Type Inference in Flexible Model-Driven Engineering

Athanasios Zolotas

Doctor of Engineering

University of York
Computer Science

September 2016

Abstract

Model-driven Engineering (MDE) is an approach to software development that promises increased productivity and product quality. Domain *models* that conform to *metamodels*, both of which are the core artefacts in MDE approaches, are manipulated to perform different development processes using specific MDE tools. However, domain experts, who have detailed domain knowledge, typically lack the technical expertise to transfer this knowledge using MDE tools. Flexible or bottom-up Model-driven Engineering is an emerging approach to domain and systems modelling that tackles this challenge by promoting the use of simple drawing tools to increase the involvement of domain experts in MDE processes. In this approach, no metamodel is created upfront but instead the process starts with the definition of example models that will be used to infer a draft metamodel. When complete knowledge of the domain is acquired, a final metamodel is devised and a transition to traditional MDE approaches is possible. However, the lack of a metamodel that encodes the semantics of conforming models and of tools that impose these semantics bears some drawbacks, among others that of having models with nodes that are unintentionally left untyped. In this thesis we propose the use of approaches that use algorithms from three different research areas, that of classification algorithms, constraint programming and graph similarity to help with the type inference of such untyped nodes. We perform an evaluation of the proposed approaches in a number of randomly generated example models from 10 different domains with results suggesting that the approaches could be used for type inference both in an automatic or a semi-automatic style.

For my parents Despoina and Michalis

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Acknowledgements

Before everyone else, I would like to express my endless gratitude to my supervisors, Professor Richard Paige and Dr. Nicholas Matragkas for their incredible support and guidance through the years this project lasted. I would like to thank them for giving me the opportunity to travel to many conferences where I met with members of the research community and I expanded my research horizons. I am also very grateful to my assessor, Dr. Radu Calinescu, for providing me with advice and feedback. I address my acknowledgements to my industrial supervisor, Mr. Chris Wedgwood, for sponsoring parts of this research project and for allowing me to drive my research in the direction I was most interested in.

A big thanks and appreciation goes to my beloved parents, Michalis and Despoina, and my sister, Dimitra, for the incredible support and endless love they give me every day. They are always present to figure out any problems arise though-out my whole life.

Coping with domain-specific difficulties could not be possible without the help of four colleagues, Dr. Dimitrios Kolovos, Dr. Sam Devlin, Dr. Robert Clarisó and Dr. James Williams. Dimitris' continuous support on the Muddles and Epsilon implementation, Sam's and Robert's selfless guidance on the machine learning and constraint programming insights, respectively, and James' advice on MDE, continuous motivation and support were priceless.

Of course, I would like to thank the people I lived with these years, Kyriakos Efthymiadis and Simos Gerasimou, for making my daily life a nice and smooth journey. Special thanks to my friends in York, Tina Alyssandraki, Chris Evripidou, Liana Kafetzopoulou, Theo Karapanagiotidis, Maria Kechagia, Ioannis Kontopoulos, Anna Ladi, Kelly Pentaraki and Tasos Tsompanidis with whom I spent some of the most beautiful and remarkable moments in my life. I must thank the friends I left behind in Greece, Kostas Nikos, Konstantina Psarra, Apostolos Stefanidis and Thanos Stefanidis who gave me good advice and support in the years this project lasted. Finally, I would like to thank my colleagues in the Enterprise Systems group, Ran Wei, Adolfo Sanchez-Barbudo Herrera, Horacio Hoyos Rodriguez, Louis Rose, Babajide Ogunyomi and Kostas Barmpis for their daily support.

None of this would be possible if the above didn't believe in me and give me opportunities, guidance, support and help to become a better person.

Declaration

I declare that the work presented in this thesis is my own, except where stated. Chapters 3 and Chapter 4 present collaborative work and clearly state the contributions of this author and the collaborators. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. Parts of this thesis have been previously published in the following research papers:

[15] **Assigning Semantics to Graphical Concrete Syntaxes.** Athanasios Zolotas, Dimitris S. Kolovos, Nicholas Drivalos Matragkas, and Richard F. Paige. *Proceedings of the Third International Workshop on Extreme Modelling (XM 2014)*, Valencia, Spain, September 2014.

[16] **Type inference in flexible Model-driven Engineering.** Athanasios Zolotas, Nicholas Matragkas, Sam Devlin, Dimitrios S. Kolovos, and Richard F. Paige. *Proceedings of the Eleventh European Conference on Modelling Foundations and Applications*, L'Aquila, Italy, July 2015.

[17] **Type Inference Using Concrete Syntax Properties in Flexible Model-Driven Engineering.** Athanasios Zolotas, Nicholas Matragkas, Sam Devlin, Dimitrios S. Kolovos, and Richard F. Paige. *Proceedings of the First International Workshop on Flexible Model-Driven Engineering (FlexMDE 2015)*, Ottawa, Canada, September 2015

[18] **Flexible Modelling for Requirements Engineering.** Athanasios Zolotas, Nicholas Matragkas, Dimitrios S. Kolovos, and Richard F. Paige. *Proceedings of the First International Workshop on Flexible Model-Driven Engineering (FlexMDE 2015)*, Ottawa, Canada, September 2015.

[19] **Constraint Programming for Type Inference in Flexible Model-Driven Engineering**, Athanasios Zolotas, Robert Clarisó, Nicholas Matragkas, Dimitrios S. Kolovos, Richard F. Paige. *International Journal on Computer Languages, Systems and Structures (COMLAN)*, 2016

Parts of this thesis have been submitted for publication in the following and are **under review** at the moment this thesis is being written:

- **Type Inference in Flexible Model-Driven Engineering using Classification Algorithms**, Athanasios Zolotas, Nicholas Matragkas, Sam Devlin, Dimitrios S. Kolovos, Richard F. Paige. ***Submitted and Under Review*** *in the International Journal on Software and Systems Modelling (SoSyM).*

CHAPTER 1

Introduction

Model-Driven Engineering (MDE) is an approach to software engineering that promotes the abstraction from technical and implementation related details towards domain-related information [20, 21]. MDE treats *models* as first class artefacts in the software development process. These models represent a variety of views and store different levels of details of the system under development: from models that omit low level details to models that can be used to automatically generate the final working artefact. *Metamodels* are defined at the beginning of the MDE process and include the syntax and the semantics of the models that *conform* to them.

Software engineers, designers and other stakeholders are key factors in the success of an MDE process. Thus, the active collaboration among all of them is important; the participation of the stakeholders, which are usually the domain experts, is of interest as it can lead to better understanding of the domain and consequently to complete and valid products [5, 22–25]. Although MDE conceals low level implementation details, focusing on abstractions, the active participation of the domain experts is not always achieved.

In this chapter the identified gap that motivates this research work is discussed. This chapter outlines the research hypothesis and lists the research objectives. An overview of the research results and contributions is also included. Finally, the structure of the thesis is presented.

Test paragraph!

1.1. Motivation and Background

In the process of building the domain knowledge needed to define a system, a business process or a Domain-Specific Language (DSL) the participation of all the stakeholders is important [5, 22–25]. They hold the knowledge of the domain and they know the concepts that need to be modelled. However, not all the stakehold-

ers share the technical expertise to transfer their undoubtedly good understanding of the domain to the technical experts responsible for building the aforementioned artefacts. It is this difficulty in bridging the knowledge the domain and the engineering experts hold in their respective areas of expertise that is known as the *"symmetry of ignorance"* [26–28].

Sometimes models fail in being a correct and thorough representation of the system. This can be either a result of bad design or a result of the symmetry of ignorance: the tacit knowledge of the domain that the domain experts hold cannot be transferred to the system or language engineers effectively. Bottom-up or flexible MDE is a revolutionary MDE approach, introduced to facilitate the bridging of that gap by changing the sequence of steps followed in tradition MDE: models are defined at the beginning and are used to help modelling engineers infer the metamodel [5, 23, 29].

