` and `Sets_Lev_SF_Both` have been tested on a dataset smaller than the metamodel-only based algorithms' one. In addition, they reuse the empty mapping models and the reference alignments created during the execution of metamodel-only based algorithms. It partially explains why the performance is high.

5.5.3 Ontology dataset

The AML algorithms have been tested over datasets from the modeling and ontology community. It is natural to select a modeling dataset since our algorithms were originally planned to match (meta)models. We have chosen an ontology dataset to show that our approach can be used in other technical spaces too. We have picked ontologies instead of (for example) schemas due to the existence of a large ontology dataset and an initiative, the OAEI, that reports the results of matching systems over the dataset.

The ontology dataset corresponds to the OAEI conference track. We have selected it because it does not contain individuals within. It facilitates the transformation from ontologies to metamodels. Note that EMFTriple [24] seems to support the translation ontology - metamodel, even containing individuals in. However, its developer, Guillaume Hillairet, has spelled out that further development is necessary to make the tool more robust in that sense.

Among the 21 test cases provided by the conference track, we have chosen 3, i.e., cmt-ekaw, cmt-sigkdd, and ekaw-sigkdd. Translation from the conference ontologies into metamodels was difficult, as EMFTriple lacks documentation describing its installation

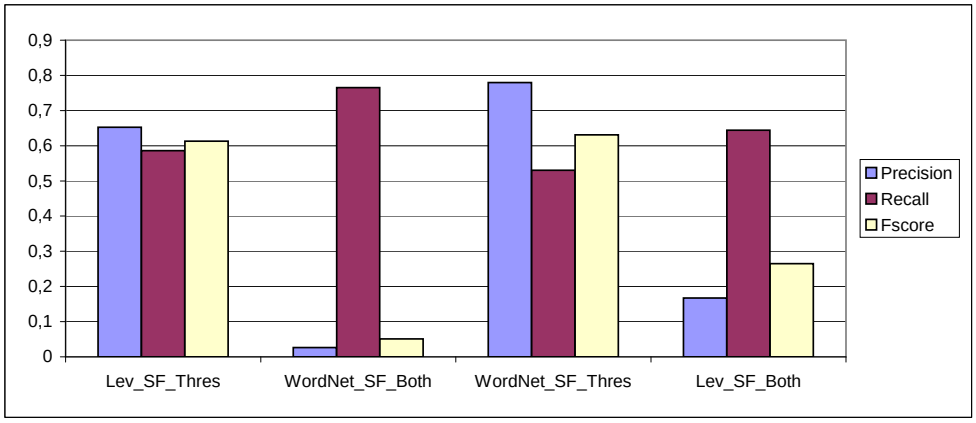


Figure 5.8: Matching metrics of metamodel-only based algorithms - conference track

and functionalities. So that, we have chosen the 3 test cases whose metamodels the EMFTriple developer was available to provide us.

Table. 5.3 shows the metamodels, involved in the 3 selected test cases, and their corresponding sizes.

Model	Size
sigkdd	93
cmt	100
ekaw	127

Table 5.3: Size of ontologies

Fig. 5.8 depicts the quality of 4 AML algorithms over the ontology dataset. `WordNet_SF_Thres` and `Lev_SF_Thres` tend to 0.6, i.e., the best fscore.

Fig. 5.9 is a Precision and Recall (PR) curve. It allows us to understand how precision varies from recall [103]. The graph has been built from the results over the ekaw-sigkdd test case. For the 4 algorithms, precision remains constant as recall ranges from 0.1 to 0.6. From 0.6, the precision of `WordNet_SF_Thres` and `Lev_SF_Thres` dramatically goes down. This also happens to `Lev_SF_Both` and `WordNet_SF_Both` in 0.7 and 1, respectively. This graph partially differs from a classical PR curve where precision decreases as recall increases. Instead of that, the curves of our algorithm curves are stable in a range, and then they decrease. The stabilization means that precision is globally increasing with recall. Moreover, the PR curve hints what algorithm to choice in what case; if one wants to obtain a high recall, then `WordNet_SF_Both` or `Lev_SF_Both` are useful. On the other hand, if one wishes a good precision, then one should select `WordNet_SF_Thres` or `Lev_SF_Thres`.

The PR curves of cmt-ekaw and cmt-sigkdd are quite similar to Fig. 5.9. The difference is the interval where precision stays the same, and the point where precision suddenly decreases (this corresponds to the maximum value of the interval). For cmt-ekaw, the interval is [0.1, 0.4], and for cmt-sigkdd is [0.1-0.5]. These results indicates that cmt-ekaw is the hardest ontology matching task we have experimented. We have used the Alignment API 4.0 to generate Precision and Recall curves.

Table. 5.4 mentions the time taken by our algorithms for matching the ontology dataset. In general, the algorithms required less than 1 minute, the fastest was `Lev_SF_Both`.

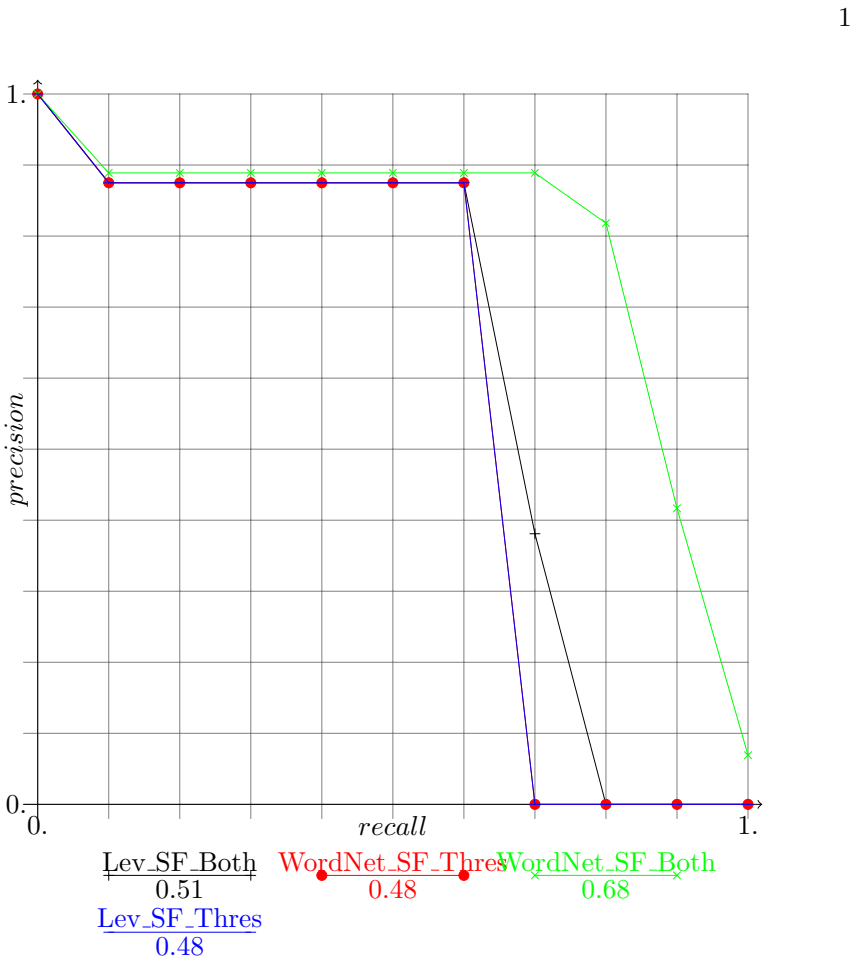


Figure 5.9: Precision and recall curve for metamodel-only based algorithms - ekaw-sigkdd test case

Algorithm	Runtime (sec.)
Lev_SF_Both	54
WordNet_SF_Thres	57
WordNet_SF_Both	56
Lev_SF_Thres	55

Table 5.4: Runtime metamodel-only based algorithms - conference track

5.5.4 AML algorithms versus other matching systems

For each kind of dataset (metamodels or ontologies), we have compared AML algorithms either to MDE matching systems (i.e., MatchBox [6]) or to ontology systems (e.g., Aflood, Aroma, etc.). We have chosen MatchBox because it has been applied to a modeling dataset similar to our own dataset. In addition, we have preferred Aflood, Aroma, etc, because their results over the conference track are publicly available.

5.5.4.1 Modeling dataset comparison

We have taken a look to the MatchBox results over the modeling test cases. Based on the data mentioned in [6], we have built a histogram showing the fscore that a MatchBox algorithm obtains on 6 test cases. Although [6] reports the results over more test cases, our histogram only includes the test cases overlapping our modeling dataset.

For the sake of comparison, the histogram contains the `Lev_SF_Both` algorithm results over the same 6 test cases. For 1 test case (i.e., Make-Ant), our algorithm obtains a higher fscore than MatchBox. For the rest of test cases, our algorithm gives fscores which are slightly under the MatchBox results; the delta ranges from 0.03 to 0.16. Because the comparison of AML and MatchBox has been done over a small dataset, we think that it is premature to say that MatchBox is better than AML in terms of accuracy. It is necessary to compare AML to MatchBox given the same dataset (it has to be larger). As stated in [6] (page 2287), this would be possible if MatchBox is extended to cover a larger spectrum of test cases. Other option would be to merge our reference alignments to the MatchBox ones. Finally, there is not a formal document reporting MatchBox runtimes, a future comparison has to consider performance too.

5.5.4.2 Ontology dataset comparison

We have compared our results to those obtained by other systems participating in the conference track of the OAEI 2009 campaign [3]. Aflood [101] provides the best fscore (=0.6) which is slightly under the `WordNet_SF_Thres` fscore. In contrast to the AML algorithms, predisposed toward a good recall, the compared ontology systems tended toward a high precision. With respect to runtime, [93] does not report the time required for the systems matching the conference track. Since the comparison has taken into account only 3 pairs of ontologies (instead of the full track), we can not conclude about the accuracy of AML algorithms with respect to existing ontology matching systems. In contrast, this comparison illustrates that AML algorithms can be applied to pairs of ontologies. For a more robust comparison, future research needs to improve the projectors (e.g., EMFTriple) that translate ontologies into metamodels.

[3]Readers interested on these results may want to take a look to [93]

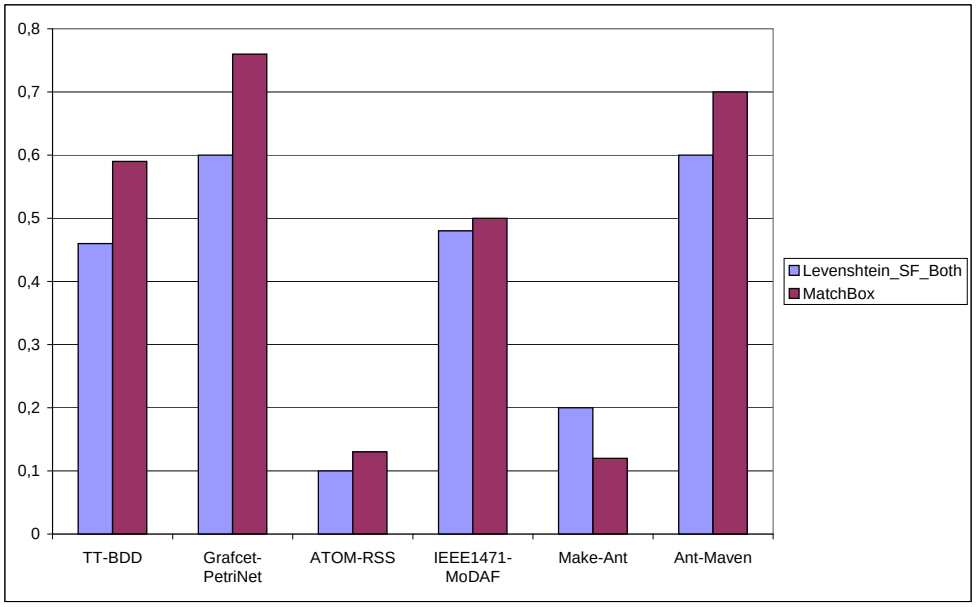


Figure 5.10: Fscores of Lev_SF_Both and MatchBox on 7 modeling test cases

5.5.5 Discussion

Let us elaborate on the experimented heuristics and algorithms:

- **Linguistic-based similarity heuristics.** The results of our 4 metamodel-only based algorithms show that the `WordNet` and `Levenshtein` heuristics were both quite accurate to match ontology test cases. The `WordNet` heuristic was less accurate to match metamodels. Because the experimented metamodels mostly describe technological concepts, the `WordNet` heuristic could not to find concepts in WordNet which is a general purpose dictionary. This suggests the need for a more specific dictionary to match the experimented metamodels.

- Related to the **selection heuristics**, `BothMaxSim` biases toward a high recall (i.e., mapping model completeness) and `ThresholdMaxSim` toward a good precision (i.e., mapping model correctness). Given the concepts a and b, `BothMaxSim` aligns a to b if (a, b) has the highest similarity value among all the correspondences involving a and b. The output mapping model can contain correspondences even if the similarity values are very low, i.e., 0.1. `ThresholdMaxSim`, in turn, selects (a, b) if its similarity value is in the range [0.492, 0.5]. Thus, a `BothMaxSim` output mapping model tends to contain more correspondences than a `ThresholdMaxSim` one. The `BothMaxSim` output mapping model keeps not only correct correspondences with a week similarity but also false negatives. As a result, `BothMaxSim` promotes a good recall but impacts precision. `ThresholdMaxSim` has the opposite behavior. To improve the precision of our algorithms, we will associated a threshold to `BothMaxSim`.

- The results of our 2 **instance-based algorithms** show that the instances are useful to strengthen the results of linguistic-based similarity heuristics. Note that the

instance-based heuristics use linguistic methods too. Thus, the experimented algorithms mostly rely on linguistic information (i.e., semantical and syntactical) to discover correspondences.

Metamodel-only based algorithms obtained a good fscore (>=0.5) to 35% of the experimentation over the modeling dataset. For the remaining test cases, the algorithms obtained a low fscore. The reasons behind it are:

- **The suitability of AML algorithms**. We have experimented only 6 algorithms over a large modeling dataset. Although the algorithms combine heuristics and thresholds reported as good by previous work [62][77], it is necessary to configure and test a larger spectrum of algorithms. Thus, one can see better the accuracy of a given algorithm in regard to others.

- **The quality of reference alignments**. For some pairs of metamodels, the AML algorithms gave many correct mappings, however poor reference alignments extracted by our approach could impact fscore. As explained in Section 5.5.1, it can happen when transformations are complex or incomplete.

At last, these experimentations validate that our algorithms can match modeling or ontology datasets:

- For the **modeling dataset**, the accuracy of our algorithms is gently under the MatchBox accuracy. From these results, we could not conclude about the quality of AML M2-to-M2 matching algorithms with respect to other MDE systems. MatchBox, in turn, reports the accuracy over only 6 out of 68 test cases extracted from the modeling repository [6]. Thus, we do not know if MatchBox is good for the rest of test cases. In a nutshell, it is necessary to straightforwardly test MDE algorithms over a common and large modeling dataset.

- For the **ontology dataset**, we can neither infer our algorithms are better than other ontology systems nor worse. More benchmarks over full sets of test cases (even test cases involving instances) are still necessary. To this end, more efforts have to be devoted in order to improve existing projectors from ontologies to metamodels and vice versa. Section 7.2 elaborates on this potential future work. Besides accuracy, future comparisons have to take into account runtime too.

5.6 Summary

We presented an approach to automate the evaluation of model matching systems. We conclude the following from the validation of our approach:

- **Modeling repositories: a growing and free source of matching test cases**. Our approach extracted 68 test cases from the "ATL Transformation Zoo". The number of ATL Zoo test cases raises every year. In addition, MDE community makes efforts to constitute other open-source model repositories, for example, the

Repository for Model-Driven Development (ReMoDD)[4]. Thus, one finds a potential source of matching test cases in the modeling repositories. Just as MDE community can benefit from these matching test cases, ontology community can do the same. We have used the EMFTriple tool to translate metamodels into ontologies, and we have developed transformations to translate our alignments to the Alignment API format and vice versa. Complexity and incompleteness of model transformations can impact the validity of gold standards. A solution would ask experts to validate such gold standards or to compare them with the reference alignments extracted by other systems such as MatchBox [6].

- **Modeling techniques to increase evaluation efficiency.** Our approach automates the whole evaluation process, going from the discovery of test cases to the assessment of algorithms accuracy.

- **Interoperability**. This chapter showed that our approach can interact with ontology matching tools to leverage evaluation, notably, the phase of result validation. For instance, we have used the Alignment API to plot PR curves from our results. The other way around is feasible as well, i.e., the ontology community may want to use our tool for drawing various kinds of graphs such as bar chart, area chart, or line chart.

- **Evaluation loosely coupled to programming language.** By using a megamodel, our approach automatically evaluated matching algorithms over extracted test cases. Note that the evaluation is independent of AML; Besides AML algorithms, we have experimented the Java Alignment API methods. Thus, if an Ant Script can execute an algorithm, developed in a given programming language, then our approach can evaluate it. The only aspect to be verified is the format yielded by such an algorithm, if it is different from our format, then it is necessary to develop a transformation between them.

- **Reusable matching heuristics.** The experimentation described matching algorithms that reuse the same heuristics in diverse ways. This shows the possibility of composing and decomposing matching algorithms with AML. For example, whereas some algorithms used the entire Similarity Flooding algorithm, others used it partially. We have tunned such an algorithm with classical parameters, i.e., threshold = 0.5 and an iteration.

[4]http://www.cs.colostate.edu/remodd

Chapter 6

Three matching-based use cases

As mentioned in Section 2.4.3.2 mapping manipulation is very common when the matching process is ended. Manipulation depends on the application domain. For example, one may want to generate executable code from M2-to-M2 mappings or analyze mappings for taking a decision. This chapter presents a use case devoted to the first case (i.e., model co-evolution) and two use cases belonging to the second one (i.e., pivot metamodels in the context of interoperability tool and model synchronization).

The main purpose is to show how an approach can combine AML algorithms and other modeling techniques to solve interesting MDE problems. Whilst the model synchronization use case involves an M1-to-M1 matching algorithm, the rest of use cases includes an M2-to-M2 matching strategy.

The model co-evolution use case proposes a solution that adapts models to evolving metamodels. The use case includes a modification with respect to the work described in [28]; whereas [28] reports the use the ATL matching transformations to discover simple and complex metamodel changes, our use case shows how to implement such transformations with AML. Note that [28] mentions a family of heuristics which has inspired the AML constructs.

An MDE solution for tool interoperability typically develops metamodels for each tool, and transformations between these metamodels and a pivot metamodel. The pivot should facilitate the construction of transformations. The second use case presents an M2-to-M2 matching-based approach to evaluate whether a pivot metamodel has been correctly chosen.

Finally, the third use case depicts how AML algorithms can leverage model synchronization in the context of Software Product Lines [31]. Unlike the use cases mentioned above, where the prototypes were completely developed by us, we have provided considerable guidance to the contributors of the third use case [104].

Each use case is structured as follows: the addressed problem, a model matching-based solution, the AML algorithms involved in the solution, and an experimentation.

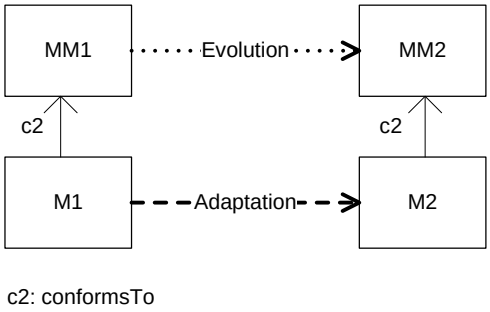


Figure 6.1: Metamodel evolution and model adaptation

6.1 Model co-evolution[1]

6.1.1 Problem

Software engineers usually have to adapt computer systems to technological and business changes. This need is rapidly increasing in systems built using the MDE paradigm. An MDE system basically consists of metamodels, terminal models, and transformations. The addition of new features and/or the resolution of bugs may change metamodels. The changes may break the consistency of related terminal models and transformations. In this work, we focus on terminal models consistency. Fig. 6.1 illustrates the problem: a metamodel *MM1* evolves into a metamodel *MM2* (see the dotted arrow). Our concern is to adapt any terminal model *M1* conforming to *MM1* to the new metamodel version *MM2* (see the dashed arrow).

The tough part of the problem is to adapt models when both simple and complex changes are involved. We explicitly distinguish two kinds of changes because complex changes need a more insightful adaptation that simple changes. Whereas a simple change describes the addition, deletion, or update of one metamodel concept, a complex change integrates a set of actions affecting multiple concepts[2]. The section proposes an M2-to-M2 matching-based solution to figure out simple and complex changes, and to adapt models to them.

6.1.2 Running examples

We have investigated the evolution of a Petri Net metamodel and the Netbeans Java metamodel. This section only describes the Petri Net metamodel, and how to represent it using the KM3 metametamodel. The Netbeans Java metamodel is fully depicted in our technical report [27].

This Petri Net example is based on the six metamodel versions provided by [105]. Fig. 6.2 and Fig. 6.3 illustrate versions 0 (*MM1*) and 2 (*MM2*), respectively. *MM1* represents simple Petri Nets. These nets may consist of any number of places and transitions. A transition has at least one input and one output place. *MM2* represents more complex Petri Nets. The principal changes between *MM1* and *MM2* are:

[1]Model co-evolution, model adaptation, and model migration are synonyms.

[2]The reader interested on examples of simple and complex changes may consult [27].