1.1.1. Bottom-up MDE

Conventional DSL definition processes start with the creation of a metamodel which is then used to instantiate models and guide the development of editors and other artefacts such as model-to-model and model-to-text transformations. Such a process implies expertise in metamodelling, and in relevant technologies. While this may be an easy or at least understandable process for MDE experts, this is not always the case with domain experts [5] who are more familiar with tools like simple drawing editors [22]. However, the involvement of domain experts is important in the definition of high quality and well-defined DSLs (i.e., those that cover all the needed aspects of a domain) [5, 24, 25, 30]. To address the aforementioned issue, flexible modelling approaches have been proposed in the literature (e.g., [4–6, 31]). Such approaches are based on sketching tools and do not require the definition of a metamodel during the initial phases of language engineering. The trade-off between formality and flexibility results in a better domain understanding by language engineers, and eventually to a higher quality language.

In flexible (or bottom-up) MDE, the process starts with the definition of example models [5, 23, 29]. These example models help language engineers to better understand the concepts of the envisioned DSL and can be used to infer *draft metamodels* manually or (semi-)automatically, which eventually leads to the definition of the final metamodel. In this fashion, a richer understanding of the domain can be developed *incrementally*, while concrete insights (e.g., type information) pertaining to the envisioned metamodel are discovered. When all the details of the domain are discovered and a final metamodel is developed, transition to traditional, rigorous MDE approaches and tools is possible. Figure 1.1 depicts the stages taking place in a typical flexible MDE process as this is interpreted by studying different flexible MDE approaches in the literature (e.g., [5, 22, 32]).

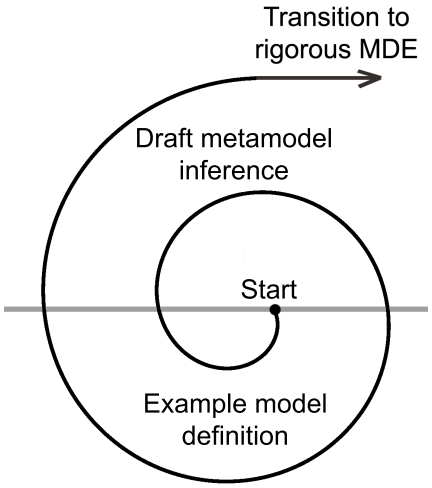


Figure 1.1.: Stages of a typical flexible MDE approach.

The sketching tools used in flexible MDE processes, allow the quick definition of exemplar models sacrificing the formality that model editors, which are based on a rigorously-defined metamodels, offer. In addition, drawing tools do not require MDE-specific expertise. The elements (nodes and edges) of these flexible example models can have type annotations assigned to them to describe the domain concept they represent and can also be amenable to programmatic model management using MDE suites like Epsilon [33].

On the other hand, since sketching tools cannot enforce syntactic and semantic correctness rules, flexible models are prone to various types of errors [16]:

1. *User input errors:* elements that should share the same type have different types assigned to them as a result of a typo on any other mistake.
2. *Changes due to evolution:* elements representing concepts that have evolved during the domain exploration process do not have their types updated.
3. *Inconsistencies due to collaboration:* when multiple domain experts collaborate in the definition of the models, multiple types representing the same concept can be introduced.
4. *Omissions:* elements can be left untyped especially when models become large as it is easier to overlook some of the elements.

The existence of such errors is an obstacle in the smooth execution of flexible MDE approaches; for example an untyped element (error #4 above) is ignored. This is an undesirable effect as the exploration of the domain and the identification of all the concepts it includes is a basic aim in flexible MDE approaches. Deploying efficient techniques that will prevent this from happening is of interest to improve the effectiveness of flexible MDE approaches. There are at least two ways for achieving that. The first includes the creation of model managements scripts which will

identify the untyped elements and request the engineers to provide their type. A second, could be that of *type inference*, where the types could be inferred and filled in either automatically or in a semi-automatic way.

1.2. Hypothesis and Objectives

The research hypothesis of this thesis is stated as follows:

> *It is feasible to use classification algorithms, constraint programming and graph similarity techniques in the two phases of flexible MDE approaches to accurately suggest the most appropriate type for each untyped node of a model or reduce the set of possible types an untyped node can have. This can reduce the effort needed to produce complete example models from which metamodels can be inferred.*

The characteristics extracted from the research hypothesis that construct the context of this research project are listed below:

1. **Feasible:** the techniques should be incorporated into the usual DSL engineering workflow, i.e. it does not take more than a few minutes to find the types.
2. **Two phases of flexible MDE:** flexible MDE processes are iterative and consist from the two stages, depicted in Figure 1.1, the aim of which is an incremental built of knowledge in the domain. The proposed solutions could be applied in both of these phases, where a metamodel is or is not already inferred, tackling the same problem, that of type inference.
3. **Flexible MDE approaches:** the proposed solutions should be applied in a specific flexible MDE approach (i.e., Muddles [4]) and could be in principle adopted into all the available approaches (i.e., those presented in Section 2.2).
4. **Type suggestion/Reduce set of possible types:** using different characteristics available in flexible models, the approaches should be able to suggest possible types for the elements left untyped. These characteristics can be the concrete and the abstract syntax, the labels of the features of the models (i.e., references and attributes) or constraints imposed by the draft metamodel (e.g., multiplicities). In addition, the proposed approaches should predict the correct type of untyped nodes with *at least* higher accuracy than a random prediction and provide a set of candidate types for each node, *at least* smaller than the total number of the already defined types.
5. **Nodes left untyped:** this thesis focuses on adopting or introducing techniques that tackle the problem of nodes' type omission errors (i.e., error #4 in Section 1.1.1) in flexible models.

1.2.1. Thesis Objectives

The research objectives of this thesis are to:

- Facilitate the incremental acquisition of domain knowledge that flexible MDE approaches offer by inferring the type of the untyped nodes in example models. This should be achieved by:
  - Identifying existing research techniques that can be used for type inference in flexible MDE.
  - Proposing new algorithms that could be used in the same direction.
- Develop the artefacts to import the aforementioned techniques and algorithms in the domain of type inference in flexible MDE.
- Evaluate the performance of the proposed approaches and the level of assistance they offer to language engineers.

1.3. Research Contributions

This thesis proposes three novel approaches for tackling the problem of type omissions in flexible MDE. More specifically, implementations of approaches from three different domains (i.e., machine learning, logic programming and graph similarity) are adapted in the flexible MDE domain. To the best of our knowledge these approaches are the first introduced in addressing the problem identified.

This thesis also proposes the classification of errors observed in the process of defining example models into four categories, that of *typos*, *type omissions*, *errors due to language evolution* and *errors due to collaboration*. A review of the available flexible or bottom-up methodologies and tools was also performed as part of the literature review for this research project. This thesis also contributes to the extension of a flexible modelling technique, called *Muddles*, which is part of the Epsilon MDE suite [33] built on top the Eclipse Modelling Framework (EMF) [34] and the GraphML file format [35].

The validation of the research hypothesis has been confirmed by evaluating the proposed approaches for a number of randomly generated flexible models by defining new metrics of success in this domain. The results and the proposed metrics can be used as benchmarks for future research in the domain. The evaluation also simulated a variety of scenarios to identify factors that affect (positively or negatively) the type inference, extracting useful guidelines for language engineers working on flexible MDE.

An outline of the research is shown in Figure 1.2.