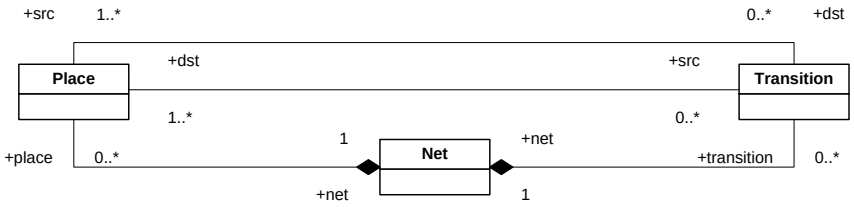


Figure 6.2: Petri Net *MM1* version 0

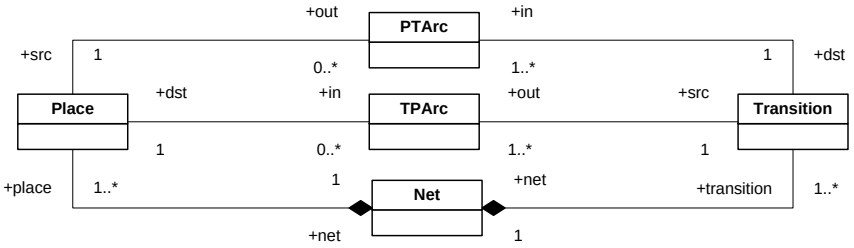


Figure 6.3: Petri Net *MM2* version 2

- References `place` and `transition` change their multiplicity from 0-* to 1-*.
- Classes `PTArc` and `TPArc` as well as references `in` and `out` are added.
- References `src` and `dst` are extracted from classes `Place` and `Transition`.

The extraction of the reference `dst` illustrates a complex change named *Extract class*. This implies to add and remove a reference, add a class, and associate classes. In considering these actions as isolated simple changes, we may skip changes without migrating involved data from *M1* to *M2*. In contrast, when we distinguish the complex change, we infer (for instance) that the added property (e.g., `dst`), contained in the new class `PTArc`, actually corresponds to the property `dst` removed from the class `Place`. Since we know the relationship between the properties we can migrate the data. We thus need to explicitly distinguish complex changes in order to properly derive adaptation transformations.

6.1.3 Solution involving matching

Our model adaptation approach adapts terminal models in three steps (Fig. 6.4). In the first step, an AML algorithm computes equivalences and changes between the metamodels *MM1* and *MM2*. In the second step, the AML algorithm output is translated into an adaptation transformation by using a HOT. Finally, the adaptation transformation is executed. Below we discuss the three steps in detail.

6.1.3.1 Matching equivalences and changes

Before giving the AML algorithm, we explain how the *equal* metamodel described Section 4.2.3 has been modified to represent simple and complex changes. The metamodel contains a package devoted to that, the package adds basic concepts such as `Deleted` and `Added` which mark a metamodel element as deleted/added from/into *MM1*. `Equal`, `Added`, and `Deleted` have been extended to describe more specific changes. For example, `EqualClass`, `EqualStructuralFeature`, `EqualReference`, `EqualAttribute` indicate the KM3

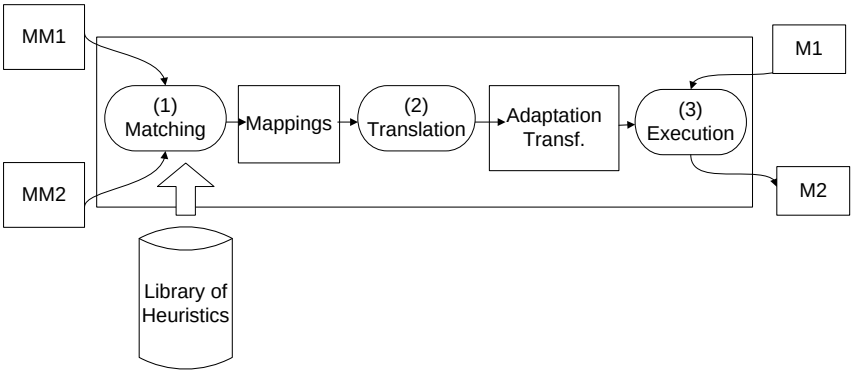


Figure 6.4: Approach for model co-evolution

types of `leftElement` and `rightElement`. The concept `AssociatedClassExtracted`, in order, links properties undergoing the change *Extract class*.

Listing 6.1: Metamodel representing types of changes

```
1  class Added extends WLink {
2   reference right container : RightElement;
3  }
4  class Deleted extends WLink {
5   reference left container : LeftElement;
6  }
7  ...
8  class EqualStructuralFeature extends Equal {}
9  class AssociatedClassExtracted extends EqualStructuralFeature {
10  reference associatedReference container : RightElement;
11 }
```

6.1.3.2 AML M2-to-M2 matching algorithm

Listing. 6.2 gives the AML algorithm delivering equivalences and changes. The algorithm includes heuristics described in Section 4.4.4. Here we elaborate on the new heuristics (lines 11-15).

Listing 6.2: AML algorithm matching metamodels, co-evolution use case

```
1  modelsFlow {
2
3   tp = CreationByFullNameAndType[map]
4   filtered = CreationAddedDeleted[tp]
5   prop = Propagation[filtered]
6   sf = SF[filtered](prop)
7   norm = Normalization[sf]
8   tmpresult = WeightedSum[0.5 : norm, 0.5 : filtered]
9   result = BothMaxSim[tmpresult]
10  merge = Merge[1.0:tp, 1.0:result]
11  diff = Differentiation[merge]
12  td = TypeDifferentiation[diff]
13  cl = ConceptualLink[td]
14  rw = Rewriting[cl]
15  f = FlattenFeatures[rw]
16
17 }
```

- `Differentiation` distinguishes between equivalent, deleted, and added metamodel elements. The heuristic compares metamodel elements to equivalences. The intuition is that not linked elements correspond to deletions and additions. `Differen-`

`tiation` marks unlinked *MM1* elements as deleted, and unlinked *MM2* elements as added.

- `TypeDifferentiation` decorates the input mapping model with the KM3 types of linked elements. For example, an `Equal` element associating classes turns into an `EqualClass`.

- `Rewriting` and `FlattenFeatures` reorganize the equivalences considering the relationships between the linked concepts: containment and inheritance. For instance, the *Rewriting* heuristic rewrites `(transition, transition)` as a child of `(Net, Net)` because of the containment relationship between these elements.

- `ConceptualLink` infers complex changes from equivalences, additions, and deletions. For example, Listing 6.3 shows a rule that verifies if the *Extract Class* change has happened. The rule assemblies two properties a (`AddedStructuralFeature`) and d (`DeletedStructuralFeature`) using the `AssociatedClassExtracted` type. The conditions are: 1) an introduced class owns the property a (line 5), and 2) this class is associated to other class that contains d (line 7).

Listing 6.3: Complex changes transformation excerpt

```
1 rule AssociatedClassExtracted {
2  from
3   d : EqualMM!DeletedStructuralFeature,
4   a : EqualMM!AddedStructuralFeature (
5       a.right.target.owner.isNewClass()
6       and
7       a.right.target.owner.isAssociatedTo(d.left.target.owner)
8    )
9  to
10   e : EqualMM!AssociatedClassExtracted
11 }
```

6.1.3.3 Translation to adaptation transformations

In this step, equivalences and changes are translated into an executable adaptation transformation via a HOT. The HOT takes as input the final mapping model, and generates as output a model transformation written in a particular transformation language (e.g., ATL, XSLT, SQL-like). The HOT follows the guidelines below:

- Yield a transformation rule for each `EqualClass` that links no abstract classes. The HOT takes referred left and right classes to yield input and output patterns.

- Create a binding for each `EqualStructuralFeatures` attached to an `EqualClass`. The binding complexity depends on the `Equal` type. While a simple `EqualStructuralFeature` generates a simple binding, `EqualStructuralFeature` extensions (e.g., `AssociatedClassExtracted`) generate more elaborated bindings. In general, sophisticated bindings instruments the code that adapt *M1* models to complex changes.

Listing. 6.4 shows an adaptation transformation, written in ATL, which is generated by a concrete HOT. This creates the transformation rule `Place2Place` (line 1) from the

Table 6.1: Size of metamodel illustrating the co-evolution use case

Example	PetriNet			Java		
Version	0	1	2	1.12	1.13	1.15
Elements	11	11	21	255	256	258

equivalence (`Place`, `Place`). The *from* part matches the elements conforming to `Place` (line 3). The *to* part creates elements conforming to `Place`. Moreover, the HOT generates a complex binding (see line 6) from the equivalence (`out`, `dst`). The binding calls an additional rule (i.e., `dstPTArc`) to initialize `dst` of `PTArc` (lines 18) using the values `dst` of `Place`.

Listing 6.4: Transformation excerpt (Petri Net example)

```
1 rule Place2Place {
2  from
3   pV1 : MM1!Place
4  to
5   pV2 : MM2!Place (
6     out <- pV1.dst -> collect (tV1 | thisModule.dstPTArc(tV1, pV1)}))
7    )
8 }
9 unique lazy rule dstPTArc {
10  from
11    transition : MM1!Transition,
12   place : MM1!Place
13   to
14    tV2 : MM2!PTArc (
15    dst <- transition
16   )
17 }
```

6.1.3.4 Adaptation transformation execution

This step simply executes the generated adaptation transformation. The transformation takes any terminal model *M1* and generates a terminal model *M2*.

6.1.4 Experimentation

Section 6.1 gives further details about the running examples. Section 6.1 provides the metrics to evaluate the results. Section 6.1 discusses the experimentation results. Finally, Section 6.1 shows the results of applying the EMF Compare tool to the running examples, and compares them to our results.

6.1.4.1 Dataset

We have results from experimentations which use 8 versions of the Netbeans Java metamodel, and 6 versions of a Petri Net metamodel provided by [105]. For the sake of readability, we just present the results in applying our approach on three versions of each metamodel. These versions are chosen because they contain significant changes. In the Java example, we choose the versions 1.12, 1.13, and 1.15. In the Petri Net example, we use the versions 0, 1, and 2. Table 6.1 shows the number of elements (classes, attributes and references) contained in the versions. We match the following couples of versions: 0

– 1, 0 – 2, 1.12 – 1.13, and 1.12 – 1.15.

6.1.4.2 Metrics

To measure the AML algorithm accuracy, we have used the classical matching metrics:

$$Precision(x) = \frac{CorrectFound(x)}{TotalFound(x)} \tag{6.1}$$

$$Recall(x) = \frac{CorrectFound(x)}{TotalCorrect(x)} \tag{6.2}$$

$$Fscore(x) = \frac{2 * Recall(x) * Precision(x)}{Recall(x) + Precision(x)} \tag{6.3}$$

The x denotes equivalences, additions/deletions, or complex changes. Besides additions and deletions, we have not evaluated other simple changes because these require no elaborated adaptation transformations. We have identified the correct equivalences and changes in two ways. In the Petri Net example, we manually discovered the changes. In the Java example, we relied on the changes logged in the Netbeans repository. We also considered other manually discovered changes. We remarked that some repository logs do not report all the performed changes.

Besides matching accuracy, we have measured the matching process performance. This has been executed on a machine with Intel Core 2 Duo (2.4 GHz) and 1GB RAM.

6.1.4.3 Results

Matching accuracy Fig. 6.5 gives the prototype accuracy. The histograms display measures (precision, recall, fcore) for each selected couple of version. The three bars (from left to right) show the accuracy of equivalences, additions/deletions, and complex changes. Some bars are missing because certain couples of versions contain no deletions/additions or complex changes.

The results show that the prototype achieves a high accuracy not only in detecting the correct equivalences, additions/deletions, but also in detecting complex changes. Taking fscore as an example, the percentage of correct equivalences, and additions and deletions ranges from 99%-100%, and 90%-100%, respectively. Averanging accross all experiments, the fscore of complex changes is 100%. In particular, our prototype fails in identifying additions/deletions instead of equivalences (1% of cases).

Performance In the Petri Net example, the matching process consumes less than 1 second. In the Java example, the matching process approximately takes 10 seconds. A table containing the execution times of the heuristics in detail can be find in [27]. Even if the matching step consumes a relevant amount of resources, we should remember that this process generates an adaptation transformation that can be used several times.

6.1.4.4 EMF Compare versus our approach

We have compared the metamodel changes computed by EMF Compare to our results. We chose EMF Compare because this is a completely available prototype. Table 6.2 shows the fscore that EMF Compare and our approach, denoted by i. and ii., deliver on the

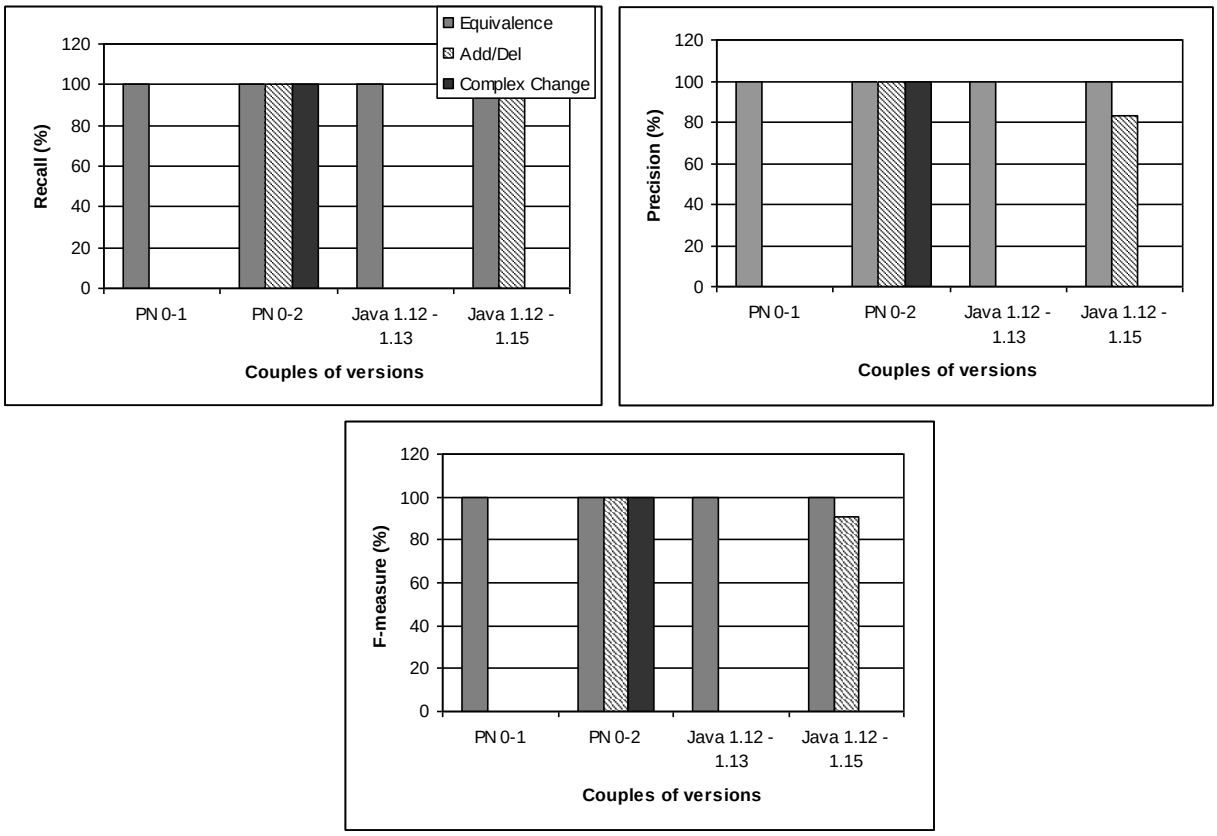


Figure 6.5: Matching accuracy results

Table 6.2: Fscore EMF Compare (i.) - Our approach (ii.)

Example	PetriNet		Java	
Couples of versions	0-2		1.12-1.15	
Approach	i.	ii.	i.	ii.
Additions-Deletions	0.8	1	1	0.9
Complex changes	0	1	0	0

Petri Net (couple 0-2) and Java (couple 1.12 - 1.15) examples. While EMF Compare is fairly good for identifying additions and deletions, this fails in rendering them as isolated actions. Because model adaptation automation needs to distinguish complex changes (i.e., not only additions and deletions), our approach is more appropriate for this purpose than EMF Compare.

6.1.5 Related work

We may divide the related approaches according to which of the two main issues they deal with: 1) discovery of equivalence and changes, or 2) derivation of adaptation transformations.

Chapter 3 has described most of approaches closer to the first problem. We now mention some of them. In the context of relational and object-oriented data bases, for example, the production of equivalences between two schemas/ontologies has been invested in [14][62]. In the MDE domain, the approaches of [106][107][108][109] present algorithms for detecting changes between UML models. Sriplakich et al. [110] identify simple changes in terminal models conforming to any metamodel. Wenzel et al. [111] present an approach which discovers fine-grained traces between versions of modeling languages, e.g., UML models, schemas, Web service description languages, and domain specific languages. The EMF Compare tool [112] reports changes between terminal model pairs or metamodel pairs. Finally, Falleri et al. [3] automatically detect equivalences between two metamodels using the algorithm *Similarity Flooding* described in [77].

In contrast to the first issue, the second one has been addressed by some recent approaches. The works described in [105][113][114][115][116] assume traces of changes are available, and derive adaptation transformations from them. In particular, [105], [116], and [115] apply stepwise automatic transactions on *MM1* to obtain *MM2*. These approaches then reuse the logs of applied transactions to derive adaptation transformations. Cicchetti et al. [114] use difference models provided by external tools.

Other model adaptation approaches are [10][117]. [10] needs a mapping model and hand-written Java code. [117], in turn, automatically copies from original to adapted model all model elements not impacted by metamodel evolution, and the user has to specify migration for the remaining elements.

The following items position our approach in comparison with the solutions mentioned above:

1. Similarly to [111][116], our approach computes equivalences and differences between any pair of metamodels (e.g., representing schemas, UML models, ontologies, grammars) .

2. Our solution overlaps the solutions presented in [105][114][115][117] in the sense of considering both simple and complex changes.

3. As in [3], in our matching process the robust algorithm *Similarity Flooding* can be executed. Falleri's main contribution is to provide 6 graph representation configurations. These configurations generate graphs that contain some metamodel information or all the metamodel information. Although light metamodel representations benefit the algorithm performance, they point out the matching process accuracy increases when all the metamodel information is represented in the graphs. This is what our approach exactly does.

4. Unlike existing approaches [105][113][114][115][116], we do not suppose that the changes are already known. We consider a more general case where the evolution of metamodels is done without someone explicitly keeping track of the applied changes.

5. Compared to [10][117][115], our approach automatically generates adaptation strategies with the least guidance from the user.

6. An experimentation shows that our approach scales to larger metamodels and models. This is an improvement on other techniques developed to date.

6.2 Pivot metamodels in the context of tool interoperability

6.2.1 Problem

The inability to provide a seamless interchange between tools can be found in many and various domains [118]. In software development, for instance, geographically distributed teams working on the same product often use several tools to trace bugs (e.g., Bugzilla, Mantis, Excel). In addition to integrate the developed modules, the teams have to centralize all the bugs logged on the diverse tools. How to be able to interoperate tools with different standards and formats?

Throughout time, varied approaches have been adopted to answer this question. Sun et al. [118] mention three solutions: XML-based exchange, parsing with GPLs, and model transformations. The authors explore model transformation as a solution. Their method basically captures the different formats as metamodels: abstract definitions of data structure and the mappings between them as transformation rules. Among the paper conclusions, we find that model transformation allows the separation of concerns (concrete and abstract syntaxes, and the mappings in between). Moreover, the authors point out that pivot (intermediate) metamodels often optimize the exchange among different tools.

The work described in [119] upholds the conclusions stated in [118]. Bezivin et al. [119] proposes an MDE solution to tool interoperability. The solution defines a metamodel for each tool, along with a pivot metamodel. The pivot metamodel abstracts a certain number of general concepts about a domain. A model transformation is defined among the pivot metamodel and every tool metamodel.

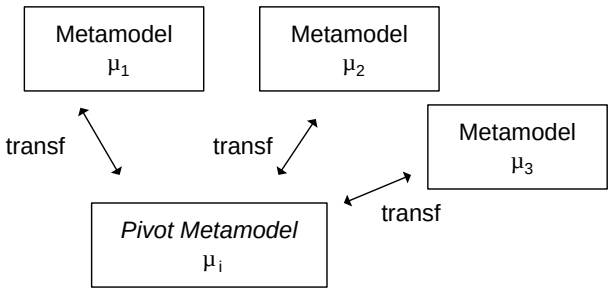


Figure 6.6: Transformations between a pivot metamodel and other metamodels

Just as [119][118] outline the advantages of model transformations in the context of tool interoperability, they argue the need for implementing a transformation for each new tool to interoperate with. We believe that the selection of pivot metamodel impacts the effort and cost for developing such transformations.

Software engineers need to be supported at building/selecting/evaluating pivot metamodels. Fig. 6.6 illustrates the generic problematic. Let μ_1 ... μ_n be a set of metamodels representing the same domain. We need a μ_i for playing the pivot role. Two instances of the generic problematic follow. The first instance illustrates an early development stage. Software engineers have to build or select μ_i from μ_1 ... μ_n so that they begin the transformation development. The second instance, in turn, describes a forward stage, where the engineers find transformations already developed, and they want to know whether μ_i was correctly chosen. This use case describes an M2-to-M2 matching-based approach that assists software engineers in the second problem: *pivot metamodel evaluation*. The first problem is out of its scope.

6.2.2 Running examples

We validate our ideas using three examples about tool interoperability. Each running example involves three metamodels representing tools and transformations between the metamodels. These examples have been contributed by the m2m Eclipse community [95].

- **Program Building** bridges Make, Ant, and Maven tools which automate software build processes [120].

- **Discrete Event Modeling** describes how bridges between Grafcet, Petrinet, and PNML have been built. Grafcet is a French representation support for discrete systems. Petrinet is a graphical and mathematical representation of discrete distributed systems. PNML (Petrinet Markup Language) is a XML-based interchange format for Petrinet.

- **Bug Tracing** implements bridges between different bug tracking tools like Bugzilla and Mantis. A third metamodel named Software Quality Control (denoted as SQC) has been defined.

Table 6.3 shows the size of each metamodel.