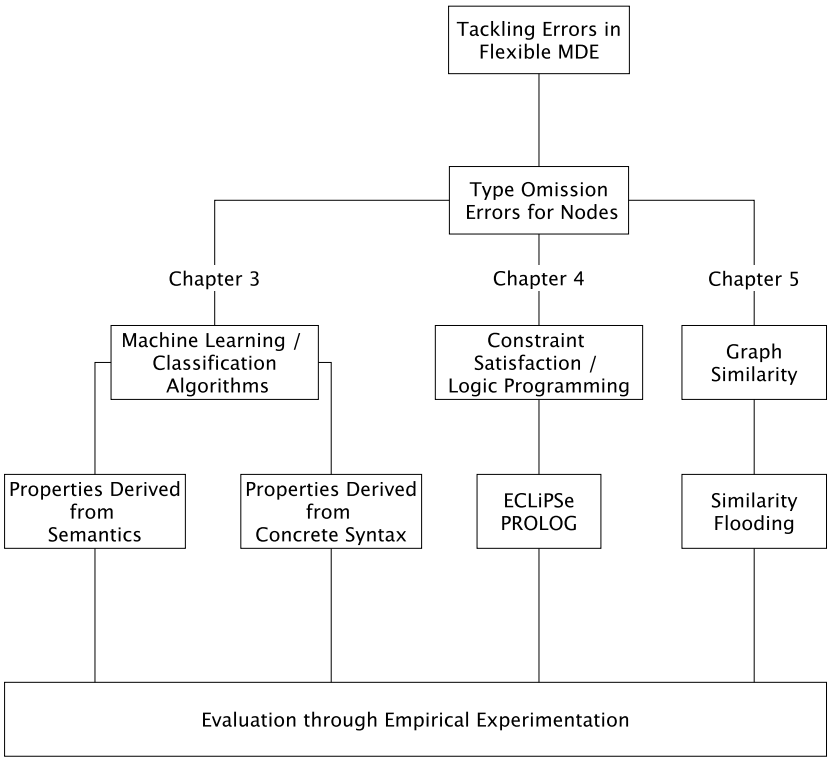


Figure 1.2.: Overview of the research project.

1.4. Thesis Structure

Chapter 2 presents the literature related to the concepts underlying this research project. More specifically, Section 2.1 introduces the reader to the principles and practices of MDE. Section 2.2 presents a systematic review of the flexible or bottom-up MDE approaches available, including the Muddles [4] used in the evaluation of the research. The reviewed approaches are categorised using different criteria, highlighting their advantages and disadvantages. Approaches related to completion of partial models are discussed in Section 2.3. In Section 2.4, the type and meta-model inference approaches reviewed are presented. A comparison and a critique of these approaches are given in Section 2.5. Sections 2.6, 2.7 and 2.8 briefly discuss the available techniques and methodologies in the domains of classification algorithms, logic programming and graph similarity respectively. These include the Classification and Regression Trees (CART), Random Forests and Support Vector Machines (SVM) in the classification domain; the PROLOG general purpose logic programming language and its implementations (e.g., SWI-Prolog) in the logic programming domain; and Similarity Flooding algorithm in the graph similarity domain.

Chapter 3 presents the first approach for tackling the problem of untyped nodes in flexible modelling. More specifically, the use of Classification and Regression

Trees (CART) is presented (Section 3.2). The two variations proposed, one based on *semantics* and the second based on *graphical* attributes of the flexible models, are explained. For both variations, the elements of the flexible models need to be encoded into a set a features, called *feature signatures*, in order to be fed into the classification algorithm. The features selected and the way they are encoded are presented in Section 3.3. Section 3.4 describes the training and classification process in detail. In Section 3.5, the empirical evaluation for both the feature signatures sets is presented. Finally, the prerequisites and the limitations of the specific approach are discussed (Section 3.6).

Chapter 4 presents the second approach for type inference in bottom-up MDE proposed in this research project. An overview of the approach is given in Section 4.2, followed by the presentation of the algorithm that represents the constrained problem that needs to be solved by the logic programming solver (Section 4.3.1). More specifically, each flexible model is represented as a constraint satisfaction problem which the algorithm solves in order to suggest the possible types for the untyped nodes. The experimental evaluation is presented in Section 4.4 while the limitations and prerequisites of the approach and the differences from the previous (CART) one are discussed in Section 4.5.

Chapter 5 presents the third approach, that of inferring the type of the nodes based on similarities in the labels of their features (attributes and references). The representation of the flexible model in a format, called RDF [36], that can be used by the similarity flooding algorithm is presented in Section 5.3. Section 5.5 presents the empirical evaluation for the approach. The limitations and the prerequisites of the approach, and the comparison with the other two approaches are also discussed (Section 5.6).

Chapter 6 concludes the thesis by reminding the research hypothesis and summarises the three proposed solutions for tackling the identified gap (Section 6.1). The evaluation results are also discussed. Finally, guidelines for future work and advancements are proposed (Section 6.2).

CHAPTER 2

Literature Review

This chapter gives the background to MDE and an overview of the key principles, practices and tools used in the domain. The benefits and the weaknesses of using MDE are also discussed. Flexible MDE approaches and metamodel and type inference literature are also presented followed by a critique on their advantages and disadvantages.

The chapter is structured as follows. Section 2.1 describes the background of MDE. The underlying principles and the tools used in MDE to define, instantiate and manipulate models are presented. In Section 2.2 we present flexible MDE approaches found in the literature. In the same section we analyse the flexible MDE approach used as proof of concept in this thesis, named *Muddles*. Related work on partial modelling and metamodel inference is also discussed. Sections 2.6, 2.7 and 2.8 give the background for the three different domains from which algorithms and techniques were adopted to provide solutions in tackling the identified research gap. More specifically, Section 2.6 presents an overview of classification algorithms. In Section 2.7 we give the background for logic programming and constraint satisfaction problems. Finally, Section 2.8 presents graph similarity, focusing on the similarity flooding algorithm used in this work.

2.1. Model-Driven Engineering

Before entering the world of software engineering, *models* were, and still are, a key part of other domains like architecture and other areas of engineering. Models are used to represent an abstract view of an artefact helping engineers to turn their focus away from implementation details and reason about the artefact under development among the stakeholders [37].

Models in software engineering are very important; some argue they are more important than in any other discipline [37]. Software engineers and domain ex-

perts can focus in expressing the domain characteristics rather than focusing on implementation details, working in a more abstract level. The defined models can be used to generate systems automatically; When the requirements or the design change, the changes can be applied to the abstract models and using the same or refined generators, the changes can be propagated to the system automatically.

The following section presents key MDE principles and tools used in MDE.

2.1.1. MDE Principles and Tools

Models are the main artefact in MDE. A generally accepted definition of a model is as follows:

> *"A model is a simplification of a system built with an intended goal in mind."* [1]

From the above, and other definitions, a common outcome is that a model describes an abstract view of a system. This description can be either textual or graphical [38]. Simplification is also important in an MDE lifecycle; A model should only include the important information, abstracted, having all the details not-of-interest pencilled out [1].

An exceptional example of a model is the *metamodel*. A metamodel - a model itself - includes the set of syntactic rules that will be used to define other models. These rules are also referred as the *concepts* and the *relations* between these concepts [39]. Thus, each model that fulfils all the rules defined in a metamodel and all of its concepts, *conforms to* the metamodel [40]. The relationships between the model and the domain it describes are shown in Figure 2.1.

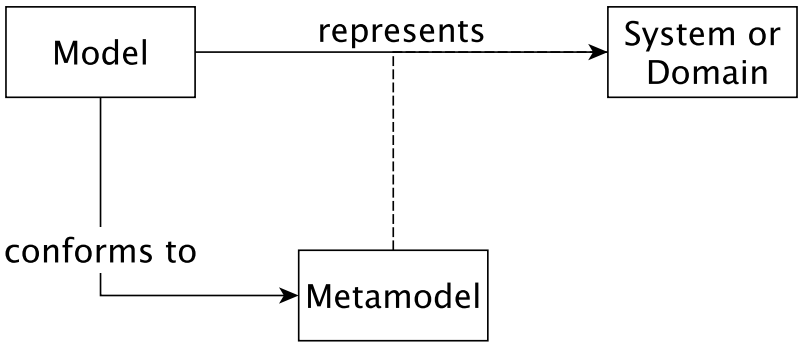


Figure 2.1.: The relationships between a model with its metamodel and the domain it represents (adapted from [1]).