Table 6.3: Size of metamodels illustrating the pivot metamodel use case

Metamodel	**Size**
Ant	169
Make	22
Maven	186
Petrinet	24
Grafcet	27
PNML	31
SQC	33
Bugzilla	56
Mantis	50

6.2.3 Solution involving matching

Let us describe our pivot evaluation solution using a simplification of the generic problematic. The plain scenario consists of evaluating what is the best pivot of a set of three metamodels μ_1, μ_2, μ_3[3]. A procedure to do that follows:

1. We form three sequences from the set of metamodels. Each sequence has three metamodels: $\{\mu_1, \mu_2, \mu_3\}$, $\{\mu_2, \mu_1, \mu_3\}$, and $\{\mu_1, \mu_3, \mu_2\}$. The main features of these sequences are:

   - The second metamodel plays the pivot role. For example, μ_2 is the candidate pivot of the sequence $\{\mu_1, \mu_2, \mu_3\}$.
   - The first and third metamodel are commutative, i.e., $\{\mu_1, \mu_2, \mu_3\}$ is equal to $\{\mu_3, \mu_2, \mu_1\}$.

2. For each sequence, we match the metamodels with regarding the order. Taking $\{\mu_1, \mu_2, \mu_3\}$ as example, we match (μ_1, μ_2), (μ_2, μ_3), and (μ_1, μ_3). We consider μ_2 a good pivot metamodel, if the result of stepwise matchings, i.e., $map(\mu_1, \mu_2)$ and $map(\mu_2, \mu_3)$, is better that the result of a direct matching, i.e., $map(\mu_1, \mu_3)$. (see Fig. 6.7).

A simple method to determinate mapping quality is to compute the ratio of the concepts having mappings in $map(\mu_1, \mu_3)$. We refer to this computation using the abbreviation MCR (Mapped Concepts Ratio). For the sequence $\{\mu_1, \mu_2, \mu_3\}$, we would compute $MCR(map(\mu_1, \mu_3))$ and $MCR(map'(\mu_1, \mu_3))$, where $map'(\mu_1, \mu_3)$ is derived from $map(\mu_1, \mu_2)$ and $map(\mu_2, \mu_3)$ by applying transitivity. As in [62], we assume the transitivity of mappings.

At a first glance, the result $MCR(map(\mu_1, \mu_3)) > MCR(map'(\mu_1, \mu_3))$ would indicate that the direct matching is the best. The $map(\mu_1, \mu_3)$ may nonetheless contain many incorrect mappings and only few correct ones. The $map'(\mu_1, \mu_3)$ may in turn contain few mappings but all of them are correct.

[3]The main reason behind this choice is that available experimental data usually involves three metamodels. We believe that the approach remains applicable to a set of n metamodels.

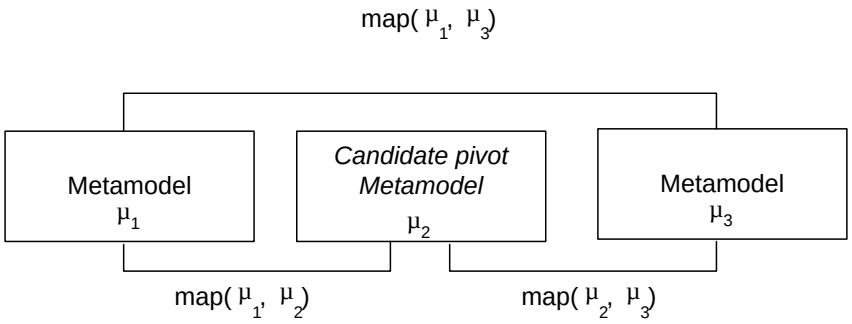


Figure 6.7: Direct matching versus stepwise matching

Because MCR is not a reliable metric, our approach uses the fscore measure. Below we list the approach steps in terms of the sequence $\{\mu_1, \mu_2, \mu_3\}$ (see Fig. 6.8). The same process has to be applied to the other sequences.

1. $map_r(\mu_1, \mu_3)$ is automatically extracted from a transformation $\mu_1 2 \mu_3$. μ_1 and μ_3 are the source and target metamodels of such a transformation. If the transformation $\mu_1 2 \mu_3$ is not available, our approach can extract the reference mapping model from the transformations $\mu_1 2 \mu_2$ and $\mu_2 2 \mu_3$ by applying the transitive and commutative properties.

2. A set of AML algorithms $s_1 ... s_n$ matches the pairs of metamodels (μ_1, μ_2), (μ_2, μ_3), and (μ_1, μ_3). Each algorithm generates $map_{s_i}(\mu_1, \mu_2)$, $map_{s_i}(\mu_2, \mu_3)$, and $map_{s_i}(\mu_1, \mu_3)$.

3. A program derives $map'_{s_i}(\mu_1, \mu_3)$ from $map_{s_i}(\mu_1, \mu_2)$ and $map_{s_i}(\mu_2, \mu_3)$ by applying transitivity and commutativity.

4. We calculate $Fs_i(\mu_1, \mu_3)$ and $F's_i(\mu_1, \mu_3)$.

5. Having the results over $\{\mu_1, \mu_2, \mu_3\}$ and the other sequences, we select the algorithm giving the best fscores for all the sequences, e.g., s_i.

6. Looking at the fscores of s_i, we choose μ_2 as a good pivot if $Fs_i(\mu_1, \mu_3) < F's_i(\mu_1, \mu_3)$.

Most of the steps above have been described in previous sections. For example, Section 5.2 describes the extraction approach used in step 1. Here we elaborate on the new introduced concepts: the transitivity and commutativity of mappings (involved in step 1) and the set of algorithms experimented by our approach (step 2).

6.2.3.1 Transitivity and commutativity of mappings

Let us introduce two base notions: $C(\mu, KM3, Concepts)$ denotes the concepts defined by a metamodel μ conforming to KM3, where $Concepts$ is the OCL query:

```
ModelElement.allInstances()->
select(e | e.oclIsTypeOf(Class) or e.oclIsKindOf(StructuralFeature))
```

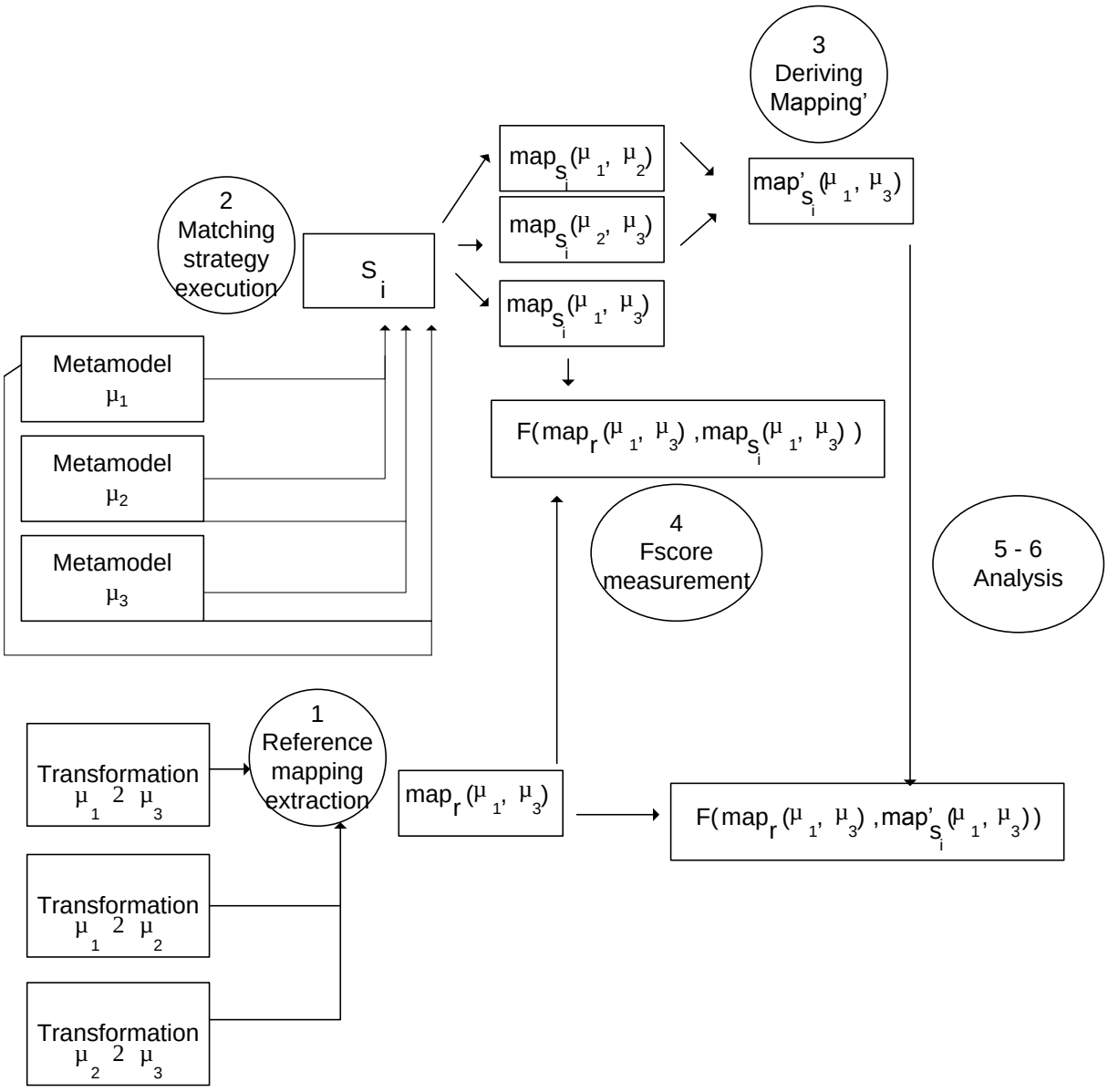


Figure 6.8: Approach for evaluating pivot metamodels

We restrict $C(\mu, KM3, Concepts)$ to classes and properties (i.e., `StructuralFeatures`) because transformations are basically written in terms of them [45].

(e, f) is a mapping of e and f, where $e \in C(\mu_1, KM3, Concepts)$ and $f \in C(\mu_2, KM3, Concepts)$. A simple mapping is an equality relationship: e and f have the same intended meaning [70].

As in [62], we assume the transitivity of mappings. Let $g \in C(\mu_3, KM3, Concepts)$, if (e, f) and (f, g), then also (e, g). Since mappings represent equivalence relations, we also suppose that mappings are commutative, i.e., $(e, f) = (f, e)$.

An operator (denoted as P) can derive mappings from $map(\mu_1, \mu_2)$ and $map(\mu_3, \mu_4)$ recalling to transitivity and commutativity of mappings as follows:

- if $\mu_1 = \mu_3$, then $P(map(\mu_1, \mu_2), map(\mu_3, \mu_4)) = map(\mu_2, \mu_4)$
- if $\mu_2 = \mu_4$, then $P(map(\mu_1, \mu_2), map(\mu_3, \mu_4)) = map(\mu_1, \mu_3)$
- if $\mu_1 = \mu_4$, then $P(map(\mu_1, \mu_2), map(\mu_3, \mu_4)) = map(\mu_2, \mu_3)$
- if $\mu_2 = \mu_3$, then $P(map(\mu_1, \mu_2), map(\mu_3, \mu_4)) = map(\mu_1, \mu_4)$

6.2.3.2 AML M2-to-M2 matching algorithms

Our approach has experimented 4 AML algorithms, i.e., `Levenshtein_ThresholdMaxSim`, `Levenshtein_BothMaxSim`, `MSR_ThresholdMaxSim`, and `MSR_BothMaxSim`. The two former have been presented in Section 5.5.2.1, we elaborate on the two latter here. What makes different `MSR_ThresholdMaxSim` and `MSR_BothMaxSim` from the other algorithms is:

- With respect to the included linguistic-based similarity heuristic: the `MSR` heuristic Section 4.4.2 instead of the `Levenshtein` heuristic.
- The use of three creation heuristics: *TypeClass*, *TypeReference*, and *TypeAttribute*, instead of `CreationByFullNameAndType` and `CreationAddedDeleted`.
- The chosen selection method, i.e., `BothMaxSim` or `ThresholdMaxSim`.

As indicated in Section 4.4.2, we exploit MSR technologies in two steps. An AML heuristic (named `RequestMSR`) recovers similarity scores from the MSR Web server first, and then another heuristic (called `MSR`) matches the metamodels by using the server results. Listing. 6.5 and Listing. 6.7 illustrate how these heuristics interact with others to leverage each step.

Listing. 6.5 shows the invocation of creation heuristics (lines 3-5) and `RequestMSR` three times. Each invocation corresponds to a KM3 metametamodel type (`Class`, `Attribute`, and `Reference`). This separation allows us to restrict the number the mappings between the metamodel concepts which in turn improve the MSR heuristics performance.

Listing 6.5: RequestMSR modelsFlow

```
1 modelsFlow {
2
3  typeClass = TypeClass[map]
4  typeRef = TypeReference[map]
```

```
5  typeAtt = TypeAttribute[map]
6  rClass = RequestMSR[typeClass](msrParamClass)
7  rRef = RequestMSR[typeRef](msrParamStrF)
8  rAtt = RequestMSR[typeAtt](msrParamStrF)
9
10 }
```

`RequestMSR` takes as input two models (`msrParamClass`, `msrParamStrF`) conforming to the parameter metamodel described in Section 4.2.2. The models indicate tokenizers and distractors normalizing class or property names. We have built 2 parameter models for each running example. For instance, Listing. 6.6 shows the `msrParamClass` model for the Bugzilla - Mantis example. Because the class names look like, for example `BugzillaRoot` and `IdentifiedElt`, `msrParamClass` specifies the distractors `root` and `elt`, and selects the `UpperCaseTokenizer`.

Listing 6.6: Parameter model for the Bugzilla - Mantis running example

```
1 <ParameterList xmlns="ParameterMM">
2   <parameters xsi:type="StringParameter" name="MSR" value="NSS-Ggoogle"/>
3   <parameters xsi:type="StringParameter" name="leftDistractor" value="root"/>
4   <parameters xsi:type="StringParameter" name="rightDistractor" value="elt"/>
5   <parameters xsi:type="StringParameter" name="rightTokenizer" value="UpperCaseTokenizer"
        ↪/>
6   <parameters xsi:type="StringParameter" name="leftTokenizer" value="UpperScoreTokenizer"
        ↪/>
7 </ParameterList>
```

Listing. 6.7 illustrates the `MSR` heuristic invocation. It uses the MSR server results which are stored in spreadsheet files: `spreadsheetMSRClass`, `spreadsheetMSRRef`, and `spreadsheetMSRAtt`. Afterward, the algorithm merges and filters correspondences, and propagates similarity (lines 6-8).

Listing 6.7: MSRBothMaxSim model flow

```
1  modelsFlow {
2
3  msrClass = MSR[typeClass](spreadsheetMSRClass, msrParamClass)
4  msrRef = MSR[typeRef](spreadsheetMSRRef, msrParamStrF)
5  msrAtt = MSR[typeAtt](spreadsheetMSRAtt, msrParamStrF)
6  inSF = Merge[1.0:msrAtt, 1.0:msrRef, 1.0:msrClass]
7  outSF = SimilarityFlooding[]
8  both = BothMaxSim[outSF]
9
10 }
```

6.2.4 Experimentation

By following the step 1 of Section 6.1, we obtained the sequences shown in Table 6.4. We decided the positions of the first and third metamodels by taking into account the available transformations. Observe that some transformations are not available, i.e., $Bugzilla2Mantis$, $Make2Maven$, and $Grafcet2PNML$. We got the corresponding map_r

Table 6.4: Features of examples illustrating the pivot metamodel use case

Example	**Metamodels**	**Sequences**	**Transformations**
Program building	Ant, Make, Maven	{Make, Maven, Ant}, {Make, Ant, Maven}, {Ant, Make, Maven}	Make2Ant, Ant2Maven
Discrete event modeling	Petrinet, Grafcet, PNML	{Petrinet, Grafcet, PNML}, {Grafcet, Petrinet, PNML}, {Grafcet, PNML, Petrinet}	Grafcet2Petrinet, Petrinet2PNML
Bug tracing	Software Quality Control (SQC), Bugzilla, Mantis	{SQC, Mantis, Bugzilla}, {Bugzilla, SQC, Mantis}, {SQC, Bugzilla, Mantis}	SQC2Bugzilla, SQC2Mantis

by applying the P operator.

Section 6.2.3.1 presents the results of applying our approach to the running examples and Section 6.2.3.1 discusses such results.

6.2.4.1 Results

Program Building Tools Fig. 6.9 shows the results of applying our approach to the metamodels representing program building tools. In general, the strategy `Levenshtein_BothMaxSim` renders the best fscores for the three sequences. The results suggest Ant as the best candidate pivot. Observe that the condition $Fs_i(\mu_1, \mu_3) < F's_i(\mu_1, \mu_3)$ is satisfied, $F(Make, Maven) < F'(Make, Maven)$, i.e., $0.3 < 0.4$. Make is, in turn, the worst pivot, $F(Ant, Maven) > F'(Ant, Maven)$, i.e., $0.6 > 0.2$. Maven does not bring further advantages since the fscores are equal, $F(Make, Ant) = F'(Make, Ant)$ and $0.2 = 0.2$.

The reason behind the selection of Ant may have a historical background. The history reports the emergence of Ant (2000) between Make (1977) and Maven (2002) [120]. We can imagine that Make inspired the creators of Ant, and that Ant inspired the creators of Maven. That would explain why Ant is a good pivot between Make and Maven.

It is possible getting more than one strategy that gives good fscores. For example, `Levenshtein_BothMaxSim` and `MSR_BothMaxSim` give valid results at matching the sequence $\{Ant, Make, Maven\}$. We however chose `Levenshtein_BothMaxSim` because it gives good results at matching the other sequences too.

Discrete Event Modeling Tools As in the previous example, the strategy `Levenshtein_BothMaxSim` gives the best fscores for the discrete event modeling example (see Fig. 6.10). The results indicate that there is not metamodels playing a good pivot role. Petrinet is better than Grafcet and PNML. However, we can not spell out Petrinet as a good pivot since $F(Grafcet, PNML)$ is equal to $F'(Grafcet, PNML)$, i.e., $0.3 = 0.3$. These fscore values may point out the need for improvements over Petrinet to make a better pivot.

Bug Tracing Tools Fig. 6.11 depicts the results for the bug tracing metamodels. Unlike the previous cases, the strategy giving the best fscores is `MSR_BothMaxSim`. Looking at these results shows that SQC is not a good pivot. The reason is that the condition $Fs_i(\mu_1, \mu_3) < F's_i(\mu_1, \mu_3)$ is not satisfied, i.e., $F(Bugzilla, Mantis) = 0.4$ and $F'(Bugzilla, Mantis) = 0.2$. Mantis is, in order, better pivot than Bugzilla.

This example has a special feature. All the running examples metamodels, except SQC, were created from tool specifications. Since there was not specification for SQC, the software engineer may have created SQC inspired by Mantis. This would explain why SQC is closer to Mantis than to Bugzilla and why Mantis is the best pivot.

6.2.4.2 Discussion

We use the fscore measure for pivot metamodel evaluation based on two premises: 1) the suitability of an AML algorithm for matching a given pair of metamodels, and 2) the quality of reference mapping models. In practice, these assumptions might be not

[Candidate pivot: Maven]

Strategy (s)	F(map_r(Make,Ant), map_s(Make,Ant))	F(map_r(Make,Ant), map'_s(Make,Ant))
MSR_BothMaxSim	0.2	0.1
Levenshtein_BothMaxSim	0.2	0.2
Levenshtein_ThresholdMaxDelta	0.1	0.0
MSR_ThresholdMaxDelta	0.1	0.0

[Candidate pivot: Ant]

Strategy (s)	F(map_r(Make,Maven), map_s(Make,Maven))	F(map_r(Make,Maven), map'_s(Make,Maven))
MSR_BothMaxSim	0.1	0.3
Levenshtein_BothMaxSim	0.3	0.4
Levenshtein_ThresholdMaxDelta	0.1	0.0
MSR_ThresholdMaxDelta	0.2	0.0

[Candidate pivot: Make]

Strategy (s)	F(map_r(Ant,Maven), map_s(Ant,Maven))	F(map_r(Ant,Maven), map'_s(Ant,Maven))
MSR_BothMaxSim	0.6	0.2
Levenshtein_BothMaxSim	0.6	0.1
Levenshtein_ThresholdMaxDelta	0.2	0.1
MSR_ThresholdMaxDelta	0.1	0.0

Figure 6.9: Fscore results: Program building example

[Candidate pivot: Grafcet]

Strategy (s)	F(map_r(PetriNet,PNML), map_s(PetriNet,PNML))	F(map_r(PetriNet,PNML), map'_s(PetriNet,PNML))
MSR_BothMaxSim	0.4	0.3
Levenshtein_BothMaxSim	0.5	0.3
Levenshtein_ThresholdMaxDelta	0.3	0.0
MSR_ThresholdMaxDelta	0.2	0.0

[Candidate pivot: Petrinet]

Strategy (s)	F(map_r(Grafcet,PNML), map_s(Grafcet,PNML))	F(map_r(Grafcet,PNML), map'_s(Grafcet,PNML))
MSR_BothMaxSim	0.2	0.3
Levenshtein_BothMaxSim	0.3	0.3
Levenshtein_ThresholdMaxDelta	0.2	0.0
MSR_ThresholdMaxDelta	0.2	0.0

[Candidate pivot: PNML]

Strategy (s)	F(map_r(Grafcet,PetriNet), map_s(Grafcet,PetriNet))	F(map_r(Grafcet,PetriNet), map'_s(Grafcet,PetriNet))
MSR_BothMaxSim	0.5	0.5
Levenshtein_BothMaxSim	0.6	0.2
Levenshtein_ThresholdMaxDelta	0.3	0.1
MSR_ThresholdMaxDelta	0.0	0.1

Figure 6.10: Fscore results: Discrete event modeling example

[Candidate pivot: Mantis]

Strategy (s)	F(map_r(SQC,Bugzilla), map_s(SQC,Bugzilla))	F(map_r(SQC,Bugzilla), map'_s(SQC,Bugzilla))
MSR_BothMaxSim	0.3	0.4
Levenshtein_BothMaxSim	0.4	0.0
Levenshtein_ThresholdMaxDelta	0.2	0.0
MSR_ThresholdMaxDelta	0.2	0.2

[Candidate pivot: SQC]

Strategy (s)	F(map_r(Bugzilla,Mantis), map_s(Bugzilla,Mantis))	F(map_r(Bugzilla,Mantis), map'_s(Bugzilla,Mantis))
MSR_BothMaxSim	0.4	0.2
Levenshtein_BothMaxSim	0.3	0.0
Levenshtein_ThresholdMaxDelta	0.3	0.0
MSR_ThresholdMaxDelta	0.2	0.2

[Candidate pivot: Bugzilla]

Strategy (s)	F(map_r(SQC,Mantis), map_s(SQC,Mantis))	F(map_r(SQC,Mantis), map'_s(SQC,Mantis))
MSR_BothMaxSim	0.3	0.3
Levenshtein_BothMaxSim	0.2	0.2
Levenshtein_ThresholdMaxDelta	0.2	0.0
MSR_ThresholdMaxDelta	0.2	0.2

Figure 6.11: Fscore results: Bug tracing example

satisfied. Firstly, the space of matching strategies is quite large, and sometimes it is difficult to find a strategy that accurately matches a pair of metamodels. Secondly, our approach extracts reference mapping models from transformations. When transformations are complex or incomplete, the approach extracts only a few correct mappings. Because the fscore measure depends on strategy and transformation, we might get low fscore values, and be unable to select the correct pivot.