It is not always the case that only metamodels can be used to define the rules of models; other kind of schemas could also be used. In any form, a modelling language, should typically include the following three constraints:

Semantics: Are used to describe the meaning of the concepts and the relationships between them in the domain the metamodel is applied to. For example,

if we take into account two languages, one that defines websites and one that describes tailoring procedures, they may both have a "button" concept but this will most probably represent two unrelated things. The semantics of a modelling language specify this.

Abstract Syntax: Contains all the concepts that appear in the domain and are of interest. For example, in a modelling language that defines websites the abstract syntax may consists of concepts like "Page", "Button", "Image", etc.

Concrete Syntax: Describes the notation of the concepts; how these concepts may appear when the model is created. In textual languages this is the keywords of the languages (e.g., in a websites' definition language this might be keywords like "img" that represents the "Image" concept, etc.) In graphical languages this is the graphical representation of each concept in a model (e.g., a button might be represented by a rectangular box, while the connection between it and the page it leads to, as a line with an arrow at the ending point)

There is no definitive rule for separating modelling language; some classifications are proposed though. The first classifies them into two categories, *domain-specific* and *general-purpose* languages.

Domain-specific: Domain-Specific languages (DSLs) contain languages such as WebML [41] for describing web applications or SQL [42] for manipulation of databases. DSLs are built to be used in a specific domain and they promote productivity in favour of portability [43]. In addition the use of DSLs allows better understanding of the written code, especially by experts of the domain and better communication between them [44]. DSLs that are used for modelling purposes are known as Domain-Specific Modelling Languages (DSMLs).

General-purpose: General-purpose languages capture a wide variety of domains. Java [45] or BPMN [46] are examples of general-purpose languages as they are used to describe multiple domains without adapting their syntax and semantics. UML [47] is an example of a general-purpose modelling language. General-purpose languages are addressed to a broader audience and they promote portability and maintainability in favour of productivity [43].

Another common classification of languages is based on the nature of concrete syntax; There are *textual*, *graphical (or visual)* and *hybrid* languages [48].

Textual: As the name suggests the commands of a textual language are expressed using words written in a text editor. Java [45] and HTML [49] are examples of textual languages. In fact, most of the programming languages are textual. Tools like Xtext [50] and EMFText [51] can be used to generate editors for textual languages based on MDE principles.

Graphical/Visual: In contrast, graphical languages have their commands expressed using shapes and icons. UML [47] is a characteristic example of a graphical language. Tools like the Graphical Modelling Project (GMP) [52] (i.e., the Graphical Modelling Framework (GMF) and Graphiti), the Sirius [53] framework that builds on-top the GMF, AToM^3 [54] and Eugenia [55] include the necessary infrastructure to generate editors for graphical languages.

Hybrid: Hybrid languages are those that combine both textual and graphical syntax. Each notation can be used to describe the system under development from different viewpoints [56].

MOF

The Object Management Group (OMG) has defined an architecture for creating modelling languages, called the *Meta Object Facility (MOF)* [3]. The architecture is given in Figure 2.2. The top level, $M3$, consists of metamodelling languages, i.e., languages that can be used to define metamodels like those appearing in $M2$ level. The UML Metamodel [47] is an example of a metamodel that is proposed by OMG to define models like Class or Activity diagrams that appear in the $M1$ level. At the bottom, the $M0$ level, consists of elements that represent the real world domain objects need to be modelled (e.g., a booking system or a video player, etc.). The layering architecture of MOF is an analogy of how computer programming languages are defined: at the top there is a language definition language (e.g., EBNF [57]), the second layer consists of a programming language (e.g., Java [45]), the third of a piece of code written in the aforementioned language (e.g., Book.java) and at the bottom the real-world objects [2].

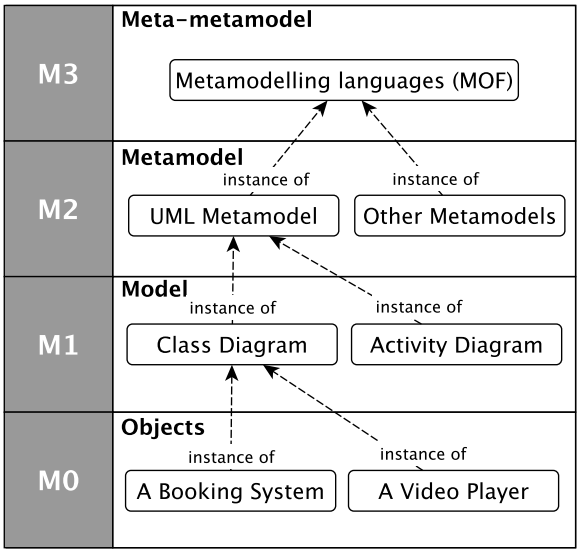


Figure 2.2.: The four layers of metamodelling infrastructures (adapted from [2] and [3]).

In MOF, models are stored using an OMG standard, the XML Metadata Interchange (XMI) [58, 59] which allows interoperability between the different MOF-based modelling suites. Unfortunately, in the MOF standard it is not possible to include constraints as part of the semantics of a metamodel, other than the structural constraints imposed by the metamodel itself. In MOF, the Object Constraint Language (OCL) [60] is used to express more complex constraints.

An example of a metamodel defined in MOF is presented in Figure 2.3. This metamodel creates a simple modelling language for expressing zoos. Each rectangle represents a concept of the language and is called a *meta-class*. The lines connecting meta-classes are called references. The references with a black diamond at their end are called *compositions* and declare that the existence of the contained class (e.g., Animal) relies on the existence of the parent class (e.g., Zoo): if the parent object is deleted then all the contained objects are also deleted. Each meta-class might also have some attributes (e.g., name, entranceFee, etc.) each of which is of a type (e.g., String, Integer, etc.). All the references and the attributes in MOF are collectively called *meta-features*. The numbers or asterisks ("*") at the end of each reference or composition are called *multiplicities* and define the minimum (*lower bound*) and maximum (*upper bound*) number of instances each object can be connected with (e.g., for the *treats* relationship, each Doctor can be connected with *many* Animals). If the minimum bound is set to 0, then this feature is *optional* as a non-instantiation of it is allowed by the metamodel. The meta-classes whose name is in italics define *abstract* meta-classes, that is the classes that cannot be instantiated in the model level. Usually, abstract classes are used to group common characteristics of the classes which *extend* them. The extension reference is expressed using a white filled arrowhead. Meta-features are *inherited* to the extension classes.

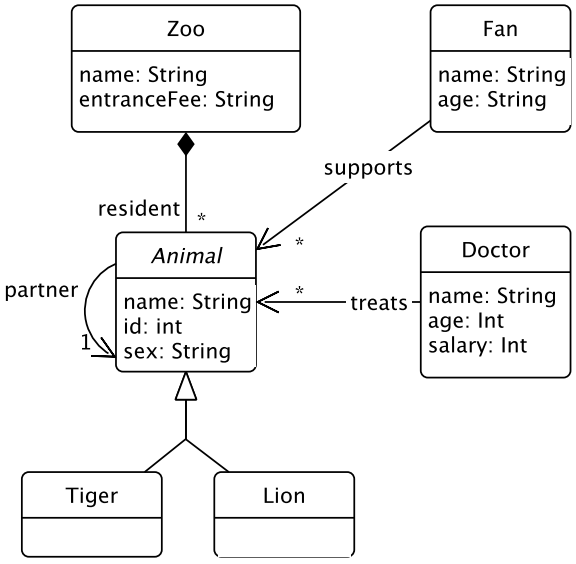


Figure 2.3.: An example of a metamodel.