The current experimentation reports low fscore values and low deltas (around 0.1) between $Fs_i(\mu_1, \mu_3)$ and $F's_i(\mu_1, \mu_3)$. A reason is the quality of the reference mapping models. For example, $Map_r(Bugzilla, Mantis)$ only contains 8 reference correspondences, this might be surprised having metamodels with 50 concepts (in average). Because the transformation $Bugzilla2Mantis$ was not available, we derived $map_r(Bugzilla, Mantis)$ from $map_r(SQC, Mantis)$ and $map_r(SQC, Bugzilla)$. Thus, the reference mapping model $map_r(Bugzilla, Mantis)$ lacks items due to the transformations $SQC2Mantis$ and $SQC2Bugzilla$ contain a few rules from which we can derive mappings by applying transitivity and commutativity. In practice, developers write a given transformation in terms of the data instances that they expect to have in the source and target models. If the data instances are not relevant, the transformation may lack rules linking certain metamodel concepts.

The uncertainty of satisfying the two premises mentioned above might question the validity of our approach. However, the experimental results indicate its applicability to pivot metamodel evaluation. For two examples, i.e., program building and diagram event modeling, our solution overlaps the pivots chosen by the running examples contributors, i.e., Ant and Petrinet. With respect to the bug tracing example, our approach suggests a pivot metamodel (Mantis) different from the contributor choice (SQC). Remark on SQC was built for the example purpose, this could explain the lack of bridge concepts needed in a pivot. In addition, our approach allows us to:

1. Figure out when a given metamodel is closer to certain metamodels than to others, e.g., SQC is closer to Mantis than to Bugzilla.

2. Identify when a metamodel has to be further refined/enriched to improve its pivot role, e.g., Petrinet in the discrete event modeling example.

3. Isolate metamodels that have to be not chosen as pivots, e.g., Make in the program building example.

4. The strategy, providing the more reliable fscores, indicates what is the predominant kind of similarity existing between the matching metamodels. For example, whereas `Levenshtein_BothMaxSim` suggests a syntactic similarity between Ant and Maven, `MSR_BothMaxSim` suggests a semantic similarity between Bugzilla and Mantis.

As shown in Fig. 6.9, it is probable to have several strategies rendering fuzzy fscores. When this happens, the complexity of evaluating pivots increases. A solution may be to benchmark more than the 4 strategies experimented here. In the near future, we wish to apply our approach to a large set of examples and matching strategies. It would be ideal to have transformations developed in both directions.

Although our approach focuses on accuracy, we mention its performance. It depends on metamodels size, and matching strategy performance. For example, the approach takes

2 minutes to discover that Mantis is the best candidate pivot for the bug tracing example. For each sequence, the approach takes 40 seconds: the `Levenshtein`-based strategies take around 5 seconds, the `MSR`-based strategies approximately takes 15 seconds. The latter time does not include the MSR Web server time response, which fluctuated between 1 and 3 hours. We believe that time spent in evaluation is not spoiled because the evaluation may suggest a pivot metamodel facilitating the development of further transformations.

As stated in Section 6.1, we concentrate on the second instance of the generic problematic, the one where transformations have been already implemented. Our approach indicates the best candidate pivots, but software engineers take the last decision. They should judge whether implementing new transformations (between the suggested pivot and the other metamodels) is cheaper than keeping the existent ones. We expect to extent our approach for addressing the first instance of the generic problematic: *pivot metamodel selection*.

6.2.5 Related Work

We mention some works that measure overlapping degree. The works have been proposed in two different disciplines: ontology development and MDE.

[121] presents a survey of approaches that evaluate ontology similarity. The authors point out four categories of evaluation: 1) compare the ontology to a gold standard, 2) use the ontology and evaluate its results, 3) involve comparisons with data instances, and 4) include human assistance. Furthermore, they propose an instance of the first category. The work puts forward a similarity measure for ontologies based on clustering notions. The idea is to partition the ontologies (a given ontology and a gold standard) into disjoint subsets. They apply similarity measures at subset level, and sum the partial values. The aggregation result represents a consolidated similarity value between the ontologies. The approach supposes the existence of data instances, however it is not always the case.

In MDE, [122] measures lexical similarity between model elements by using WordNet [75]. This approach counts the number of mappings, and then calculates the percentage of mapped model elements. On the other hand, [123] generates different candidate metamodels from a knowledge base and chooses the best one. The authors propose a similarity measure to make the choice. They basically counts the number of model elements having good syntactical proximity. The disadvantage in *count-based* approaches is that the counted mappings can include false items, which disrupt the similarity measure. The advantage is that the approach is independent of gold standards. In contrast to count-based approach, we rely on the fscore measure that gives a ratio of correct mappings but we need gold standards (that may be not available).

6.3 Model synchronization

6.3.1 Problem

An interesting challenge in Model Driven Software Product Lines (MD-SPL) is to maintain code artifacts and models synchronized. An MD-SPL uses as main assets metamodels, models, and transformations to generate concrete products in the line [31]. Metamodels represent diverse views of the product line, e.g., business logic, architecture, platform, and

programming language. Models describe specific details of a product. A transformation chain closes the gap between models and code.

Implementing transformations that generate 100% of the source code for an application is very difficult. That is why developers often modify the generated source code by hand. As a result, models and code become incoherent: models represent the state of the system during the design phase, but the source code represents the current implementation state.

6.3.2 Solution involving matching

Meneses et al. [104] proposes a (semi)automatic approach to update models as source code changes. The synchronization process includes the following steps:

1. Obtain an AST (Abstract Syntax Trees) model from the manually modified Java source code. The authors use Modisco project tools[4] to leverage this step.

2. Compare two versions of the AST model: the model generated by the transformation chain (named *MV1*) to the model obtained in the previous step (called *MV2*). Find what elements of *MV1* change in *MV2*. Here the authors use an AML M1-to-M1 matching algorithm that gives as a result a diff model. Section 6.3 presents the algorithm in detail.

3. Identify how the changing *MV1* elements are related to elements of other models yielded by the transformation chain. To do that, the authors use the approach described in [124]. The approach automatically generates traceability models every time a transformation chain is executed.

4. Build a reconciliation model from the output models of previous steps: AST models, diff model, and traces.

5. Update a given business model using the reconciliation model. ATL transformations perform the 4th and 5th step.

The remaining sections focus on the 2nd step of Meneses' approach which refers to the AML M1-to-M1 matching algorithm.

6.3.3 Running example

As indicated so far, Meneses' approach needs to find the changes that an AST Model *V2* introduces into *V1*. Meneses wants to track certain kinds of changes, i.e., addition, elimination, or renaming of attributes, methods, or classes. To identify such changes, it is necessary to judge attributes, methods, or classes as follows:

- Since a single Java class can not have two attributes with the same name, one discovers attribute changes by comparing names.

- To detect method changes, one trusts on the method names, return types, and parameters.

[4] http://www.eclipse.org/gmt/modisco/

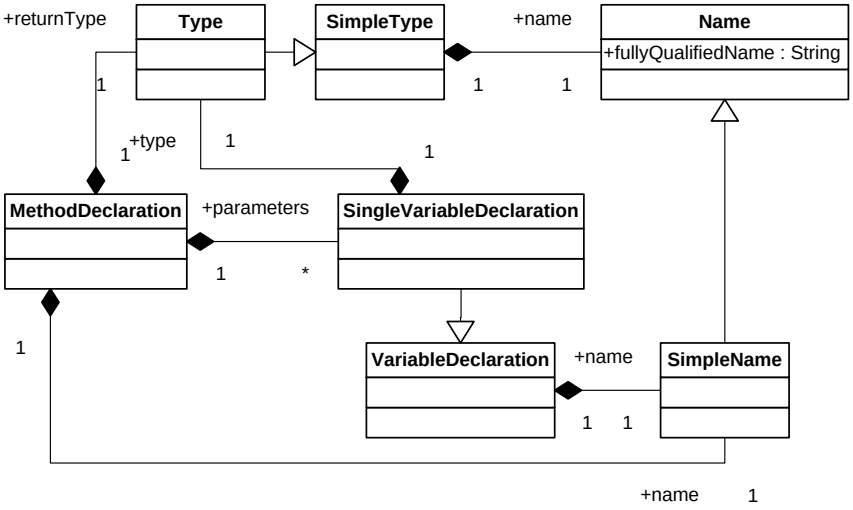


Figure 6.12: Excerpt of the AST Java metamodel

- Class changes are identified by regarding class names as well as the contained attributes and methods.

Let us take a look to the AST metamodel to get a concrete idea about the matching algorithm implementation Fig. 6.12. We concentrate on the method changes. To match `MethodDeclaration`, it is necessary to compare the `name`, `returnType`, and `parameters` properties. Since the properties are not simple attributes but references, it is needed to match the referred elements too. Thus, the algorithm has to include matching rules for `SimpleName`, `Type`, and `SingleVariableDeclaration`. `Type` requires a comparison of the `name` property, and `SingleVariableDeclaration` a comparison of the `name` and `type` properties. An additional characteristic of the AST metamodel is the presence of the `fullyQualifiedName` property instead of the classical `name` property. This invalidates the use of the default EMF Compare algorithm which relies on the `name` property to match models. In addition, this confirms the need of a customized model matching algorithm like the one we present below.

6.3.3.1 AML M1-to-M1 matching algorithm

Listing. 6.8 indicates the heuristics delivering `MethodDeclaration` changes (find the full algorithm in Appendix C):

- `CMethodD` creates links between `MethodDeclaration` elements.
- `WeigthedSum` and `Threshold` have the same functionality explained in previous sections.
- `SMethodDName`, `SMethodDReturnT`, and `SMethodDParameters` compare the `name`, `returnType`, and `parameters` properties. Note the `sim` helper, lines 37, 41, and 45. For example, given a `name` element of *MV1* and a `name` element of *MV2*, the `sim` helper looks for a correspondence linking such elements in the `sSN` model. The `SSimpleName` heuristic yields `sSN`, and reuses the `simStrings` helper in the same fashion

as `Levenshtein` heuristic does. The `averSimSets` helper has a behavior similar to `sim` but for collections of elements. Firstly, `averSimSets` seeks for correspondences linking elements of a collection of *MV1* and *MV2* (e.g., `parameters`). Then, `averSimSets` computes an average between the number of correspondences successfully found in the input mapping model and the total number of correspondences existing between the collections elements. Listing. 6.9 shows the heuristics computing the input mapping model of `SMethodDParameters`, i.e., `tSVD`. This block involves the `WeightedSum` and `Threshold` heuristics too. In addition, we want to focus on the `CSimpleName` heuristic which creates input links for `SSimpleName`. Since other fragments of the AST metamodel involve the `SimpleName` class, the `CSimpleName` condition (lines 9-13) has to restraint the creation of links between the `SimpleName` elements associated to the `MethodDeclaration` class: `SingleVariableDeclaration` and `MethodDeclaration`.

- `JavaASTDifferentiation` is an external ATL transformation which marks with `Added` and `Deleted` the `MethodDeclaration` elements not having correspondences in the input mapping model.

Listing 6.8: Excerpt of the AML algorithm matching AST Java models

```
1  strategy JDTAST {
2
3  uses JavaASTDifferentiation[IN1:EqualModel(m1:JavaAST, m2:JavaAST)]()
4
5  create CSimpleName () {
6   leftType : SimpleName
7   rightType : SimpleName
8   when
9   thisLeft.refImmediateComposite().oclIsKindOf(JavaAST!SingleVariableDeclaration) and
10  thisRight.refImmediateComposite().oclIsKindOf(JavaAST!SingleVariableDeclaration)
11  or
12  thisLeft.refImmediateComposite().oclIsKindOf(JavaAST!MethodDeclaration) and
13  thisRight.refImmediateComposite().oclIsKindOf(JavaAST!MethodDeclaration)
14
15 }
16
17 sim SSimpleName ()
18 ATLLibraries{
19  (name='Strings', path='../AMLLibrary/ATL/Helpers')
20 }
21 JavaLibraries{
22  (name='match.SimmetricsSimilarity', path='../AMLLibrary/Jars/simmetrics.jar')
23 }
24 {
25  is thisLeft.fullyQualifiedName.simStrings(thisRight.fullyQualifiedName)
26 }
27 ...
28
29 create CMethodD () {
30  leftType : MethodDeclaration
31  rightType : MethodDeclaration
32  when
33   true
34 }
35
36 sim SMethodDName (IN1:EqualModel (m1:JavaAST, m2:JavaAST)) {
37  is thisModule.sim(thisLeft.name, thisRight.name)
38 }
39
```

```
40 sim SMethodDReturnT (IN1:EqualModel (m1:JavaAST, m2:JavaAST)) {
41  is thisModule.sim(thisLeft.returnType, thisRight.returnType)
42 }
43
44 sim SMethodDParameters (IN1:EqualModel (m1:JavaAST, m2:JavaAST)) {
45  is thisModule.averSimSets(thisLeft.parameters, thisRight.parameters)
46 }
47
48 modelsFlow {
49
50  cSN = CSimpleName[map]
51  sSN = SSimpleName[cSN]
52
53  ...
54  cMD = CMethodD[map]
55  sMDN = SMethodDName[cMD](sSN)
56  sMDR = SMethodDReturnT[cMD](tT)
57  sMDP = SMethodDParameters[cMD](tSVD)
58
59  wMD = WeightedSum[0.4:sMDN, 0.3:sMDR, 0.3:sMDP]
60  tMD = Threshold[wMD]
61
62  d = JavaASTDifferentiation[tMD]
63
64 }
```

Listing 6.9: AML algorithm matching AST models, SingleVariableDeclaration excerpt

```
1
2  cSVD = CSingleVD[map]
3  sSVDN = SSingleVDName[cSVD](sSN)
4  sSVDT = SSingleVDType[cSVD](sT)
5  wSVD = WeightedSum[0.5:sSVDN, 0.5:sSVDT]
6  tSVD = Threshold[wSVD]
```

6.3.4 Experimentation

[104] does not report the experimentation dataset. To give an idea of the AML M1-to-M1 matching algorithm performance, we have applied it to a pair of AST models provided by Meneses. The AST model *V1* has 377 elements and *V2* has 426. The algorithm matched the models in 7 seconds and found the expected changes.

6.4 Summary

Here we summarize the findings of each use case.

Model co-evolution This use case presented how to use AML to leverage model adaptation. An AML algorithm computes equivalences and changes between two metamodels. A Higher-Order Transformation translates equivalences and changes into an executable adaptation transformation. We reported the accuracy of our algorithm which is pretty good; our algorithm always discovers the changes, and only fails by identifying simple changes when in truth there is an equivalence (in 1% of the cases).

Readers interested in the position of AML with respect to other model migration tools may want to take a look to [125], this paper presents the advantages and disadvantages of the tools in different situations. In particular, the paper hints AML as a suitable tool for reverse-engineering model migration (i.e., the case where a trace of changes does not exist).

AML has a good performance and minimizes hand-written code and guidance from user. On the other hand, the paper points out that some AML matching transformations have to be modified in order to support more complex changes. The first AML version tackles the taxonomy of changes proposed [105], except breaking and non-resolvable changes.

Pivot metamodels in the context of interoperability tool This work showed how to perform the evaluation of pivot metamodels by using AML algorithms and the assessment approach described in Chapter 5. We applied our approach to three running examples taken from the ATL Zoo [126]. The experimentation illustrates that the approach make it possible to figure out 1) what is the metamodel that maps concepts of a set of metamodels in the best way, 2) when a given metamodel is closer to certain metamodels than to others, 3) when metamodel has to be further refined/enriched to improve its pivot role, and 4) when a metamodel must not be chosen as a pivot.

Moreover, this use case compared the accuracy of the `MSR` matching transformation to the `Levenshtein` one. For 2 of the 3 examples, the `Levenshtein`-based algorithm was more accurate than the `MSR`-based one. Note that it is also the case for the test cases described in Section 5.5; there, `Levenshtein`-based algorithms reported better fscores than the `WordNet`-based ones. In contrast to `WordNet`, the `MSR` heuristic found more relateness between technical concepts. At the same time, the `MSR` heuristic opened new possible matches since a large corpora (such as Google) was exploited. Based on these results, we conclude that the use of a more technical dictionary may improve matching results over the modeling dataset.

Model synchronization This use case depicted that it is possible to use AML for M1-to-M1 matching. Again the matching is incremental, a given algorithm matches constituent/related elements first, and then the algorithm uses computed correspondences to decide if other principal elements are equivalent. AML M1-to-M1 matching algorithms reuses selection and aggregation heuristics, and similarity helpers typically included in M2-to-M2 matching algorithms.

Chapter 7

Conclusions

This chapter summarizes the thesis contributions and presents future work related to the core of the thesis (i.e., model matching calculation) and its associated use cases.

7.1 Contributions

7.1.1 Survey of model matching approaches

In contrast to ontology (or schema) matching approaches, which commenced to appear 30 years ago, (meta)model matching approaches have recently emerged. As a result, there exist poor surveys of them. We have contributed a broad survey of existing model matching approaches. We have adapted ontology matching survey criteria to MDE. We have used such criteria to study and compare existing approaches. The modeling community may use the adapted criteria to easily classify other emerging approaches and then maintain this survey up to date.

7.1.2 Matching heuristics independent of technical space and abstraction level

Looking at existing matching algorithms shows repetitive code. Even if these algorithms match pairs of (meta)models by taking into account standard features, different fragments of code have had to be developed to support (for example) either MOF or Ecore metamodels. This thesis investigated how to promote the reusability of matching heuristics among technical spaces and abstraction levels (i.e., metamodels and/or models) by using DSLs and modeling techniques.

Based on a domain analysis, we have proposed five kinds of matching heuristics. A matching algorithm is the combination of heuristics which are incarnations of such kinds. Alignments (which refer to *Left* and *Right* inputs) and a stepwise process allow the interaction among heuristics combined in an algorithm.

We have contributed a DSL (called AML) whose constructs overlap the abstractions mentioned in the previous paragraph. The constructs aims loosely coupling of matching heuristics to a given technical space or abstraction level, in addition, they factor code.

We have chosen modeling techniques to make AML programs executable. A compiler translates AML heuristics and matching process specification into a set of ATL transformations and an Ant transformation chain, correspondingly. (Meta)models and *equal* models represent inputs and alignments, respectively. An additional component can translate inputs into the internal AML format (i.e., a contemporary metamodeling format such as Ecore or KM3) if they differ from it. We have implemented the DSL on top of the AmmA suite.

By using AML, we have developed a library containing matching heuristics and algorithms. The heuristics exploit linguistic/structural information and sample instances. Some transformations use external resources such as dictionaries (e.g., WordNet) or online large corporas (e.g., Google). We have reused existing code to interface AML with these kinds of resources.

To validate that our M2-to-M2 matching algorithms go beyond the modeling technical spaces, we have applied them not only to pairs of metamodels but also to pairs of ontologies. Moreover, we have implemented an M1-to-M1 matching algorithm to show that some AML heuristics used in M2-to-M2 algorithms can be reused in M1-to-M1. Therefore, such matching heuristics are independent of abstraction level.

Our experimentations have demonstrated that one can use AML to build customizable matching algorithms. Each algorithm involves *generic* and *narrowed* matching heuristics. The former kind matches any pair of (meta)models and the latter is adapted to certain pairs in order to improve generic heuristics results. Thus, developers just need to focus on narrowed heuristics and on how to combine, both, generic and narrowed. If the developers reuse generic matching heuristics, then algorithm development time may be reduced.

We have contributed AML and its library to Eclipse. From there, users can download the tool for free, and post inquiries in a newsgroup. The decision on implementing a matching algorithm by using either a DSL (such as AML) or a GPL (such as Java) is not clear for all cases; it depends on the priorities one has. For example, if software project managers wish to reduce development effort to short-term, it may be easier to ask programmers to develop an algorithm with the GPL they use daily. In contrast, if their goals are: 1) to reduce development effort to medium(or long)-term (i.e., assuming costs of learning curve for a given DSL), and 2) to facilitate matching algorithm understanding to users not having a large programming background, DSLs such as AML appear to be a promising direction for committing such goals.