Figure 2.4 shows an example model that is an instance of the metamodel given in Figure 2.3 expressed in the object diagram notation of UML [47]. Each element has a label which includes a column symbol (":") and the name of the meta-class this object is an instance of. For example, the "JurassicZoo" object is an instance of the "Zoo" meta-class. Each object also instantiates all its features (the references and the attributes) that are not optional. Although this is UML's way to instantiate a metamodel, this does not mean that in MDE all the metamodels are instantiated using this notation. The modellers are able to define their own concrete syntax, either a textual or a graphical one. For example, in this Zoo DSL a concrete syntax could possibly include icons of tigers and lions to the instantiated meta-classes "Tiger" and "Lion" respectively and not rectangles as is done in the object diagram.

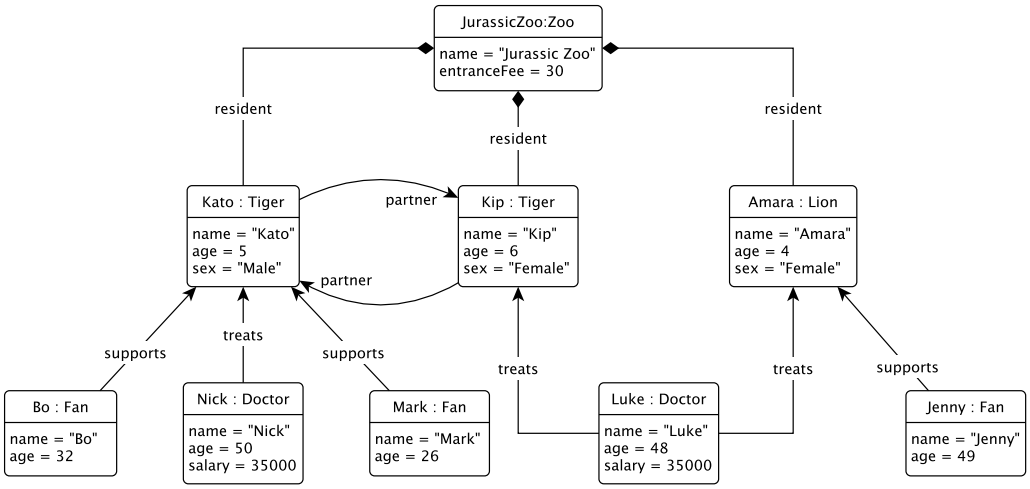


Figure 2.4.: An example of a model that conforms to the metamodel of Figure 2.3.

Beyond the four-layer that is used in this thesis, another proposed architecture is that of *multilevel modelling* also called *deep metamodelling* which enables modelling in different numbers of layers [61]. Such an architecture is argued to offer simplicity in model and metamodel description in some scenarios, examples of which are presented in [61].

Model Management

In MDE, models are manipulated to produce artefacts of interest. The most common model management activities are presented below.

Model Transformation: Model transformations are an important procedure in MDE and allows the definition of mappings between different artefacts [2]. There are three basic categories of model transformations: *model-to-model (M2M)*, *model-to-text (M2T)* and *text-to-model (T2M)*.

In M2M transformations the *source* model is transformed to another model which is the *target* model by executing a set of pre-defined *rules*. In a M2M

transformation there might be more than one source and more than one target models. The *multiplicity* of the transformation declares that number. Moreover, the *type of a* M2M transformation can be either endogenous or exogenous meaning that the source and the target model(s) conform to the same or different metamodels, respectively. An example of a M2M transformation is given in Figure 2.5.

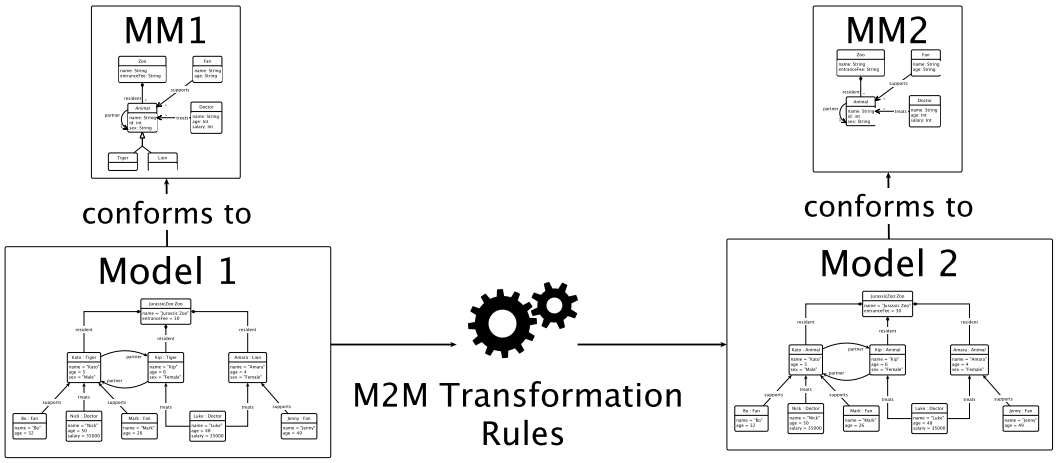


Figure 2.5.: An example of a model-to-model transformation between instances of two different metamodels.

In M2T transformations the source of the transformation is a model and the target is a piece of text (e.g., runnable code, documentation, etc.). An example is given in Figure 2.6. The opposite transformation (text-to-model) accepts as input a textual artefact and produces as output a model. Such a procedure is typical in reverse engineering operations [2].

There is a number of tools developed to perform model transformations. Among others the most widely used are the Epsilon Transformation Language [62], part of the Epsilon [33] suite, the Atlas Transformation Language (ATL) [63], the OMG's QVT [3] and VIATRA [64]. Languages like the Epsilon Generation Language (EGL) [65], Xpand [66,67] and Acceleo [68] can be used for model-to-text transformations specifically. Xtend [69,70], a Java-like language, is not a dedicated transformation language but it is used to perform both M2M and M2T transformations [48].

Model Querying: Models can be seen as a store of information. Thus, querying models to extract the data stored in them is important. For example, a model that conforms to a metamodel that describes zoos will most probably contain some instances of animals. In the event where the goal of the MDE procedure is the production of a list of all the animals of the zoo (i.e., a simple M2T transformation) then the model should be initially queried to get all the instances of the "Animal" meta-class before completing the transformation.

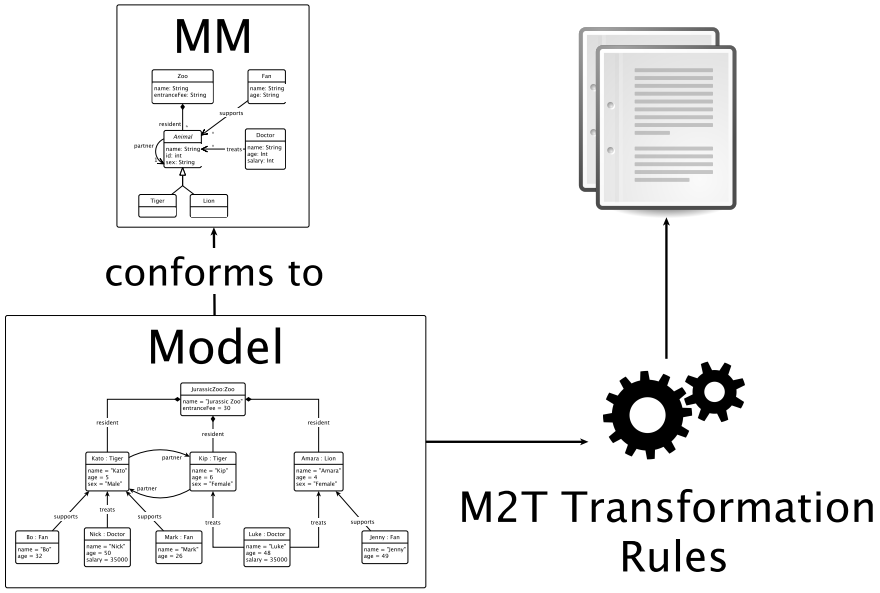


Figure 2.6.: An example of a model-to-text transformation.

Model querying is a core part of every model management process. The Epsilon Object Language (EOL) [38], OCL [60] and ATL [63] are languages that can be used, among other things, to perform model querying.

Model Validation: Models need to conform to some syntactic and semantic rules that the metamodel imposes [71]. Otherwise they become *inconsistent*. Model validation helps in the direction of identifying if models conform to these rules. In addition, there might be cases where models omit information, thus they are *incomplete* [72]. As MDE promotes the chain procedure of transformations between different artefacts, from models to a deployed system, it is critical for inconsistencies and incompleteness to be discovered otherwise they will be propagated. OCL [60] and the Epsilon Validation Language (EVL) [73] can be used for checking constraints on models. In [74], ATL [63] is used for checking models.

Other activities: Other model management activities include model *comparison* where a model containing similarities and differences between two other models is produced in the form of *traces*. Finally, model *merging* combines two or more models into one model.

In the following sections we discuss two modelling tools that are used in this thesis: the *Eclipse Modelling Framework (EMF)* [34] that implements the MOF architecture and the *Epsilon* suite [72] that is used for model management. In principle, the approaches presented in this thesis can be applied using any other tools described above; EMF and Epsilon are used for proof of concept and were selected with no other criterion rather than the familiarity of the author with these technologies.

Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) [34] is probably the most widely used metamodelling infrastructure and is part of the Eclipse Foundation [75]. It aligns with the four-layer MOF architecture and provides modellers with a metamodelling language called *Ecore*. Facilities to automatically generate graphical editors of metamodels and models are part of the framework. The *Emfatic* [76] textual language can be used to define metamodels in EMF. An example of Emfatic code for defining the metamodel presented in Figure 2.3 is given in Listing 2.1.

```
@namespace(uri="zooMMExample",prefix="zooMMExample")
package top;

class Zoo {
  attr String name;
  attr String entranceFee;
  val Animal[*] resident;
}

class Doctor {
  attr String name;
  attr int age;
  attr int salary;
  ref Animal[*] treats;
}

class Fan {
  attr String name;
  attr int age;
  ref Animal[*] supports;
}

abstract class Animal {
  attr String name;
  attr int ID;
  attr String sex;
  ref Animal[1] partner;
}

class Lion extends Animal {}

class Tiger extends Animal {}
```

Listing 2.1: Emfatic code to define the metamodel presented in Figure 2.3.

Epsilon

A compatible with the EMF framework tool is Epsilon [72], which stands for Extensible Platform for Specification of Integrated Languages for mOdel maNagement. As the name suggests, Epsilon consists of a number of languages that allow model management and manipulation. The core of the Epsilon suite is the *Epsilon Object Language (EOL)* [38], an imperative language that is inspired from Javascript [77] and OCL [60]. All the languages responsible for different model management procedures build atop EOL. Figure 2.7 shows the Epsilon architecture and summarises the languages available in the suite. The *Epsilon Transformation Language (ETL)* [62] is the language responsible for executing M2M transformations while the *Epsilon Generation Language (EGL)* [65] offers the infrastructure for M2T transformations. The *Epsilon Validation Language (EVL)* [73] allows validation of models while the *Epsilon Merging Language (EML)* [78] supports merging of multiple models.

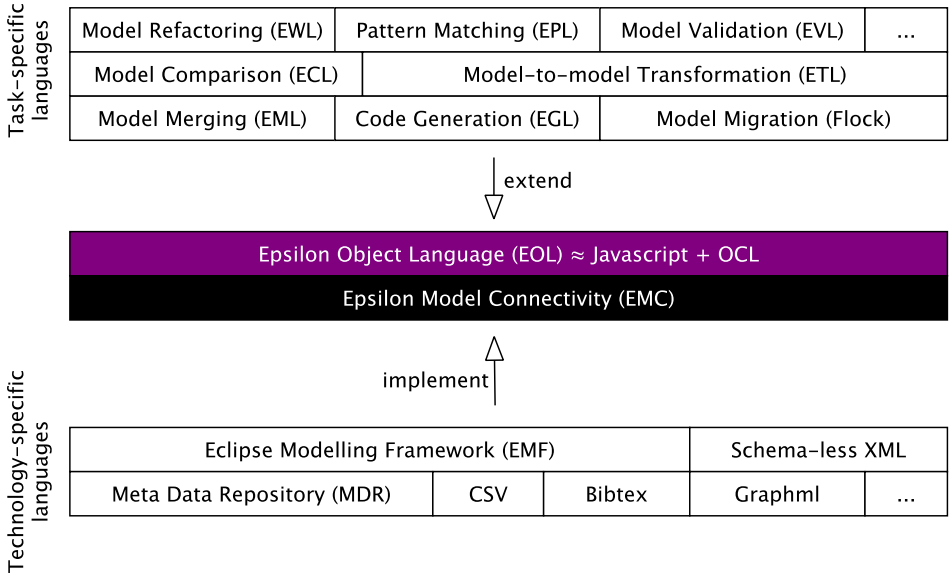


Figure 2.7.: The architecture of the Epsilon suite[1].

One of the benefits of Epsilon is that it is a technology agnostic model management suite. Metamodels and models can be expressed and manipulated in any format due to an intermediate layer called the *Epsilon Model Connectivity (EMC)* layer (see Figure 2.7). A new structure or filetype can be used as an input, if not already supported, by implementing a driver (parser) that translates the structure into the EMC's façade. At the moment there are EMC drivers for EMF models, XML files (with and without a schema), CSV files, BibTeX and much more. As of 2014, Epsilon also supports GraphML [79] diagrams as part of the *Muddles* [4] flexible modelling approach. An extensive presentation of the GraphML driver and

[1]Based on figure from `https://www.eclipse.org/epsilon/doc/`

the Muddles approach are given separately in Section 2.2.1 as this is the technology used by a proof of concept in this thesis.

2.1.2. Strengths and Weaknesses of MDE

MDE is promising some benefits in the software development processes it is deployed [2]. In this section these benefits are presented and verified based on surveys conducted in the sources presented. The shortcomings of MDE are also given.

Strengths

One of the most claimed benefits MDE brings is that of increased productivity. Mohagheghi et al. [80] performed a survey on how MDE is applied in industry. The outcome was that in general MDE contributed to increased productivity, however this was not always the case. Some studies suggested no or negative change in the productivity mainly due to the lack of good tools and well-define MDE processes. In the same survey, more claimed benefits of MDE were verified: better product quality, increased maintainability of software and reduced labour-hours due to the automation code generators bring [80]. That increase in productivity and thus the gain of adopting MDE is not always visible in some teams working on a software development project. For example, the team working in the maintainability of the system may find these models very beneficial while the team who worked to develop these to be used in the maintenance will only see this as an extra, time-consuming effort that adds no benefit to their own team [2].