7.1.3 Modeling artifacts to automate matching algorithm evaluation

Evaluation allows the classification of algorithms in terms of strengths and weaknesses. From evaluation results, the user gets some guidelines about what algorithm to choose given a pair of (meta)models. Being aware of evaluation importance, we have made contributions in that sense.

Firstly, our approach addresses the lack of matching evaluation test cases. The approach extracts reference alignments from model transformations. Having a large set of pairs of (meta)models and reference alignments, we can perform more extensive benchmarks.

Secondly, the approach tackles the issue of low evaluation efficiency. The key is the generation of Ant scripts from a megamodel; the generated Ant script lists all the test cases and delegates to another Ant script the execution of actions over them. By modifying the latter Ant script, one can easily add, delete, or modify actions. For example, one can plot a special kind of curve from matching results. Note that this may considerably increase evaluation efficiency, above all, the phase of processing results.

To validate that our evaluation approach is also applicable to diverse technical spaces, we have tested the quality of our algorithms over ontology test cases. It is possible by means of the AmmA-based EMFTriple tool that translates the ontologies (involved in the test cases) into metamodels. We have implemented a transformation that translates the reference alignments, used by the ontology systems, to our own format. Furthermore, a bi-directional transformation enables the use of sophisticate tools for matching graphics (e.g., the Alignment API [85]).

7.1.4 Three uses cases based on matching

We have contributed three uses cases to show matching applicability in diverse domains:

- The **co-evolution** use case depicts how migrating transformations can be derived from M2-to-M2 mappings. The solution supports simple and complex migration tasks.

- The **pivot metamodel evaluation** use case presents M2-to-M2 mappings as a notion of distance to evaluate what is the best pivot of a set of metamodels. Moreover, the use case shows the heuristic exploiting Google online corpora in action (i.e., the `MSR` heuristic presented in Section 4.4.4).

- The **model synchronization** use case illustrates how AML can be used to develop M1-to-M1 matching algorithms as well.

7.2 Future Work

This section presents future work grouped in four aspects: language, evaluation, use cases, and tools.

7.2.1 Language

7.2.1.1 AML applicability to real contexts

This work showed the feasibility of reusing matching heuristics independently of technical spaces and abstraction levels. However, it is necessary to validate the approach applicability to real contexts. Some future trends concerning that follow:

- Configuration of further M2-to-M2 matching algorithms. In addition, the algorithms have to be tested not only on metamodels (or ontologies) but on other representation formalisms (e.g., database schemas).

- Testing AML in more M1-to-M1 matching algorithms. Model transformation generation has gained interest in modeling community. In response to this, the bulk of our thesis was devoted to metamodel matching algorithms. In contrast, we have dedicated only one use case (i.e., model synchronization) to model matching, therefore, a future trend is to study AML model matching capabilities.

7.2.1.2 Mapping manipulation construct

For the use cases requiring complex mappings (e.g., co-evolution), we have developed user-defined matching transformations, i.e., ATL transformations. As mapping manipulation logic highly depends on the application domains, more work is needed to determinate whether a set of notations can factorize such a logic.

7.2.1.3 Combining construct

Something concerning (meta)model matching is the risk of low performance when algorithms take large (meta)models as input. The current AML version executes a transformation for each method combined in an algorithm, this impacts performance. It is necessary to have a construct whose semantic is the execution of matching heuristics in a simple step, we imagine a *combining* construct. The idea would be to keep matching heuristics like they are now (i.e., embedding only a concrete matching logic), and to combine them in the `modelsFlow block` by using the combining operator. The AML compiler would be the responsible for combining the heuristics at compilation time. For each combining operator, the compiler would generate an ATL transformation including an `and` condition that chains heuristics comparison criteria. Like that, one keeps AML modularity and one increases performance as well. The combining operator is to be implemented in the future, we plan to combine heuristics conforming to the same kind (e.g., `create`, `sim`, etc).

7.2.1.4 Bootstrapping

For M1-to-M1 matching, AML lets developers explicitly specify `create` matching transformations with their respective types. These matching transformations are necessary to define the searching step scope and then take care of the algorithm performance. A disadvantage is that M1-to-M1 matching algorithms tend to be verbose or complex (as shown in the model synchronization use case). The user has to develop several `create` matching transformations (a heuristic for each pair of types) or only a `create` matching transformation containing a large OCL condition (for validating the pairs of types). A direction to alleviate this issue would be to automate the generation of AML M1-to-M1 `create` matching transformations. Seeing everything like a model (even AML strategies) makes bootstrapping possible.

The idea is (firstly) to match the metamodels which input models conform to, (secondly) the developer marks pairs of correspondences from the output mapping model (each correspondence indicates a $LeftType$ and $RightType$). Finally, a HOT generates an AML program containing a `create` heuristic for each marked correspondence. This

idea is to be implemented in the near future.

7.2.2 Evaluation

7.2.2.1 Stretching the spectrum of test cases

To evaluate matching algorithm accuracy, AML can use reference mappings extracted from model transformations. Our experimentations show that model transformation is a promising niche to get new test cases from. Although our approach is more robust that related work (i.e., MatchBox [6]), it would be interesting to have an extension supporting imperative code. In addition, our reference alignments are inputs that experts may refine in order to improve their quality. Another way of improvement would be to compare our reference alignments with the gold standards extracted by other systems such as MatchBox. In the near future, we want to contribute our test cases to a matching evaluation initiative such as the OAEI.

The experimented test cases represent diverse domains and sizes. However, it would be desired to experiment more test cases including larger (meta)models. Even if there exist open source large (meta)models (e.g., the EAST-ADL metamodel [127]), a problem is the lack of their gold standards in proper formats. These gold standards are often informally defined in text documents. A future trend is to exploit this kind of documents to extract gold standards. Text processing techniques could be useful for that.

7.2.2.2 Further evaluation of model matching systems

Section 5.5 gives the fscores obtained by the AML algorithms. The values give an idea about AML algorithms accuracy, however further benchmarks have to be done to really figure out strengths and weakness of model matching systems (among them AML, MatchBox, etc.). To do that, an important part is to establish common modeling datasets. Another aspect is to create MDE matching evaluation campaigns or to propose a modeling track to a mature matching evaluation initiative (such as the OAEI). Note that we have made efforts in that direction, for example, we have proposed a large modeling dataset. If we contribute such a dataset to the OAEI (and our transformations from reference mapping models to the Alignment API format), it would be possible to compare not only model matching algorithms but also ontology matching systems.

7.2.2.3 Evaluation based on data mining

AML matching algorithms produce intermediate mapping models. In addition, our evaluation approach yields metric models. It would be interesting to plug a data mining tool on top of metric and mapping models. This could help us to infer interesting conclusions from matching results, for instance, how to improve an algorithm in terms of

performance or accuracy.

7.2.3 Use cases

7.2.3.1 Model co-evolution

This use case adapts models to evolving metamodels. An aspect to investigate in the future would be the impact of metamodel evolution on model transformations.

7.2.3.2 Pivot metamodels in the context of interoperability tool

This use case employs fscore as a notion of distance to evaluate pivot metamodels. A future direction may be to explore the count-based approach described in Section 6.2.3.1. Graph visualizations can be built from count-based results: pivot metamodels correspond to nodes and count-based results represent edges. One could use the visualization as a mean to assist: 1) pivot selection, 2) pivot metamodel construction, or 3) model transformation development. An example concerning the third item follows: a graph visualization could suggest if the development of a transformation chain $A \rightarrow C \rightarrow B$ is cheaper than a direct transformation from A to B, where A, B, C are pivot metamodels.

7.2.3.3 Model synchronization

This use case presented an M1-to-M1 matching algorithm including an external transformation devoted to mark changing model elements. When comparing this transformation to the `Differentiation` ATL transformation used in the co-evolution use case, we figure out common patterns. DSL constructs could be extracted from them. Such DSL notations would facilitate the implementation of the diff operation.

7.2.4 Tools

7.2.4.1 Projectors

The AML metamodel importer (see Section 4.3.1.4) translates different formalisms (i.e., OWL, MOF) to Ecore or KM3 (the internal AML formats). Thus, it is possible to apply AML M2-to-M2 matching transformations to models built in other technical spaces. However, the experimentation of Section 5.5 revealed that EMFTriple [24] (i.e., the AmmA-based projector used to translate OWL ontologies into Ecore metamodels) needs a few improvements to fully support the translation, above all, when ontology individuals are involved.

7.2.4.2 User involvement

Our experimentations have shown that efforts are necessary to improve user experience at mapping refinement with AMW. A direction would be to enhance the AMW GUI or integrate AML with EMF Compare [87] (whose GUI has been specialized). In the last case, a transformation from the `Equal` metamodel to the EMF Compare format is needed.

An AML algorithm generates intermediate mapping models. By regarding such models, users could understand how mapping models evolve along the matching process, for example, when a similarity value changes. This could give ideas about how to improve algorithm accuracy. Here the AM3 tools [128], which are based on megamodels, will be

useful to navigate mapping models yielded by matching algorithms.

7.2.4.3 Runtime improvements

Instance-based algorithms reported better runtime than metamodel-only based algorithms. A reason is that the experimented metamodel-only based algorithms include matching transformations invoking Java code which impact performance. Another rationale is that we have manually captured the runtime from the console, as a result, we could have introduced errors in the reported times. Some future work concerning runtime follows:

- Improve runtime when Java code is invoked from AML.
- Create a profiler to automatically log in a model the runtime of AML matching transformations.

Chapter 8

Résume étendu

8.1 Contexte et problématique

L'Ingénierie Dirigée par les Modèles (IDM) est une branche de l'ingénierie du logiciel. Selon [30], l'IDM est une généralisation de la programmation orientée objet (OOP). Les concepts principaux de l'OOP sont les *classes* et les *instances* et deux relations *instance de* et *hérite de*. Un objet est une instance d'une classe et une classe peut étendre une autre classe. Pour l'IDM, le terme fondamental est celui de *modèle*. Un modèle représente un point de vue d'un système et il est défini par le langage de son *métamodèle*. Autrement dit, un modèle contient des *éléments* conforment aux *concepts* et aux relations exprimées dans le métamodèle.

Les deux relations de base entre un *modèle* et son *métamodèle* sont *représenté par* et *conforme à*. Un modèle représente une partie d'un système et il conforme à un métamodèle. De même un métamodèle est conforme à un autre métamétamodèle, habituellement cette régression est stoppée en considérant que le métamodèle "primitif" est conforme à lui même. Un programme, un document XML, une base de données, etc., sont tous des représentations de systèmes informatiques, donc ce sont des modèles objets d'intérêts potentiel pour l'IDM.

Les notions de concepts et d'éléments peuvent correspondre à celles de classes et d'instances respectivement. Ceci suggère seulement des similarités entre l'IDM et l'OOP. Mais en regardant cela de plus près on découvre comme l'IDM complémente l'OOP. Par exemple, les modèles permettent la représentation des classes ainsi que la représentation d'autres aspects d'un système. En outre, l'IDM introduit la notion de transformation de modèle, il s'agit relations ou *alignements* indiquant comme dériver un modèle cible d'un modèle source. Les alignements sont écrits avec les concepts des métamodèles source et cible.

La transformation de modèles est utilisée dans les techniques contemporaines de génération de code.

Le terme modèle est souvent associé aux modèles UML [32]. UML fournit des diagrammes pour représenter non seulement la structure du logiciel (diagrammes de classes) mais également son comportement et ses interactions. UML fait partie de l'initiative MDA, l'approche de modélisation de l'OMG. MDA est l'acronyme anglais de Model Driven Architecture signifiant "architecture dirigée par les modèles". Le but de MDA est de ré-

soudre les problèmes de portabilité, de productivité et d'interopérabilité concernant les systèmes logiciels. Afin de résoudre ces problèmes, le MDA propose la séparation du logiciel en modèles PIM (Platform Independent Model) et modèles PSM (Platform Specific Model). Un modèle PIM considère l'espace du problème et un modèle PSM l'espace de la solution. Un modèle PIM est transformé en un ou plusieurs modèles PSM. Enfin, un modèle PSM est transformé en code. En plus d'UML, MDA recommande d'autres technologies comme : MOF [11], XMI [33], OCL [34], etc. MOF est un métamétamodèle définissant les concepts comme `Classes` et les relations comme `Associations` et `Attributs`. XMI sérialise des modèles dans le format XML. Finalement, OCL permet la définition des requêtes et des contraintes sur des modèles.

Favre [36] suggère MDA comme une incarnation de l'IDM implémentée avec l'ensemble des technologies définies par l'OMG. En outre, Kent [37] trouve que le MDA ne couvre pas toutes les dimensions de l'ingénierie du logiciels (notamment celle du développement de logiciels comme un processus). Ainsi, l'IDM est plus qu'UML et MDA. L'IDM s'étend au delà de l'ingénierie du logiciel pour couvrir d'autres disciplines, dont l'ingénierie de langages [20]. Les langages dédiés (dénotés DSLs par Domain Specific Languages en anglais) ont gagné de l'importance en raison des avantages en termes d'expressivité, et de vérification sur les langages généralistes, e.g. Java.

Kurtev [15] explique le potentiel de l'IDM dans les DSLs : un métamodèle et un ensemble de transformations décrivent la syntaxe abstraite et la sémantique d'un DSL. De plus, les techniques de projection établissent une passerelle entre des *espaces techniques*. Un espace technique est une notion dénotant une technologie, par exemple, l'IDM, EBNF [16], RDF/OWL [17], etc. Chaque espace technique a son métamétamodèle propre. Pour la modélisation de DSLs, les projecteurs relient les espaces techniques de l'IDM et de l'EBNF : ils dérivent des modèles à partir des programmes exprimées dans la syntaxe concrète d'un DSL et vice versa[19]. L'intérêt des projecteurs est de permettre une passerelle simple et pratique entre des espaces techniques très différents et profitant ainsi des avantages offerts par chacun.

L'IDM commence à intéresser fortement l'industrie. Par exemple, le standard AUTOSAR, développé par les constructeurs automobiles, définit un métamodèle de 5000 concepts pour spécifier les architectures de logiciels dans l'automobile[1]. Dans un second temps ce métamodèle a évolué pour répondre à un nouveau cahier des charges. Le problème est que l'on doit maintenant migrer ou faire le parallèle entre les anciens concepts et ceux du nouveau cahier des charges pour créer un nouveau métamodèle pour AUTOSAR. Une approche pour la migration est l'implémentation des transformations de modèles[129]. Toutefois l'industrie a besoin de technologies matures et qui passent à l'échelle. En réponse au défi de la scalabilité (le besoin de grandes (méta)modèles et de transformations de modèles), la recherche académique et l'industrie investissent dans des outils de modélisation. Notamment trois outils actuellement existent : EMF [10], ATL [7], et AM3 [23]. Ce sont les plus populaire grâce à leur nature "open source" et leur communauté d'utilisateurs très actifs.

EMF permet la définition, édition et manipulation de métamodèles ainsi que la génération de code source Java à partir de métamodèles. A l'image de MDA, EMF a son propre métamétamodèle appelé Ecore. MOF et Ecore ont des concepts et relations équivalentes (e.g. `Classes` est similaire à `EClasses`), une différence entre eux est que Ecore contient

des notions spécifiques à Java, par exemple, `EAnnotation`.

Nous avons déjà mentionné comment les résultats de l'IDM peuvent être utilisés dans les DSLs. Inversement, ATL est un DSL pour la définition des transformations de modèles. ATL fournit un ensemble de notations (en partie inspirées par OCL) pour naviguer des modèles source et restreindre la création des éléments cibles. De telles notations sont plus concises et mieux adaptées à l'expression des transformations qu'un langage généraliste comme Java. Pour ATL les transformations sont des modèles, ceci augmente le pouvoir d'automatisation de l'IDM : les transformations de modèles peuvent être générées en utilisant des transformations d'ordre supérieur (dénotées HOTs par Higher-Order Transformations en anglais)[8][9].

AM3 est un outil validant l'approche de mégamodelisation. Un *mégamodèle* est une sorte de carte représentant des artefacts de modélisation ainsi que les relations entre eux. A titre d'illustration, un mégamodèle peut représenter les transformations de modèles associés à un système logiciel issue de l'IDM. Un mégamodèle a pour but de faciliter la compréhension du système, on sait quels sont modèles consommés et produits par les transformations et comment ces dernières interagissent.

Les outils supportant la modélisation et le développement des transformations de modèles atteignent aujourd'hui un certain niveau de maturité. Une prochaine étape est l'automatisation de ces tâches, notamment du développement des transformations. Plusieurs approches ont étudié ce point et une solution est la *découverte des alignements* (nommée *matching* en anglais) [2, 3, 4, 5, 6]. Cette opération a été étudiée par d'autres disciplines comme les bases de données, la réécriture de termes ou le développement des ontologies. Au lieu d'établir des alignements à la main (ce qui est sujet à erreurs et coûteux), une *stratégie d'alignement* (également appelée *algorithme d'alignement*) découvre de manière automatique les liens à établir. Toutefois ce calcul d'alignements ne peut pas être dans les cas réalistes et complexes complètement automatique. Une stratégie d'alignement repose souvent sur un ensemble d'heuristiques, chaque heuristique juge un aspect particulier du métamodèles, par exemple, les noms des concepts ou la structure. Finalement, l'utilisateur peut raffiner les alignements à la main, et à partir d'eux un programme peut dériver une transformation de modèles.

La thèse de Marcos Didonet del Fabro représente des alignements sous la forme d'un *modèle de tissage*[2]. Un modèle de tissage contient des relations entre des éléments de (méta)modèles. Cette notion diffère du terme utilisé dans la programmation orientée aspect (AOP par Aspect Oriented Programming en anglais)[60]. Tandis que la première fait référence au tissage des modèles, AOP tisse du code source exécutable. De plus, [2] implémente une stratégie d'alignement comme une chaîne de transformations d'alignement de modèles, chaque transformation corresponds à une heuristique particulière et est développées avec le langage ATL. La chaîne peut être configurée en sélectionnant des transformations d'alignement des modèles ou des paramètres appropriés. Un outil, nommé AMW, permet le raffinement manuel des alignements. Dans la dernière étape, une HOT dérive une transformation de modèles à partir des alignements découverts et raffinés.

Les résultats de la thèse de Didonet del Fabro et l'intérêt récent de la communauté de l'IDM par l'alignement sont les motivations de cette thèse et son point du départ. Nous étendons le travail de[2], nous ne nous concentrons pas seulement sur les *stratégies d'alignement des métamodèles* mais aussi sur les *stratégies d'alignement des modèles*, ap-

pelées respectivement M2-to-M2 et M1-to-M1. La première sorte de stratégie découvre des alignements entre deux métamodèles. Les transformations de modèles peuvent être dérivées à partir de tels alignements. La dernière sorte d'algorithme détermine des alignements entre deux modèles. Ces alignements sont utiles pour comparer différents points de vue. Par exemple, ils peuvent améliorer l'exactitude des stratégies d'alignement M2-to-M2 ou influencer positivement d'autres opérations de l'IDM, comme la synchronisation de modèlesă: maintenir modèles et code source consistantes lors quŠun système évolue.

Pour supporter le développement des stratégies d'alignement, soit M2-to-M2 or M1-to-M1, il faut s'attaquer à des problèmes qui ne sont pas discutés dans la thèse de Didonet del Fabro. Cette thèse remarque l'importance d'améliorer les algorithmes d'alignements puis qu'aucun algorithme automatique aligne les paires de (méta)modèles d'une manière parfaite. Les expérimentations faites démontrent la possibilité d'utiliser des chaînes de transformation pour améliorer les algorithmes. Cependant, elles révèlent des problèmes concernant la *réutilisabilité des heuristiques d'alignement* et *l'évaluation des algorithmes dŠalignement*.

Notre premier point concerne la reutilisabilité. Les transformations ATL définies par[2] alignent seulement des paires de métamodèles conforment à Ecore. Bien que ces transformations comparent des caractéristiques standards (e.g. les noms), elles peuvent être plus ou moins applicables aux métamodèles conforment à d'autres métamétamodèles (e.g. MOF). En revanche, son applicabilité substantiellement diminue lors qu'on souhaite aligner des modèles. Nous nommons ce problème *couplement des heuristiques d'alignement aux métamodèles*.

Le deuxième point concerne l'évaluation. Il s'agit d'une tâche essentielle dans l'opération d'alignement, elle compare des alignements découverts avec des alignements de référence[12]. Pour évaluer les algorithmes, on a besoin de tests d'usage, c'est-à-dire, des paires de (méta)modèles et des alignements de référence correspondants. Toutes les approches d'alignements de modèles antérieurs à ce travail définissent leurs propres tests d'usage et méthodologies. En conséquence il est difficile d'établir un consensus sur leurs qualités et faiblesses. Nous dénotons ces problèmes comme le *manque d'un ensemble commun de tests d'usage* et une *évaluation déficiente*.

Cette thèse adresse les deux problèmes mentionnés ci-dessus. De plus, pour démontrer que notre travail dépasse les espaces techniques typiques de l'IDM (Ecore, MOF, etc.), nous appliquons nos stratégies d'alignement aux paires des ontologies OWL. Comme les métamodèles, les ontologies sont des formalismes de représentations de données. Une différence entre les metamodèles et les ontologies est le domaine d'application. Dans la dernière décennie, la communauté de l'ingénierie du logiciel a promu les metamodèles alors que les communautés du Web sémantique et de l'intelligence artificielle ont vu émerger les ontologies. Une ontologie est un corpus de connaissances décrivant un domaine particulier au travers d'un vocabulaire de représentation[21]. Par exemple, les ontologies peuvent représenter des ressources Web afin de les rendre manipulables par des programmes. Deux raisons justifient notre choix des ontologies par rapport à d'autres formalismes de représentation (e.g. des schémas de bases de données). Tout d'abord, les ontologies peuvent être traduites en métamodèles. La deuxième et plus importante raison est que la communauté des ontologies a une procédure mature d'évaluation appelée OAEI[22] qui systématiquement évalue des systèmes d'alignements d'ontologies et publie

leurs résultats sur internet. La disponibilité de tels résultats facilite la comparaison de notre travail avec d'autres systèmes.