Hutchinson et al. [81] performed an empirical study on the MDE brought where it was applied by submitting questionnaires to MDE practitioners, interviewing MDE professionals and by having on-site observations in companies applying MDE processes. A proportion of 58%-66% of the professionals that responded to the questionnaires (about 250 responses received) claimed that MDE has brought benefits to the personal and team productivity, the maintainability and the portability. A significant proportion (17%-22%), in contrast, disagreed. According to the same study the most important impacts MDE had was in better communication between stakeholders, improved code generation, better understanding of the problem at the abstract level and better design and documentation of the solution [81]. A survey conducted by Bone [82] between 122 MDE professionals verified the above claims as the average benefit that MDE brought to the overall project was judged to be medium-high (rated 3.89 out of 5.00).

The benefits of MDE in productivity and quality of the products were reported by Motorola in a study they published in [83]. More specifically, the authors claim that the benefits were due to a number of reasons, among others that of automation of labour-intensive tasks, reuse of designs and tests, focus of design on the application rather than the platform, etc. [83].

Weaknesses

The adoption of MDE in the industry is slow, although it offers all the aforementioned benefits [2]. Selic [84] identifies possible reasons for that from three different perspectives: *cultural and social* factors, *economic* factors and *technical* issues. From the cultural perspective, technology minded people, like programmers, are reluctant to leave the experience that coding offers and move to a more abstract level [84]. This is also due to the fact that most of the coders do not see the system as a whole but as individual components that are at some point integrated [84]. This human factor side is also verified by industrial experts [2]. Regarding the economic factors, Selic identifies that MDE is becoming slowly adopted because firstly managers are not willing to spend man hours on training programmers in learning new technologies like MDE and secondly because the cost of buying new tools used in MDE is an obstacle for them. The survey conducted by Mohagheghi also supports this claim [80]. In addition, the productivity and product quality will drop initially, until reaching the benefits that MDE offers [85]. Finally, the technical issues are grouped into three categories of challenges: *capability*, *scalability* and *usability* challenges [84].

In [86], Steimann criticises MDE by highlighting one of the benefits of MDE, that of simplification, is one of its drawbacks for adoption. More specifically, he claims that models are more valuable for those who are not good in programming, as models tend to oversimplify the system. If a system should be generated from the model, then the model might be too complicated and as a result people who do not know how to program will not be able to understand it. People who know how to program and possibly will be able to understand the model, prefer to code the system directly. However, as Fowler suggests in [87], the critical point for MDE to be useful is to define the "right" level of abstraction for each of the steps in the software development process and thus the right level of abstraction for the programming step, as well.

2.2. Bottom-up MDE

As mentioned above, one of the perceived weaknesses in the adoption of MDE is that of reduced usability of MDE tools and the extra training that MDE requires. MDE experts may be familiar with MDE tools and processes however *this is not always the case with domain experts* who hold the knowledge of the domain that is being modelled [5]. These professionals are more familiar with tools like simple drawing editors rather than complex MDE suites [22]. However, their involvement in the definition of high quality and well-defined DSLs and models that cover all the needed aspects of a domain is essential [5, 24, 25, 30]. A branch of MDE was introduced to help tackling the aforementioned shortcoming of MDE, that of *bottom-*

up or *flexible* MDE. The following sections introduce flexible MDE approaches and tools.

2.2.1. Muddles

In this section, we present the *Muddles* flexible modelling approach proposed in [4]. Muddles is used for the evaluation of the proposed approaches for type inference in this thesis.

Muddles allows language engineers to execute model management programs, like model-to-text transformations, on example models at the early stages of language development. This way language engineers are able to explore at the initial phases if the models of the envisioned language are fit for purpose [4]. In addition, the adoption of simple drawing editors, which arguably can be used by domain experts, enhance the participation of these experts in the language definition process resulting in better DSLs [4, 5].

The process starts by having domain experts and languages engineers draw example models of the envisioned DSL. These models are then annotated, by either the language engineers or domain experts, with types and type-related information like attributes. Following an automatic multipass model-to-model transformation the diagrams are transformed into a model that conforms to the Muddle metamodel and are ready to be consumed by model managements suites like the Epsilon platform [33]. An overview of the approach is shown in Figure 2.8. We now explain each step in detail through an example.

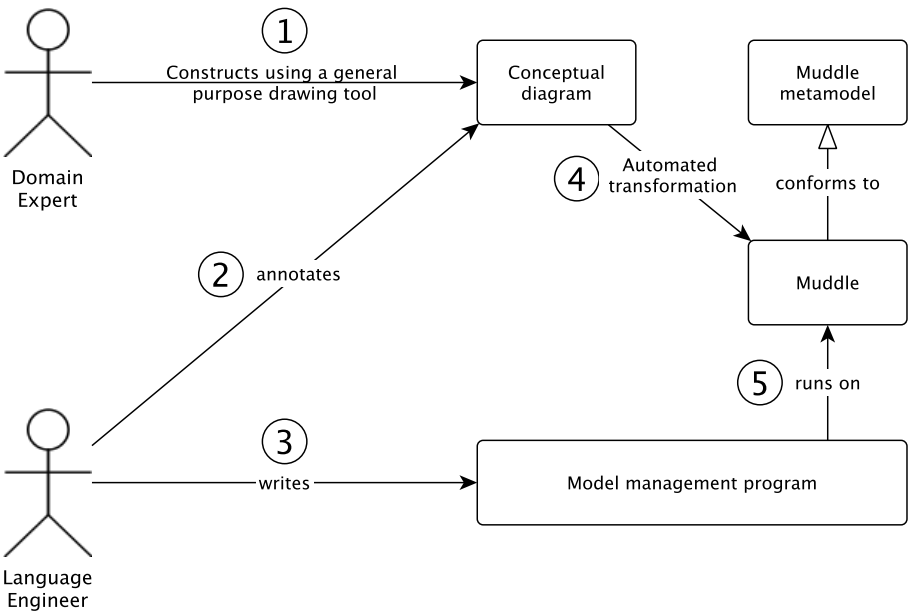


Figure 2.8.: An overview of the Muddles approach (based on Fig. 1 from [4]).

Example

Assume that the intention of a domain expert is to create a language that can be used to manage a zoo. One requirement for this language is to be able to list the names of all the animals and people involved in the zoo. The process starts by using a simple drawing editor to create an example model of a zoo (step (1)). In Muddles [4], the yEd [88] editor is used to create the example models which comply to the GraphML [35] modelling language. An example model is shown in Figure 2.9. At this point the diagrams are not annotated with types.

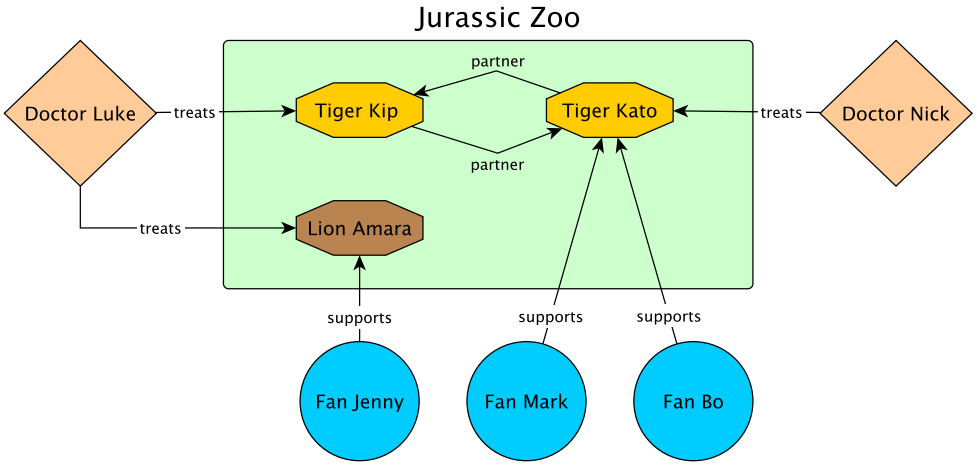


Figure 2.9.: An example model diagram in yEd representing a zoo configuration. Shapes and colours are not bound to types but can be used by domain experts for the better presentation of the example models.