8.2 Contributions de la thèse

Notre contribution dans cette thèse se décline en quatre points détaillés ci-dessous.

8.2.1 État de l'art des approches d'alignement des modèles

À la différence des approches d'alignement des ontologies (ou des schémas de base de données) qui ont commencé à apparaître il y a 30 ans, les approches d'alignement des (méta)modèles ont fait leur apparition récemment, par conséquence il y a peu d'études. Nous avons contribué à un large état de l'art des approches d'alignement des (méta)modèles. Nous avons adapté les critères d'alignement d'ontologies à l'IDM et nous avons étudié et comparé les approches existantes en utilisant ce critères. La communauté de l'IDM pourra utiliser les critères adaptés pour facilement classifier d'autres approches et faire la mise à jour du de cet état de l'art.

8.2.2 Heuristiques d'alignements de modèles indépendantes des espaces techniques et des niveaux d'abstraction

En regardant les algorithmes existants d'alignements on découvre du code source avec des duplications de code. Même si les algorithmes alignent des (méta)modèles se basent sur des caractéristiques similaires, il faut différents fragments de code pour supporter l'alignement des métamodèles Ecore ou des métamodèles MOF. Cette thèse a étudié comment promouvoir la reutilisabilité des heuristiques d'alignement parmi différents espaces techniques et niveaux d'abstraction (i.e. métamodèles ou modèles) en utilisant un DSL et quelques techniques de modélisation.

Sur la base d'une analyse de domaine, nous avons proposé cinq types d'heuristiques d'alignement. Un algorithme d'alignement est la combinaison d'heuristiques, dont chaque heuristique est l'incarnation d'un type. Des alignements (référençant les modèles source objet de l'alignement) et un processus graduel permettant l'interaction des heuristiques combinées dans un algorithme.

Nous avons contribué à un DSL (nommé AML) dont les notations recouvrent les abstractions mentionnées dans le paragraphe précédent. Leur but est d'autoriser un faible couplage entre les heuristiques d'alignement et un espace technique ou un niveau d'abstraction donné.

Ces notations peuvent être facilement traduites en code exécutable. Nous avons choisi quelques techniques de modélisation pour traduire les programmes AML en modules exécutables. Un compilateur traduit les heuristiques d'alignement AML et le processus graduel en plusieurs transformations ATL et en une chaîne de transformations spécifiées avec Ant[130]. Des (méta)modèles et des *comparaisons* de modèles représentent modèles source et alignements. Un composant additionnel traduit les modèles source dans le format interne d'AML (i.e. un format de modélisation standard comme Ecore) si nécessaire. Nous avons implémenté AML au-dessus de la plate-forme AmmA[15].

En utilisant AML, nous avons développé une bibliothèque d'heuristiques et d'algorithmes. Les heuristiques exploitent l'information linguistique, la structure ou les

instances de données des méta(modèles). Quelques heuristiques utilisent des ressources externes comme un dictionnaire (e.g. WordNet [75]) ou un corpus linguistiques en ligne (e.g. Google). Nous avons réutilisé du code source existant pour interfacer AML avec des ressources externes.

Pour valider que nos algorithmes d'alignement M2-to-M2 dépassent les espaces techniques typiques de l'IDM, nous avons testé les algorithmes non seulement sur des pairs de métamodèles mais aussi sur des pairs d'ontologies. Par ailleurs, nous avons développé un algorithme d'alignement M1-to-M1 pour montrer que quelques heuristiques AML utilisées dans les algorithmes M2-to-M2 sont également réutilisables dans les algorithmes M1-to-M1. Donc, des heuristiques sont indépendantes de l'espace technique et du niveau d'abstraction.

Nos expérimentations ont démontré qu'on peut utiliser AML pour construire des algorithmes d'alignement paramétrables. Chaque algorithme inclus des heuristiques d'alignement *génériques* et *spécifiques*. La première sorte d'heuristique aligne n'importe quelle paire de (méta)modèles, la deuxième est adaptée à certaines paires de (méta)modèles et son but est d'affiner les résultats des heuristiques génériques. De cette manière, les développeurs doivent juste se concentrer sur les heuristiques spécifiques et sur la combinaison des heuristiques génériques et spécifiques. Si les développeurs réutilisent les heuristiques génériques, alors le temps du développement des algorithmes peut être réduit.

AML et sa bibliothèque sont entièrement disponibles sur le site d'Eclipse. Des utilisateurs peuvent télécharger l'outil gratuitement depuis le site et poser des questions sur les forums de discussion. La décision d'implémenter un algorithme d'alignement en utilisant soit un DSL ou un langage généraliste n'est pas claire dans tous les cas. Ceci dépend des priorités. Par exemple, si un chef de projets informatiques souhaite réduire l'effort du développement à court terme, il serait peut-être plus simple de demander aux programmeurs de développer des algorithmes avec un langage généraliste qu'ils utilisent quotidiennement. Par contre, si le but est : 1) de réduire l'effort du développement à long terme (prenant en charge le coût d'apprentissage d'un DSL), et 2) de faciliter aux utilisateurs débutants la compréhension des algorithmes d'alignement, les DSLs, dont AML, semblent une direction prometteuse pour accomplir de tels objectifs.

8.2.3 Artefacts de modélisation pour automatiser l'évaluation des algorithmes d'alignement

L'évaluation permet la classification des algorithmes en termes de qualités et faiblesses. A partir des résultats d'une évaluation les utilisateurs obtiennent des indications sur quel algorithme choisir pour aligner certaines paires de (méta)modèles. Tout d'abord nous proposons une approche pour remédier au problème du manque de test d'usage requis pour l'évaluation des algorithmes. Cette approche extrait des tests d'usage depuis les transformations des modèles : les transformations indiquant les (méta)modèles à aligner ainsi que les alignements de référence. En ayant une large collection de tests d'usage nous pouvons faire des comparaisons plus solides. Ensuite l'approche s'occupe du deuxième problème concernant le performance de l'évaluation. Notre approche exécute automatiquement des algorithmes d'alignement sur des tests d'usage. La clef est de générer un script Ant à partir d'un mégamodèle listant tous les tests d'usages extraits. Ce script Ant fait appel

à un autre script Ant indiquant les actions à exécuter sur les tests. On peut facilement modifier le dernier script pour ajouter, supprimer ou modifier les actions. Par exemple, on peut faire un graphique spécial à partir des résultats d'alignement. Ceci peut augmenter considérablement l'efficacité de l'évaluation, surtout, l'étape du traitement des résultats.

Pour valider que notre approche d'évaluation est applicable dans des espaces techniques différents, nous avons testé la qualité des nos algorithmes sur des tests d'usage des ontologies existantes. Ceci est possible grâce à l'outil AmmA/EMFTriple[24] qui traduit les ontologies (indiquées par les tests) en métamodèles. De plus, nous avons implémenté une transformation pour traduire les alignements de référence dans le format reconnu par notre système. Notamment la transformation bidirectionnelle respective permet l'utilisation des outils de rendérisation des métriques d'alignement (e.g. the Alignment API [85]).

8.2.4 Trois cas d'étude basés sur l'alignement des (méta)modèles

Nous avons contribué à trois cas d'utilisation pour montrer l'applicabilité de l'alignement des (méta)modèles dans plusieurs domaines :

1. La coévolution consiste à adapter les modèles conformes à un métamodèle et qui évoluent dans le temps. Le premier cas d'étude propose une solution de coévolution : un algorithme d'alignement M2-to-M2 découvre les changements simples et complexes entre deux versions d'un métamodèle donné. Ensuite, une HOT dérive une transformation d'adaptation à partir des changements découverts. Rose et al. [29] offre une comparaison d'outils de coévolution de modèles, parmi eux on y trouve AML. Cet article remarque AML comme un outil performant et réducteur de l'effort requit de la part de l'utilisateur pour faire une tâche de coévolution.

2. Le deuxième cas d'étude présente des alignements M2-to-M2 comme une notion de distance pour évaluer quel est le meilleur pivot parmi une collection de métamodèles. Par ailleurs, ce cas d'étude montre le fonctionnement d'une heuristique qu'exploite le corpus linguistique de Google.

3. Finalement, le troisième cas d'étude illustre comment AML peut être utilisé pour développer un algorithme d'alignement M1-to-M1.

8.3 Publications associées à la thèse

1. A Domain Specific Language for Expressing Model Matching. In Actes des Journées sur l'IDM, 2009 [25].

2. Automatizing the Evaluation of Model Matching Systems. In Workshop on matching and meaning, part of the AISB convention, 2010 [26].

3. AML: A Domain Specific Language to Manage Software Evolution. FLFS Poster. Journées de l'ANR, 2010.

4. Adaptation of Models to Evolving Metamodels. Research Report, INRIA, 2008 [27].

5. Managing Model Adaptation by Precise Detection of Metamodel Changes. In Proc. of ECMDA, 2009 [28].

6. A Comparison of Model Migration Tools. In Proc. of Models, 2010 [29].

8.4 Bilan de perspectives

Voici les principales perspectives de recherche qui apparaissent à l'issue de cette thèse :

8.4.1 Diversifier les heuristiques et algorithmes d'alignement

Nos expérimentations montrent que les algorithmes d'alignement M2-to-M2 sont applicables dans plusieurs espaces techniques de l'IDM et également dans des ontologies. Cependant il faudrait tester les algorithmes sur d'autres espaces techniques (e.g. les schémas de bases de données) et améliorer les outils qui traduisent les ontologies en métamodèles, notamment lors-qu'une ontologie contient des instances de données. À l'exception du troisième cas d'étude, la thèse est consacrée à l'alignement M2-to-M2, donc il serait nécessaire de regarder de plus près l'efficacité d'AML dans le développement des algorithmes M1-to-M1. Nous avons contribué à une bibliothèque d'heuristiques que ne dépasse pas en taille les bibliothèques existantes (e.g. Coma++ [62]). La justification est que nous avons voulu tester l'efficacité que les heuristiques, proposées dans d'autres contextes, ont dans l'IDM. Il serait souhaitable d'explorer d'autres heuristiques. Notamment les heuristiques exploitant les instances de données et les remarques de la part des utilisateurs.

8.4.2 Élargir la collection de tests d'usage

Nos expérimentations démontrent que les transformations de modèles sont une source prometteuse de tests d'usage, dont, des alignements de référence et des pairs de (méta)modèles. Notre approche exploite surtout la partie déclarative des transformations. Afin de profiter au maximum des transformations, il faudra exploiter également la partie impérative. Par ailleurs, des experts pourraient raffiner les alignements de référence extraits et donc augmenter leur qualité.

Les tests d'usage extraits couvrent des domaines et des tailles diverses, cependant, il serait intéressant d'expérimenter des tests d'usage encore plus larges. Même si des métamodèles larges sont disponibles en 'open source' (e.g. EAST-ADL [127]), un problème est le manque d'alignements de référence dans un format approprié. Ces alignements de référence sont fréquemment définis dans des documents textuels. Une perspective serait d'exploiter ce type de documents pour obtenir des alignements. Là, des techniques de traitement de texte peuvent être utiles.

8.4.3 Évaluer d'avantage les systèmes d'alignement existants

Nos expérimentations donnent une idée de l'exactitude des algorithmes AML par rapport à deux systèmes d'alignement (i.e. the Alignment API [85] et MatchBox [6]). Pourtant, il serait souhaitable de comparer d'avantage les systèmes d'alignement existants (y compris AML, the alignment API et MatchBox) pour approfondir nos connaissances sur leurs qualités et faiblesses lors qu'on aligne des (méta)modèles. Pour accomplir cet objectif il faudrait établir une très large collection de tests d'usage ainsi qu'une procédure d'évaluation. Nous avons fait des efforts dans cette direction : nous avons généré une collection de tests d'usage et nous comptons contribuer à une initiative d'évaluation mature,

par exemple l'OAEI[22].

8.4.4 Perspectives sur les cas d'étude

8.4.4.1 Coévolution des modèles

La première version d'AML supporte la plupart des changements complexes fixés par la classification de Wachsmuth [105] à l'exception des changements nommés "breaking and non resolvable changes". Une perspective serait de supporter tous les types de changement ainsi que leur impact sur les transformations de modèles.

8.4.4.2 Évaluation des métamodèles pivots

Nous avons utilisé une métrique pour mesurer le niveau de superposition entre un ensemble de métamodèles et donc évaluer le meilleur métamodèle pivot. Il serait intéressant de tester d'autres métriques et d'analyser non seulement l'évaluation mais aussi la sélection et la construction des métamodèles pivots.

8.4.4.3 Synchronisation des modèles

A l'image du première cas d'étude, le troisième cas a requis une transformation de différentiation. Nous avons comparé de telles transformations et nous avons remarqué du code source commun, donc nous envisageons un nouveau DSL que faciliterait leur développement.

Bibliography

[1] AUTOSAR Development Partnership: AUTOSAR specification V3.1. (2008)

[2] Didonet del Fabro, M.: Metadata management using model weaving and model transformation. PhD thesis, Université de Nantes (2007)

[3] Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: MoDELS. Volume 5301 of Lecture Notes in Computer Science., Springer (2008) 326–340

[4] Kargl, H., Wimmer, M.: Smartmatcher - how examples and a dedicated mapping language can improve the quality of automatic matching approaches. In: CISIS. (2008) 879–885

[5] Kolovos, D.S.: Establishing correspondences between models with the epsilon comparison language. In: ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, Berlin, Heidelberg, Springer-Verlag (2009) 146–157

[6] Voigt, K., Ivanov, P., Rummler, A.: Matchbox: combined meta-model matching for semi-automatic mapping generation. In: SAC. (2010) 2281–2288

[7] Jouault, F., Kurtev, I.: Transforming models with ATL. In: Proceedings of the Model Transformations in Practice Workshop, MoDELS 2005, Montego Bay, Jamaica (2005)

[8] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: In Proc. of ECMDA 2009, Enschede, The Netherlands, Springer (june 2009)

[9] Tisi, M., Cabot, J., Jouault, F.: Improving higher-order transformations support in ATL. In: International Conference on Model Transformation (ICMT 2010). (2010) to appear

[10] Hussey, K., Paternostro, M.: Tutorial on advanced features of EMF. In: EclipseCon, `http://www.eclipsecon.org/2006/Sub.do?id=171` (Retrieved June 2010)

[11] OMG: MOF Specification, version 1.4, OMG document formal/2002-04-03. (2002)

[12] OAEI: Towards a methodology for evaluating alignment and matching algorithms, `http://oaei.ontologymatching.org/doc/oaei-methods.1.pdf`. (Retrieved June 2010)

[13] Euzenat, J., Shvaiko, P.: Ontology Matching. Springer, Heidelberg (DE) (2007)

[14] Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. The VLDB Journal **10**(4) (2001) 334–350

[15] Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, OR, USA, ACM (2006) 602–616

[16] Parr, T.J., Quong, R.W.: ANTLR: A predicated-LL(k) parser generator. Software—Practice and Experience **25**(7) (July 1995) 789–810

[17] W3C: Resource Description Framework, `http://www.w3.org/RDF/`. (Retrieved June 2010)

[18] W3C: OWL Web Ontology Language, `http://www.w3.org/TR/owl-features/`. (Retrieved June 2010)

[19] Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L., eds.: Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, Proceedings, ACM (2006) 249–254

[20] Bézivin, J., Heckel, R.: 04101 Summary Language Engineering for Model-driven Software Development. In: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany (2005)

[21] Chandrasekaran, B., Josephson, J., Benjamins, R.: What are ontologies, and why do we need them? IEEE Intelligent Systems **14**(1) (1999)

[22] OAEI: Ontology Alignment Evaluation Initiative, `http://oaei.ontologymatching.org/`. (Retrieved June 2010)

[23] Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2004)

[24] Hillairet, G.: EMFTriple, `http://code.google.com/p/emftriple/`. (2009)

[25] Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: A Domain Specific Language for Expressing Model Matching. In: Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09), Nancy, France (2009)

[26] Garcés, K., Kling, W., Jouault, F.: Automatizing the evaluation of model matching systems. In: Workshop on matching and meaning 2010, Leicester, United Kingdom (2010) 7 – 12

[27] Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Adaptation of models to evolving metamodels. Technical report, INRIA (2008)

[28] Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing Model Adaptation by Precise Detection of Metamodel Changes. In: In Proc. of ECMDA 2009, Enschede, The Netherlands, Springer (june 2009)

[29] Rose, L., Herrmannsdoerfer, M., Williams, J., Kolovos, D., Garcés, K., Piage, R., Polack, F.: A comparison of model migration tools. In: Models 2010. (2010) to appear

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

[31] Greenfield, J., Short, K.: Software factories: assembling applications with patterns, models, frameworks and tools. In Crocker, R., Jr, G.L.S., eds.: OOPSLA Companion, ACM (2003) 16–27

[32] OMG: UML 2.0 Infrastructure Specification OMG document ptc/03-09-15. (2003)

[33] OMG: XMI 2.1.1 XML Metadata Interchange. OMG document formal/07-12-01

[34] OMG: OCL 2.0 Specification, OMG Document formal/2006-05-01. (2006)

[35] Boström, P., Neovius, M., Oliver, I., Waldén, M.A.: Formal transformation of platform independent models into platform specific models. In Julliand, J., Kouchnarenko, O., eds.: B. Volume 4355 of Lecture Notes in Computer Science., Springer (2007) 186–200

[36] Favre, J.M.: Towards a basic theory to model model driven engineering. In: Workshop in Software Model Engineering. In conjunction with UML2004, Portugal (2004)

[37] Kent, S.: Model driven engineering. In: Proceedings of IFM 2002. LNCS 2335, Springer-Verlag (unknown 2002) 286–298

[38] Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Bologna, Italy (2006) 171–185

[39] Vara, J.M.: M2DAT: a technical solution for model-driven development of Web information systems. PhD thesis, University Rey Juan Carlos (2009)

[40] Sun Microsystems: Java Metadata Interface (JMI) Specification. (June 2002)

[41] OMG: MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10. (2002)

[42] Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In Haddad, H., ed.: SAC, ACM (2006) 1280–1287

[43] Taentzer, G., Carughi, G.T.: A graph-based approach to transform XML documents. In Baresi, L., Heckel, R., eds.: FASE. Volume 3922 of Lecture Notes in Computer Science., Springer (2006) 48–62

[44] Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In Briand, S.K.L., ed.: Proceedings of MODELS/UML'2005. Volume 3713 of LNCS., Montego Bay, Jamaica, Springer (October 2005) 264–278

[45] OMG: MOF QVT Final Adopted Specification, OMG document ptc/2005-11-01. (2005)

[46] Kurtev, I.: State of the art of QVT: A model transformation language standard. In Schürr, A., Nagl, M., Zündorf, A., eds.: AGTIVE. Volume 5088 of Lecture Notes in Computer Science., Springer (2007) 377–393

[47] Laarman, A.: Achieving QVTO & ATL interoperability. In: Model transformation with ATL, 1st international workshop mtATL 2009, Nantes, France (2009) 119–133

[48] Vanhooff, B., Ayed, D., Baelen, S.V., Joosen, W., Berbers, Y.: Uniti: A unified transformation infrastructure. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: MoDELS. Volume 4735 of Lecture Notes in Computer Science., Springer (2007) 31–45

[49] Gašević, D., Djurić, D., Devedzic, V.: Model driven architecture and ontology development. Springer Verlag, pub-SV:adr (2006)

[50] Obeo: Acceleo: MDA generator, `http://www.acceleo.org/pages/home/en`. (Retrieved June 2010)

[51] Oldevik, J., Neple, T., Grønmo, R., Aagedal, J.Ø., Berre, A.J.: Toward standardised model to text transformations. In Hartman, A., Kreische, D., eds.: ECMDA-FA. Volume 3748 of Lecture Notes in Computer Science., Springer (2005) 239–253

[52] Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. CSURV: Computing Surveys **37** (2005)

[53] van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography, `http://homepages.cwi.nl/~arie/papers/dslbib/dslbib.html#tex2html1`. (Retrieved August 2010)

[54] Simos, M., Creps, D., Klinger, C., Levine, L., , Allemang, D.: Organization domain modelling (ODM) guidebook version 2.0. Technical report, Synquiry Technologies, Inc (1996)

[55] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, 1990 (Software Engineering Institute, Carnegie Mellon University)

[56] Taylor, R.N., Tracz, W., Coglianese, L.: Software development using domain-specific software architectures. In: ACM SIGSOFT Software Engineering Notes. (1995) 27–37

[57] Pollice, G.: Compiler vs. Interpreter, `http://web.cs.wpi.edu/~gpollice/cs544-f05/CourseNotes/maps/Class1/Compilervs.Interpreter.html`. (Retrieved August 2010)

[58] Jouault, F.: Contribution to the study of model transformation languages. PhD thesis, Université de Nantes (2006)

[59] Didonet Del Fabro, M., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: A generic model weaver. In: Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05). (2005)

[60] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: ECOOP 2001 — Object-Oriented Programming 15th European Conference. Volume 2072 of Lecture Notes in Computer Science. Springer-Verlag, Budapest, Hungary (June 2001) 327–353

[61] Kolovos, D., Ruscio, D.D., Pierontino, A., Piage, R.: Different models for model matching: An analysis of approaches to support model differencing. In: CVSM'09. (2009)

[62] Do, H.H.: Schema Matching and Mapping-based Data Integration. PhD thesis, University of Leipzig (2005)

[63] Mitra, P., Noy, N.F., Jaiswal, A.R.: Ontology mapping discovery with uncertainty (2005)

[64] Doan, A., Domingos, P., Halevy, A.: Learning to match the schemas of data sources. A multistrategy approach. Machine Learning **50**(3) (2003) 279–301

[65] Li, W.S., Clifton, C.: Semantic integration in heterogeneous databases using neural networks. In: Proceedings of the Twentieth International Conference on Very Large Databases, Santiago, Chile (1994) 1–12

[66] Li, W.S., Clifton, C., Liu, S.Y.: Database integration using neural networks: Implementation and experiences. Knowl. Inf. Syst **2**(1) (2000) 73–96

[67] Li, Y., Liu, D.B., Zhang, W.M.: Schema matching using neural network. In: WI '05: Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence, Washington, DC, USA, IEEE Computer Society (2005) 743–746

[68] Ehrig, M., Staab, S., Sure, Y.: Bootstrapping ontology alignment methods with APFEL (2005)

[69] Lee, Y., Sayyadian, M., Doan, A., Rosenthal, A.S.: eTuner: tuning schema matching software using synthetic scenarios. The VLDB Journal **16**(1) (2007) 97–122

[70] Ehrig, M.: Ontology Alignment: Bridging the Semantic Gap. Volume 4 of Semantic Web And Beyond Computing for Human Experience. Springer (2007)

[71] Knuth, D.E.: The Art of Computer Programming. second edn. Volume 2: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts (1973)

[72] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady. (1966) 707–710

[73] Needleman, S., Wunsch, C.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology (1970) 443–453

[74] Wikipedia, t.f.e.: Hyponym, `http://en.wikipedia.org/wiki/Hyponymy`. (Retreived July 2010)

[75] Miller, G.A.: WordNet: A lexical database for english. In: Communications of the ACM Vol. 38, No. 11. (1995) 39–41

[76] Lee, M.L., Yang, L.H., Hsu, W., Yang, X.: XClust: clustering XML schemas for effective integration. In: CIKM, ACM (2002) 292–299

[77] Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: A versatile graph matching algorithm and ist application to schema matching. In: Proc. 18th ICDE, San Jose, CA (2002)

[78] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management, New York, NY, USA, ACM (2006) 5–12

[79] Hunt, J.W., McIlroy, M.D.: An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ (1976)

[80] Engmann, D., Maßmann, S.: Instance matching with coma++. In: BTW Workshops. (2007) 28–37

[81] Wang, T., Pottinger, R.: Semap: a generic mapping construction system. In Kemper, A., Valduriez, P., Mouaddib, N., Teubner, J., Bouzeghoub, M., Markl, V., Amsaleg, L., Manolescu, I., eds.: EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings. Volume 261 of ACM International Conference Proceeding Series., ACM (2008) 97–108

[82] Doan, A., Domingos, P., Halevy, A.: Reconciling schemas of disparate data sources: A machine-learning approach. In: In SIGMOD Conference. (2001) 509–520

[83] Maedche, A., Motik, B., Silva, N., Volz, R.: MAFRA - an ontology MApping FRAmework in the context of the semantic web. In: ECAI-Workshop on Knowledge Transformation for the Semantic Web, Lyon, France (07 2002)

[84] Kensche, D., Quix, C., 0002, X.L., Li, Y.: GeRoMeSuite: A system for holistic generic model management. In Koch, C., Gehrke, J., Garofalakis, M.N., Srivastava, D., Aberer, K., Deshpande, A., Florescu, D., Chan, C.Y., Ganti, V., Kanne, C.C., Klas, W., Neuhold, E.J., eds.: Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007, ACM (2007) 1322–1325

[85] Euzenat, J.: An API for ontology alignment. In McIlraith, S.A., Plexousakis, D., van Harmelen, F., eds.: International Semantic Web Conference. Volume 3298 of Lecture Notes in Computer Science., Springer (2004) 698–712

[86] Fleurey, F., Baudry, B., France, R.B., Ghosh, S.: A generic approach for automatic model composition. In Giese, H., ed.: MoDELS Workshops. Volume 5002 of Lecture Notes in Computer Science., Springer (2007) 7–15

[87] Toulme, A.: Presentation of EMF compare utility. In: Eclipse Modeling Symposium. (2006)

[88] Falleri, J.R.: Generic and Useful Model Matcher, `http://code.google.com/p/gumm-project/`. (Retrieved January 2009)

[89] Konrad, V.: Towards combining model matchers for transformation developments. In: 1st International Workshop on Future Trends of Model-Driven Development at ICEIS'09. (2009)

[90] Didonet Del Fabro, M., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: SAC. (2007) 963–970

[91] Noy, F., N., Musen, A., M.: The PROMPT suite: interactive tools for ontology merging and mapping. International Journal of Human-Computer Studies **59**(6) (2003) 983–1024

[92] Melnik, S.: Generic Model Management: Concepts and Algorithms. PhD thesis, University of Leipzig (2004)

[93] Euzenat, J., Ferrara, A., Hollink, L., Isaac, A., Joslyn, C., Malaisé, V., Meilicke, C., Nikolov, A., Pane, J., Sabou, M., Scharffe, F., Shvaiko, P., Spiliopoulos, V., Stuckenschmidt, H., Ľváb Zamazal, O., Svátek, V., Trojahn, C., Vouros, G., Wang, S.: First results of the ontology alignment evaluation iniciative 2009. In: Proceedings of the 4th International Workshop on Ontology Matching, Collocated with the 8th International Semantic Web Conference. (2009)

[94] Chapman, S.: SimMetrics, `http://sourceforge.net/projects/simmetrics/`. (2009)

[95] Eclipse.org: Model to Model (M2M), `http://www.eclipse.org/m2m/`. (Retrieved June 2010)

[96] Veksler, V.D., Grintsvayg, A., Lindsey, R., Gray, W.D.: A proxy for all your semantic needs. In: In Proc. CogSci 2007. (2007)

[97] Bézivin, J., Brunelière, H., Jouault, F., Kurtev, I.: Model engineering support for tool interoperability. In: Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica (2005)

[98] Jaccard, P.: Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. Bulletin de la Société Vaudoise des Sciences Naturelles **37** (1901) 241–272

[99] Larson, J.A., Navathe, S.B., Elmasri, R.: A theory of attribute equivalence in databases with application to schema integration. tose **15**(4) (April 1989) 449–463

[100] Eclipse.org: TCS project, `http://www.eclipse.org/gmt/tcs/`. (2008)

[101] Seddiqui, M.H., Aono, M.: An efficient and scalable algorithm for segmented alignment of ontologies of arbitrary size. Web Semant. **7**(4) (2009) 344–356

[102] David, J., Guillet, F., Briand, H.: Matching directories and OWL ontologies with AROMA. In: CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management, New York, NY, USA, ACM (2006) 830–831

[103] Davis, J., Goadrich, M.: The relationship between Precision-Recall and ROC curves. In: The 23rd international conference on Machine learning, Pittsburgh, Pennsylvania (2006) 233–240

[104] Meneses, R., Casallas, R.: A strategy for synchronizing and updating models after source code changes in Model-Driven Development. In: Models and Evolution: Joint MoDSE-MCCM 2009 Workshop on Model-Driven Software Evolution (MoDSE) Model Co-Evolution and Consistency Management (MCCM). (2009) 186–189

[105] Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In Ernst, E., ed.: Object-Oriented Programming, 21st European Conference, ECOOP 2007, Berlin, Germany, Proceedings. Volume 4609 of Lecture Notes in Computer Science., Springer (2007) 600–624

[106] Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. SIGSOFT Softw. Eng. Notes **28**(5) (2003) 227–236

[107] Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE '05, New York, NY, USA, ACM (2005) 54–65

[108] Girschick, M.: Difference detection and visualization in UML class diagrams. Technical report, TU Darmstadt (2006)

[109] Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In Crnkovic, I., Bertolino, A., eds.: ESEC/SIGSOFT FSE, ACM (2007) 295–304

[110] Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting collaborative development in an open MDA environment. In: ICSM, IEEE Computer Society (2006) 244–253

[111] Wenzel, S., Kelter, U.: Analyzing model evolution. In Robby, ed.: ICSE, ACM (2008) 831–834

[112] Eclipse.org: EMF Compare, `http://wiki.eclipse.org/index.php/EMF_Compare`. (2008)

[113] Gruschko, B., Kolovos, D., Paige., R.: Towards synchronizing models with evolving metamodels. In: Workshop on Model-Driven Software Evolution, MODSE 2007,Amsterdam, the Netherlands. (2007)

[114] Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in Model-Driven Engineering. In: EDOC '08: Proceedings of the 12th IEEE International EDOC Conference, München, Germany (2008)

[115] Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: MoDELS. Volume 5301 of Lecture Notes in Computer Science., Springer (2008) 645–659

[116] Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. Lecture Notes in Computer Science **5301** (September 2008) 630–644 In K. Czarnecki and I. Ober and J.-M. Bruel and A. Uhl and M. Voelter (eds.) Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008). Toulouse, France, October 2008.

[117] Rose, L., Kolovos, D., Paige, R., Polack, F.: Model migration with Epsilon Flock. In: ICMT. (2010)

[118] Sun, Y., Demirezen, Z., Jouault, F., Tairas, R., Gray, J.: A model engineering approach to tool interoperability. In Gasevic, D., Lämmel, R., Wyk, E.V., eds.: SLE. Volume 5452 of Lecture Notes in Computer Science., Springer (2008) 178–187

[119] Bézivin, J., Brunelière, H., Jouault, F., Kurtev, I.: Model Engineering Support for Tool Interoperability. In: Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica (2005)

[120] Wikipedia.org: Apache Ant, `http://en.wikipedia.org/wiki/Apache_Ant`. (Retrieved June 2010)

[121] Brank, J., Grobelnik, M., Mladenić, D.: Automatic evaluation of ontologies. Springer (2007)

[122] Van Belle, J.P.: A simple metric to measure semantic overlap between models: Application and visualization. In: Internet and Information Technology in Modern Organizations: Challenges & Answers, Proceedings of The 5th International Business Information Management Association Conference. (2005)

[123] Bouaud, J., Séroussi, B.: Automatic generation of a metamodel from an existing knowledge base to assist the development of a new analogous knowledge base. In: AMIA 2002 Annual Symposium. (2002) 66–70

[124] Yie, A., Wagelaar, D.: Advanced traceability for ATL. In: 1st International Workshop on Model Transformation with ATL (MtATL 2009). (2009) pp.78–87

[125] Rose, L.M., Herrmannsdoerfer, M., Williams, J.R., Kolovos, D.S., Garcés, K., Paige, R.F., Polack, F.A.: A comparison of model migration tools. In: MoDELS, Springer (2010)

[126] Eclipse.org: ATL Use Cases, `http://www.eclipse.org/m2m/atl/usecases/ModelsMeasurement/`. (2009)

[127] ATESST: EAST ADL 2.0 specification, `http://www.atesst.org/home/liblocal/docs/EAST-ADL-2.0-Specification_2008-02-29.pdf`. (Retrieved June 2010)

[128] Eclipse.org: The AtlanMod MegaModel Management (AM3) Project, `http://www.eclipse.org/gmt/am3/`

[129] Herrmannsdoerfer, M., Benz, S., Jürgens, E.: COPE - automating coupled evolution of metamodels and models. In Drossopoulou, S., ed.: ECOOP. Volume 5653 of Lecture Notes in Computer Science., Springer (2009) 52–76

[130] Apache.org: The Apache Ant, `http://ant.apache.org/`. (2008)

[131] Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform random generation of huge metamodel instances. In Paige, R.F., Hartman, A., Rensink, A., eds.: ECMDA-FA. Volume 5562 of Lecture Notes in Computer Science., Springer (2009) 130–145

List of Abbreviations

AML AtlanMod Matching Language, page 3

AmmA AtlanMod Model Management Architecture, page 3

AMW AtlanMod Model Weaver, page 17

DSL Domain Specific Language, page 3

GPL General Purpose Language, page 3

KM3 Kernel MetaMetaModel, page 9

M1-to-M1 Model matching strategy, page 2

M2-to-M2 Metamodel matching strategy, page 2

MDA Model-Driven Architecture, page 6

MDE Model-Driven Engineering, page 1

MOF Meta-Object Facility, page 6

OCL Object Constraint Language, page 6

OMG Object Management Group, page 6

OWL Web Ontology Language, page 3

QVT Query/Views/Tranformations, page 12

RDF Resource Description Framework, page 3

SQL-DDL SQL Data Definition Language, page 9

TCS Textual Concrete Syntax, page 19

UML Unified Modeling Language, page 6

XMI XML Metadata Interchange, page 6

XML Extensible Markup Language, page 1

XSD XML Schema Definition, page 9

List of Figures

List of Tables

List of listings

Appendix A: AML abstract syntax

This appendix gives the AML abstract syntax in two formats: class diagrams and KM3 code. The KM3 version contains comments (highlighted in green) indicating where the code of the main AML constructs (i.e., *import*, *models block*, *matching method*, and *models flow block*) stars and ends.

Listing 1: AML abstract syntax in KM3 notation

```
class Matcher extends MElement {
 reference methods[*] container : Method oppositeOf matcher;
 reference matchers[*] container : MatcherRef oppositeOf unit;
 reference modelsBlock[0-1] container : ModelsBlock oppositeOf matcher;
 reference modelsFlowsBlock[0-1] container : ModelsFlowsBlock oppositeOf matcher;
 reference referenceModels[*] container : ReferenceModel oppositeOf matcher;
}

-- @begin Import

class MatcherRef extends LocatedElement {
 reference unit : Matcher oppositeOf matchers;
 attribute name : String;
}

-- @begin Import

-- @begin Models block

class ModelsBlock extends LocatedElement {
 reference models[*] ordered container : Model;
 reference matcher : Matcher oppositeOf modelsBlock;
}

abstract class Model extends LocatedElement {
 attribute name : String;
 reference referenceModel container : ReferenceModel oppositeOf models;
}

class WeavingModel extends Model {
 reference wovenModels[*] container : InputModel;
}

class MappingModel extends Model {
 reference leftModel[0-1] container : InputModel;
 reference rightModel[0-1] container : InputModel;
}

class InputModel extends Model {}

class ReferenceModel extends LocatedElement {
 attribute name : String;
 reference elements[*] : MetaElement oppositeOf referenceModel;
 reference models[*] : Model oppositeOf referenceModel;
 reference matcher : Matcher oppositeOf referenceModels;
}
```

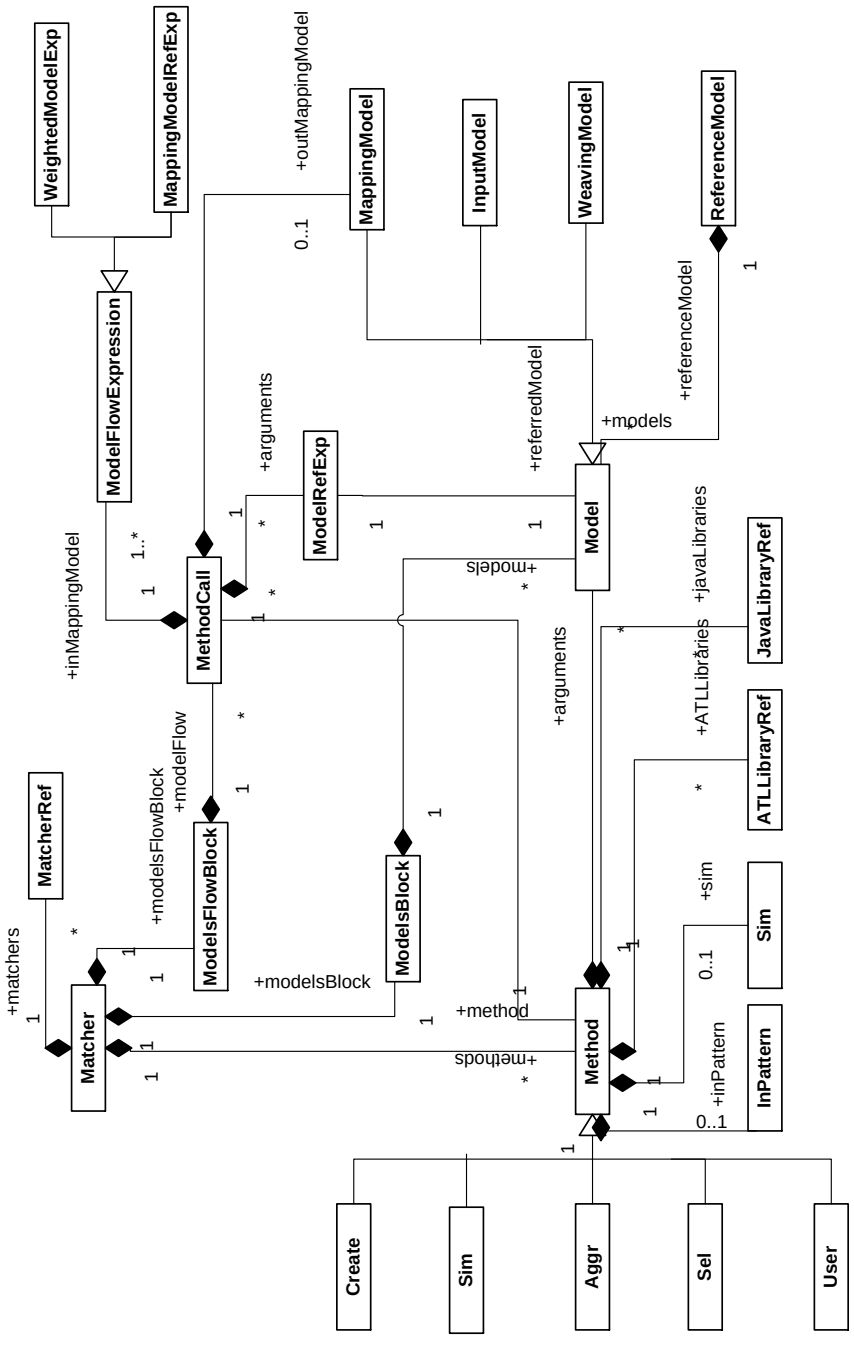


Figure 1: AML metamodel

```
47
48 -- @end Models block
49
50 -- @begin Matching methods
51
52 class Method extends MElement {
53  reference inMappingModel[1-*] container : MappingModel;
54  reference arguments[*] container : Model;
55  reference inPattern container : InPattern oppositeOf method;
56  reference outPattern container : OutPattern oppositeOf method;
57  reference sim[0-1] container : Sim oppositeOf method;
58  reference variables[*] ordered container : RuleVariableDeclaration oppositeOf method;
59  reference matcher : Matcher oppositeOf methods;
60  reference ATLLibraries[*] container : ATLLibraryRef oppositeOf method;
61  reference javaLibraries[*] container : JavaLibraryRef oppositeOf method;
62 }
63
64 class CreateEqual extends Method {
65  reference equalInPattern[0-1] container : EqualInPattern oppositeOf method;
66 }
67
68 class SimEqual extends Method {}
69
70 class AggrEqual extends Method {}
71
72 class SelEqual extends Method {}
73
74 class ExternalMethod extends Method {}
75
76 abstract class LibraryRef extends LocatedElement {
77  attribute name : String;
78  attribute path : String;
79 }
80
81 class ATLLibraryRef extends LibraryRef {
82  reference method : Method oppositeOf ATLLibraries;
83 }
84
85 class JavaLibraryRef extends LibraryRef {
86  reference method : Method oppositeOf javaLibraries;
87 }
88
89 class InPattern extends LocatedElement {
90  reference elements[1-*] container : InPatternElement oppositeOf inPattern;
91  reference method : Method oppositeOf inPattern;
92  reference filter[0-1] container : OclExpression;
93 }
94
95 class EqualInPattern extends LocatedElement {
96  reference rightElement container : EqualMetaElement;
97  reference leftElement container : EqualMetaElement;
98  reference method : CreateEqual oppositeOf equalInPattern;
99 }
100
101 abstract class PatternElement extends VariableDeclaration {}
102
103 abstract class InPatternElement extends PatternElement {
104  reference mapsTo : OutPatternElement oppositeOf sourceElement;
105  reference inPattern : InPattern oppositeOf elements;
106  reference models[0-*] : Model;
107 }
108
109 class SimpleInPatternElement extends InPatternElement {}
110
111
112 class Sim extends LocatedElement {
113  reference value container : OclExpression;
114 }
```

```
115
116 -- @end Matching methods
117
118 -- @begin ModelsFlow block
119
120 class ModelsFlowsBlock extends LocatedElement {
121  reference matcher : Matcher oppositeOf modelsFlowsBlock;
122  reference modelsFlows[*] ordered container : MethodCall oppositeOf block;
123 }
124
125 abstract class ModelFlowExpression extends LocatedElement {}
126
127 class MappingModelRefExp extends ModelFlowExpression {
128  reference referredMappingModel : MappingModel;
129 }
130
131 class ModelRefExp extends LocatedElement {
132  reference methodCall : MethodCall oppositeOf arguments;
133  reference referredModel : Model;
134 }
135
136 class MethodCall extends ModelFlowExpression {
137  reference method : Method;
138  reference outMappingModel[0-1] container : MappingModel;
139  reference inMappingModel[1-*] container : ModelFlowExpression;
140  reference arguments[*] ordered container : ModelRefExp oppositeOf methodCall;
141  reference block : ModelsFlowsBlock oppositeOf modelsFlows;
142 }
143
144 class WeightedModelExp extends ModelFlowExpression {
145  attribute weight : Double;
146  reference modelFlowExp container : ModelFlowExpression;
147 }
148
149 -- @end ModelsFlow block
150
151 -- @begin OCL
152
153 abstract class OclExpression extends LocatedElement {}
154
155 class ThisModuleExp extends OclExpression {}
156
157 class ThisEqualExp extends OclExpression {}
158
159 class ThisSimExp extends OclExpression {}
160
161 class ThisInstancesExp extends OclExpression {
162  reference instancesOp container : OclExpression;
163 }
164
165 abstract class ThisNodeExp extends OclExpression {}
166
167 class ThisRightExp extends ThisNodeExp {}
168
169 class ThisLeftExp extends ThisNodeExp {}
170
171 class EqualSim extends OclExpression {}
172
173 class ThisWeightExp extends OclExpression {}
174
175 class ThisEqualModelExp extends OclExpression {}
176
177 class SummationExp extends OclExpression {
178  reference sumExpression container : OclExpression;
179 }
180
181 -- @end OCL
```

Appendix B: AML concrete syntax

This appendix gives the AML concrete syntax written with TCS. Comments (highlighted in green) indicate where the code of the main AML constructs (i.e., *import*, *models block*, *matching method*, and *models flow block*) stars and ends.

Listing 2: AML concrete syntax in TCS notation

```
template Matcher main context
 : "strategy" name "{"
  [
   matchers
   methods
   (isDefined(modelsBlock)? [modelsBlock])
   (isDefined(modelsFlowsBlock)? [modelsFlowsBlock])
  ]
   "}"
 ;

-- @begin Import

 template MatcherRef
 : "imports" name
 ;

-- @end Import

-- @begin Models block

template ModelsBlock
 : "models" "{" [
  models
   ] "}"
 ;
template Model abstract;

template InputModel addToContext
 : name ":" referenceModel{refersTo = name, lookIn = #all, autoCreate = ifmissing};

template MappingModel addToContext
 : name
  (isDefined(leftModel) and isDefined(rightModel)?
   ":" "EqualModel" "(" leftModel "," rightModel ")"
   )
 ;

template WeavingModel addToContext
 : name
  ":" "WeavingModel"
  "(" referenceModel ")"
  "(" wovenModels {separator = ","} ")";

template ReferenceModel
 : name{autoCreate = ifmissing, createIn = '#context'.referenceModels}
 ;
```

```
48
49  template MetaElement
50   : referenceModel{refersTo = name, lookIn = #all, autoCreate = ifmissing} "!" name
51   ;
52
53  template EqualMetaElement
54   : name
55   ;
56
57  -- @end Models block
58
59  -- @begin Matching methods
60
61  template Method abstract addToContext;
62
63  template CreateEqual context
64   :  "create" name "(" arguments{separator = ","} ")"
65    (isDefined(ATLLibraries) ? "ATLLibraries" "{" [ATLLibraries {separator = ","} ] "}")
66    (isDefined(javaLibraries) ? "JavaLibraries" "{" [javaLibraries {separator = ","} ] "}"
       ↪)
67    "{" [
68     (isDefined(equalInPattern) ?
69      equalInPattern
70     )
71     inPattern
72     (isDefined(variables) ?
73       "using" "{" [
74        variables
75       ] "}"
76     )
77    ] "}"
78   ;
79
80  template SimEqual context
81   :  "sim" name "(" arguments{separator = ","} ")"
82    (isDefined(ATLLibraries) ? "ATLLibraries" "{" [ATLLibraries {separator = ","} ] "}")
83    (isDefined(javaLibraries) ? "JavaLibraries" "{" [javaLibraries {separator = ","} ] "}"
       ↪)
84    "{" [
85     inPattern
86     (isDefined(variables) ?
87       "using" "{" [
88        variables
89       ] "}"
90     )
91     sim
92    ] "}"
93   ;
94
95  template AggrEqual context
96   :  "aggr" name "(" arguments{separator = ","} ")"
97    (isDefined(ATLLibraries) ? "ATLLibraries" "{" [ATLLibraries {separator = ","} ] "}")
98    (isDefined(javaLibraries) ? "JavaLibraries" "{" [javaLibraries {separator = ","} ] "}"
       ↪)
99    "{" [
100    inPattern
101    (isDefined(variables) ?
102      "using" "{" [
103       variables
104      ] "}"
105    )
106    sim
107   ] "}"
108  ;
109
110 template SelEqual context
111  :  "sel" name "(" arguments{separator = ","} ")"
112   (isDefined(ATLLibraries) ? "ATLLibraries" "{" [ATLLibraries {separator = ","} ] "}")
```

```
113     (isDefined(javaLibraries) ? "JavaLibraries" "{" [javaLibraries {separator = ","} ] "}"
            ↪)
114     "{" [
115      inPattern
116      (isDefined(variables) ?
117        "using" "{" [
118         variables
119        ] "}"
120      )
121     ] "}"
122    ;
123
124   template ExternalMethod context
125    : "uses" name "[" inMappingModel{separator = ","} "]" "(" arguments{separator = ","} ")
          ↪"
126      (isDefined(ATLLibraries) ? "ATLLibraries" "{" [ATLLibraries {separator = ","} ] "}")
127      (isDefined(javaLibraries) ? "JavaLibraries" "{" [javaLibraries {separator = ","} ] "}
          ↪")
128    ;
129
130   template LibraryRef abstract ;
131
132   template ATLLibraryRef
133    : "(" "name" "=" name{as = stringSymbol} "," "path" "=" path{as = stringSymbol} ")"
134    ;
135
136   template JavaLibraryRef
137    : "(" "name" "=" name{as = stringSymbol} "," "path" "=" path{as = stringSymbol} ")"
138    ;
139
140   template InPattern
141    : (isDefined(elements) ?
142       "from" [
143        elements{separator = ","}
144       ]
145     )
146      (isDefined(filter) ?
147       "when" [
148        filter
149        ]
150      )
151    ;
152
153   template EqualInPattern
154    : "leftType" ":" leftElement
155      "rightType" ":" rightElement
156    ;
157
158   template InPatternElement abstract addToContext;
159
160   template SimpleInPatternElement
161    : varName ":" type
162     (isDefined(models) ? "in" models{separator = ",", refersTo = name, lookIn = #all})
163    ;
164
165   template Sim
166    : "is" 'value'
167    ;
168
169   -- @end Matching methods
170
171   -- @begin ModelsFlow block
172
173   template ModelsFlowsBlock
174    : "modelsFlow" "{" [
175     modelsFlows
176      ] "}"
177    ;
```

```
178
179 template ModelFlowExpression abstract;
180
181 template WeightedModelExp
182  : weight ":" modelFlowExp;
183
184 template MethodCall
185  : (isDefined(outMappingModel) ? [outMappingModel "="])
186     method{refersTo = name, lookIn = #all, autoCreate = ifmissing, createIn = '#context
        ↪'.methods}
187     "[" (isDefined(inMappingModel) ? inMappingModel{separator = ","}) "]"
188    (isDefined(arguments) ? "(" arguments{separator = ","} ")" )
189  ;
190
191 template ModelRefExp
192  : referredModel{refersTo = name}
193  ;
194
195 template MappingModelRefExp
196  : referredMappingModel{refersTo = name}
197  ;
198
199 -- @end ModelsFlow block
200
201 -- @begin OCL
202
203 template OclExpression abstract operatored;
204
205 template ThisModuleExp
206  : "thisModule"
207  ;
208
209 template ThisNodeExp abstract;
210
211 template ThisRightExp
212  : "thisRight"
213  ;
214
215 template ThisLeftExp
216  : "thisLeft"
217  ;
218
219 template ThisEqualExp
220  :  "thisEqual"
221  ;
222
223 template ThisWeightExp
224  :  "thisWeight"
225  ;
226
227    template ThisSimExp
228  :  "thisSim"
229  ;
230
231 template ThisInstancesExp
232  : "thisInstances" "(" instancesOp ")"
233  ;
234
235 template SummationExp
236  : "Summation" "(" sumExpression ")"
237  ;
238
239 template ThisEqualModelExp
240  : "thisEqualModel"
241  ;
242
243 -- @end OCL
```

Appendix C: An M1-to-M1 matching algorithm for AST models

Listing 3: AML algorithm matching AST models

```
strategy JDTAST {

uses JavaASTDifferentiation[IN1:EqualModel(m1:JavaAST, m2:JavaAST)]()

create CMethodD () {
 leftType : MethodDeclaration
 rightType : MethodDeclaration
 when
  true
}

create CName () {
 leftType : Name
 rightType : Name
 when
 thisLeft.refImmediateComposite().oclIsTypeOf(JavaAST!Type) and
 thisRight.refImmediateComposite().oclIsTypeOf(JavaAST!Type)
}

create CType () {
 leftType : SimpleType
 rightType : SimpleType
 when
 thisLeft.refImmediateComposite().oclIsTypeOf(JavaAST!MethodDeclaration) and
 thisRight.refImmediateComposite().oclIsTypeOf(JavaAST!MethodDeclaration)
}

create CSingleVD () {
 leftType : SingleVariableDeclaration
 rightType : SingleVariableDeclaration
 when
 thisLeft.refImmediateComposite().oclIsTypeOf(JavaAST!MethodDeclaration) and
 thisRight.refImmediateComposite().oclIsTypeOf(JavaAST!MethodDeclaration)
}

create CSimpleName () {
 leftType : SimpleName
 rightType : SimpleName
 when
 thisLeft.refImmediateComposite().oclIsTypeOf(JavaAST!SingleVariableDeclaration) and
 thisRight.refImmediateComposite().oclIsTypeOf(JavaAST!SingleVariableDeclaration)
 or
 thisLeft.refImmediateComposite().oclIsTypeOf(JavaAST!MethodDeclaration) and
 thisRight.refImmediateComposite().oclIsTypeOf(JavaAST!MethodDeclaration)
}

sim SName ()
ATLLibraries{
```

```
 (name='Strings', path='../AMLLibrary/ATL/Helpers')
}
JavaLibraries{
 (name='match.SimmetricsSimilarity', path='../AMLLibrary/Jars/simmetrics.jar')
}
{
 is thisLeft.fullyQualifiedName.simStrings(thisRight.fullyQualifiedName)
}

sim SType (IN1:EqualModel (m1:JavaAST, m2:JavaAST)) {
 is thisModule.sim(thisLeft.name, thisRight.name)
}

sim SSimpleName ()
ATLLibraries{
 (name='Strings', path='../AMLLibrary/ATL/Helpers')
}
JavaLibraries{
 (name='match.SimmetricsSimilarity', path='../AMLLibrary/Jars/simmetrics.jar')
}
{
 is thisLeft.fullyQualifiedName.simStrings(thisRight.fullyQualifiedName)
}


sim SSingleVDName (IN1:EqualModel (m1:JavaAST, m2:JavaAST)) {
 is thisModule.sim(thisLeft.name, thisRight.name)
}

sim SSingleVDType(IN1:EqualModel (m1:JavaAST, m2:JavaAST)) {
 is thisModule.sim(thisLeft.type, thisRight.type)
}

sim SMethodDName (IN1:EqualModel (m1:JavaAST, m2:JavaAST)) {
 is thisModule.sim(thisLeft.name, thisRight.name)
}

sim SMethodDParameters (IN1:EqualModel (m1:JavaAST, m2:JavaAST)) {
 is thisModule.averSimSets(thisLeft.parameters, thisRight.parameters)
}

models {

 map : EqualModel (m1:JavaAST, m2:JavaAST)

}

modelsFlow {

 cSN = CSimpleName[map]
 sSN = SSimpleName[cSN]

 cT = CType[map]
 sN = SName[CName[map]]
 sT = SType[cT](sN)

 cSVD = CSingleVD[map]
 sSVDN = SSingleVDName[cSVD](sSN)
 sSVDT = SSingleVDType[cSVD](sT)

 wSVD = WeightedAverage[0.5:sSVDN, 0.5:sSVDT]
 tSVD = Threshold[wSVD]

 cMD = CMethodD[map]
 sMDP = SMethodDParameters[cMD](tSVD)
 sMDN = SMethodDName[cMD](sSN)

 wMD = WeightedAverage[0.5:sMDP, 0.5:sMDN]
```

```
117  tMD = Threshold[wMD]
118
119  d = JavaASTDifferentiation[tMD]
120
121 }
122 }
```

Appendix D: AML Web resources

A set of key Web resources about AML is listed below.

Wiki `http://wiki.eclipse.org/AML`
The AML wiki introduces the tool and explains its installation process. Note that AML has been contributed to Eclipse.org. Thus, interested users can get access to the AML source code for free.

AML use cases **Model co-evolution.** `http://www.eclipse.org/m2m/atl/usecases/ModelAdaptation/`

AML demo Interested readers may want to see a demo showing the functionalities of each AML component. The demo is available at `http://www.eclipse.org/m2m/atl/usecases/ModelAdaptation/AMLEdited_1024x720.htm`

Appendix E: AML Positioning

This appendix positions AML with respect to the criteria given in Section 2.4, i.e., input, output, matching algorithm blocks, and evaluation. For each criteria, we summarize what AML currently does.

Input

Criteria		AML
M2-to-M2	SQL-DDL	
	XSD	✔
	OWL	✔
	MOF (XMI)	✔
	Ecore	✔
	Others	KM3
M1-to-M1		✔

Table 1: Positioning AML with respect to the input criterion

The AML metamodel importer (see Section 4.3.1.4) translates inputs from different technical spaces (i.e., OWL, MOF) to Ecore or KM3 (the internal AML formats). Thus, it is possible to apply AML M2-to-M2 matching transformations to models built in other technical spaces. The experimentation explained in Section 5.5 however revealed that EMFTriple [24] (i.e., the AmmA-based tool used to translate OWL ontologies into Ecore metamodels) needs a few improvements to fully support the translation, above all, when ontology individuals are involved.

Section 3.3 shows GeromeSuite as the approach supporting the largest spectrum of technical spaces. AML supports a large number of technical spaces as well. Moreover, it has an advantage over GeromeSuite; unlike GeromeSuite, AML uses a standard to represent models, i.e., Ecore, which allows the manipulation of large models (around 250000 elements [131]). Therefore, by using Ecore, AML promotes its integration with recent EMF-based tools and even with early frameworks which make efforts in closing the gap between them and EMF.

The comparison of Table. 3.1 to Table. 1 shows that AML is the only approach allowing heuristics reusable in M2-to-M2 and M1-to-M1 matching algorithms. The implementation of several M2-to-M2 algorithms and an M1-to-M1 program supports this affirmation. EMF Compare has reusable heuristics too, however they match models conforming to same metamodel. In AML, $LeftModel$ and $RightModel$ can conform to differing metamodels.

Output

Criteria		AML
Similarity value	Discrete	
	Continuous	✔
Notation	Endogenous	✔
	Exogeneous	
Cardinality	1:1	✔
	1:n	✔
	m:1	✔
	m:n	✔
App. domain relationships		✔

Table 2: Positioning AML with respect to the output criterion

AML inherits the genericity of AMW [2] to represent simple and complex mappings associated to several application domains. The current AML matching transformations however only yield simple mappings; there is not a notation for explicit translation of simple mappings into complex. Although there exist this limitation, AML remains applicable.

Matching algorithm blocks

Criteria		AML
Normalization	Label-based	✔
	Structure-based	
Searching	Cartesian product	✔
	Fragment-based	✔
Similarity computation		✔
Aggregation		✔
Selection		✔
Iteration	Manual	✔
	Automatic	
Mapping manipulation		Diff, Co-evolution, Model Synchronization
User Involvement	Initial mappings	✔
	constraints	✔
	Additional inputs, e.g, mismatches, synonyms	✔
	Combination of heuristics	T
	Mapping refinement	G

Table 3: Positioning AML with respect to the matching building blocks criterion

Normalization *Label-based.* The AML library provides tokenizers that normalize the morphology of labels (see Section 4.4.4.

Structure-based. Unlike other MDE approaches applying this kind of normalization [6][88][84], we keep models conforming to their original metamodels. Section 4.2.6 described an experimentation supporting this choice. According to our experimentations, the translation of models into graphs or trees has the following disadvantages: 1) imposes an extra step to matching algorithms, 2) impacts their performance (the algorithms process verbose data structures) and development (the code navigating the data structure becomes complex).

Searching AML algorithms perform a fragment-based searching. AML provides a construct devoted to it, i.e., `create`. This construct allows the declaration of types representing $LeftModel$ and $RightModel$ elements.

For M2-to-M2 matching, the AML library provides pre-defined matching transformations which focus on certain types, e.g., `Class`, `Relation`, etc.

For M1-to-M1 matching, AML lets developers explicitly specify `create` matching transformations with their respective types.

Similarity The AML library contains 4 linguistic-based, 2 constraint-based, 3 instance-based, and 2 structure-level techniques (see Section 4.4.4 for more details).

Currently, the number of AML similarity matching transformations do not overcome the contributed one in early work (in particular Coma++ [62]). The rationale is that the thesis mainly investigated the efficiency that techniques, used in other technical spaces, have in MDE. Thus, we have implemented only the techniques that related work reports like good (e.g., *Similarity Flooding* [77][3]) or techniques exploiting Web resources (e.g., Google MSR [96]).

Aggregation and Selection AML only provides 1 aggregation and 3 selection techniques. These techniques (along with the thresholds) have been borrowed from Coma++ [62]. It would be interesting to test other techniques and thresholds, for example, constraint-based selection techniques.

Iteration The current version of AML allows manual iteration. AML can be extended to incorporate an automatic iteration construct. It is necessary to add a `for` or/and `while` construct into the AML concrete syntax, and modify the compiler to translate the construct into a `for` Ant task. To support simple (e.g., $n < 5$) and complex (e.g., OCL expressions) iteration conditions, the `for` Ant task has to be extended too.

Mapping manipulation AML manipulates mappings by means of user-defined matching transformations and HOTs. The output criteria elaborates on the first mean, we now refer to the second one. The AML library provides the `HOT_match` transformation that translates M2-to-M2 simple mappings into ATL code. To support the translation of complex mappings into ATL code, it is necessary to develop a new HOT superimposing `HOT_match` (Section 6.1 illustrates that). Since a good knowledge of ATL is required, we believe that it is hard to implement HOTs. Future research has to practice Tisi's guidelines about HOT development [9].

User involvement The user can provide initial mappings to AML programs. The `aggr` construct explicitly allows users to associate weights to matching transformation results as well. By using parameter models, the user can select the dictionary (or resource) that a linguistic-based similarity heuristic may need. Thus, AML algorithms can take a large spectrum of inputs provided by the user. The only constraint is that inputs have to be models or XML files (from which models can be extracted).

The user specifies the combination of AML matching transformations in a textual manner. They can manually refine mapping models by using the AMW editor [2].

Evaluation

Criteria		AML
Dataset (Metamodels)	Number of pairs	Metamodels: 68, Ontologies: 3
	Size	Largest metamodel: 865, Largest ontology: 127
	Domains	The ATL Zoo, The OAEI Conference track
	Exec. Time (min.)	Metamodels: ranges from 2 to 280 min, Ontologies: < 1 min
	Fscore	Metamodels: >=0.5 for 35% of experimentations, Ontologies: >=0.5 for 41% of experimentations
Determination of reference alignments	Experts	✔
	Existing software artifacts	✔

Table 4: Positioning AML with respect to the evaluation criterion

We have applied AML algorithms to datasets from diverse domains, sizes, and formats. Table. 4 gives an idea about the accuracy and performance obtained by the algorithms. Finally, the table shows that AML uses expected mappings defined from scratch or extracted from existing software artifacts (i.e., model transformations).

Une approche pour l'adaptation et l'évaluation de stratégies génériques d'alignement de modèles

Kelly Johany Garcés-Pernett

Mots-clés: Génie des Modèles, Transformation de modèles, Alignement de modèles.

L'alignement de modèles est devenu un sujet d'intérêt pour la communauté de l'Ingénierie Dirigée par les Modèles. Le but est d'identifier des correspondances entre les éléments de deux métamodèles ou de deux modèles. Un scénario d'application important est la dérivation des transformations à partir des correspondances entre métamodèles. De plus, les correspondances entre modèles offrent un grand potentiel pour adresser d'autres besoins. L'établissement manuel de ces correspondances sur des (méta)modèles de grande taille demande une grande quantité de travail et est source d'erreurs. La communauté travaille donc à automatiser le processus en proposant plusieurs stratégies d'alignement formulées comme la combinaison d'un ensemble d'heuristiques. Un premier problème est alors que ces heuristiques sont limitées à certains formalismes de représentation au lieu d'être réutilisables. Un second problème réside dans la difficulté à évaluer systématiquement la qualité des stratégies. Cette thèse propose une approche pour résoudre les problèmes ci-dessus. Cette approche développe des stratégies dont les heuristiques sont faiblement couplées aux formalismes. Elle extrait un jeu de tests d'usage à partir d'un répertoire de modèles et elle utilise finalement un mégamodèle pour automatiser l'évaluation. Pour valider cette approche, nous développons le langage dédié AML construit sur la plateforme AmmA. Nous contribuons à la définition d'une bibliothèque d'heuristiques et de stratégies AML. Pour montrer que notre approche n'est pas limitée au domaine de l'IDM nous testons celle-ci dans le domaine des ontologies. Finalement, nous proposons trois cas d'étude attestant l'applicabilité des stratégies AML dans les domaines de la coévolution des modèles, de l'évaluation des métamodèles pivots et de la synchronisation des modèles.

Adaptation and evaluation of generic model matching strategies

Kelly Johany Garcés-Pernett

Keywords: Model-Driven Engineering, Model transformation, Model matching.

Model matching is gaining importance in Model-Driven Engineering (MDE). The goal of model matching is to identify correspondences between the elements of two metamodels or two models. One of the main application scenarios is the derivation of model transformations from metamodel correspondences. Model correspondences, in turn, offer a potential to address other MDE needs. Manually finding of correspondences is labor intensive and error-prone when (meta)models are large. To automate the process, research community proposes matching strategies combining multiple heuristics. A problem is that the heuristics are limited to certain representation formalisms instead of being reusable. Another problem is the difficulty to systematically evaluate the quality of matching strategies. This work contributes an approach to deal with the mentioned issues. To promote reusability, the approach consists of strategies whose heuristics are loosely coupled to a given formalism. To systematize model matching evaluation, the approach automatically extracts a large set of modeling test cases from model repositories, and uses megamodels to guide strategy execution. We have validated the approach by developing the AML domain specific language on top of the AmmA platform. By using AML, we have implemented a library of strategies and heuristics. To demonstrate that our approach goes beyond the modeling context, we have tested our strategies on ontology test cases as well. At last, we have contributed three use cases that show the applicability of (meta)model matching to interesting MDE topics: model co-evolution, pivot metamodel evaluation, and model synchronization.