After the example model is created, language engineers and/or domain experts can start annotating it with types and type related information (step (2)). GraphML does not support the notion of "Type" for the nodes and edges. This is done by exploiting its extensibility facilities [35]. The information is provided using the built-in parameters input window of yEd. For example, one can write in the type input field for the node named "Tiger Kip" the word "Tiger" declaring that its type is "Tiger". Using the ">" symbol the inheritance relationship is expressed (e.g., "Tiger" > "Animal" defines that the class "Tiger" extends the class "Animal"). In the same manner, the types of the relationships are also defined. For example, the type of the directed edges from "Doctor" nodes (diamond shapes) to "Animal" nodes (hexagons) can be defined as instances of the "treats" relationship. In a muddle, the **types are not bound to the shape**; in the same drawing, the same shape can represent different types (e.g., a hexagon in Figure 2.9 represents both elements of type "Tiger" and "Lion").

Each node and edge might also have some attributes. These are defined as lines of text written in the "Properties" input field (e.g., String name = Kip, Integer age = 12, etc.). Beyond types and attributes, Muddles capture additional information

using the extensions summarised in Table 2.1.

Table 2.1.: Element properties (based on Table 1 from [4]).

Extension	For	Description	Example
Type	Node, Edge	The type of the element	Lion, Doctor < Person
Properties	Node, Edge	Descriptors and values for primitive attributes of nodes/edges	String name = Jenny, Integer age = 25
Default	Node, Edge	Descriptor of the slot under which the first label of the node/edge should be made accessible	name, label
Source role	Edge	Descriptor of the role of the source end of the edge	source, sourceNode
Target role	Edge	Descriptor of the role of the target end of the edge	target, targetNode
Role in source	Edge	Descriptor of the role of the edge in its source node	patient 0..5, partner 0..1
Role in target	Edge	Descriptor of the role of the edge in its target node	carer *, employee *

The values of the *Source role*, *Target role*, *Role in source*, and *Role in target* fields of an edge define the name and multiplicity of the respective roles. For example, the edges whose type is defined as "treats" can have these values set to *"source"*, *"target"*, *"treats *"* and *"treatBy 1"* respectively. The numbers at the end declare the multiplicities of the reference (e.g., a "Doctor" treats many "Tigers", a "Tiger" is treated by 1 "Doctor"). Model management programs use this information to access and manipulate elements of the diagram.

Language engineers can then write model management scripts to query the drawing (step (3)). In Muddles, the Epsilon [33] platform can be used used to write these scripts. An example written in EOL [38] is given in Listing 2.2.

```
var fans = Fan.all();
for (f in fans) {
  ("Fan: " + f.name).println();
}
```

Listing 2.2: EOL commands executed on the drawing

Assuming that the circular elements (typed as "Fan") have a String attribute called "name" assigned to them, then the script returns the names of all of them. As such, muddles can be programmatically processed like other models, without having to transform them to a more rigorous format (e.g., Ecore). In order for this to happen, a multipass model-to-model transformation is executed in the background (step (4)) that transforms the GraphML drawings to a model that conforms to the Muddles metamodel (the metamodel is presented in Figure 2.10). The steps of the

transformation are listed below.

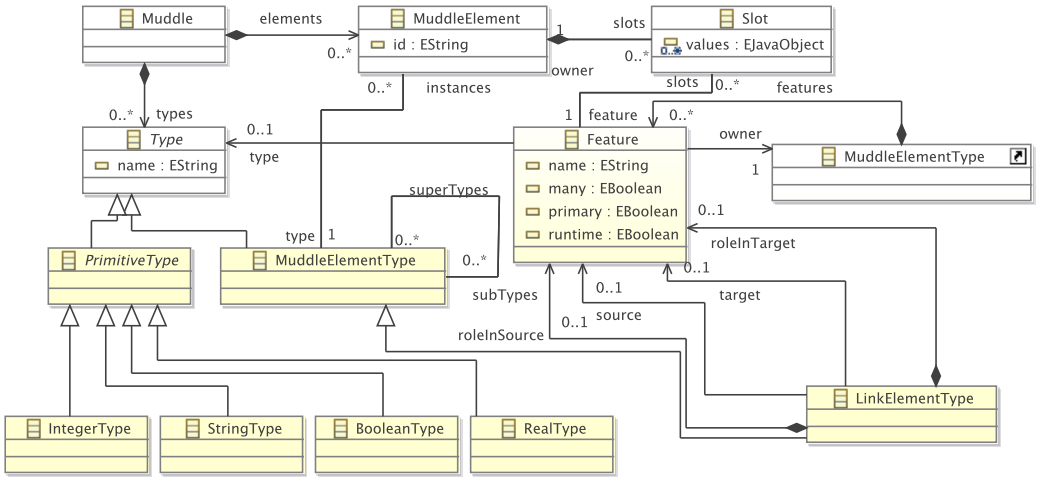


Figure 2.10.: The Muddle metamodel.

1. For every typed node an instance of the *MuddleElement* class is created. If the type assigned to this node does not already exist, an instance of the *MuddleElementType* is also created. Its *name* attribute value is the value of the type written in the appropriate field in the diagram. Untyped nodes are ignored.

2. When all the nodes are created then the transformation algorithm iterates through all of them to create and attach each of their attributes. The attributes are created as instances of the *Feature* class and their type is one of the appropriate descendants of the *PrimitiveType* class.

3. Then both the labelled and unlabelled edges are parsed: a new *Feature* is added to the type of the source muddle element of the edge. A *Slot* is added to the source node while the target of the edge is set to the values of the aforementioned slot.

4. The unlabelled and untyped edges are parsed again and the algorithm tries to fit their targets into appropriate slots of the source muddle elements (i.e., slots that already contain at least one value of the same type).

5. For the remaining edges (i.e., typed) the transformation follows the same behaviour as in step 1 but this time the instances are of type *LinkElementType* instead of *MuddelElementType*. In this step the *Feature* instances for the role in source, role in target, source and target are also created and attached to the *LinkElementType*.

6. The same process as in step 2 is followed for the attributes of the edges.

7. Finally, the multiplicities of the roles are set based on the maximum number of values of their slots.

The model management scripts written are now executed on the drawing. As the transformation described above is transparent, steps (3) to (5) (see Figure 2.8) are done all in once. As mentioned above, the Muddles approach is part of the Epsilon suite and it can be used by installing the Epsilon platform [2].

Muddles [4] is a flexible MDE approach that is used for evaluating the type inference approaches proposed in this work. Other approaches in flexible MDE are proposed in the literature, a presentation of which follows.

2.2.2. metaBUP

MetaBUP [5, 29] is a tool that promotes the interactive construction and evolution of metamodels based on example models. Domain experts use simple drawing editors like yEd [88] and Dia [89] to express example models which can then be used to automatically extract the metamodel. The approach and the tool are based on five requirements that are listed below [5]:

1. *Bottom-up:* The approach should be based on the creation of example models rather than a (even draft) metamodel at the beginning.

2. *Interactive:* An interactive, *iterative* approach will help in the definition of a first draft metamodel based on the example models, and as example models evolve, the metamodel should evolve as well.

3. *Exploratory:* Domain experts should be able to annotate different elements of the example models with the intended behaviour which should then be applied to the draft metamodel. If there are clashing annotations this should be reported to the language engineers.

4. *Guided by best-practices:* As the users of this approach will be non-MDE professionals, guidance on best practices should be given based on well known design patterns and refactorings.

5. *Technology agnostic:* The metamodels built should not be affected by any implementation related restrictions imposed by the underlying MDE technology used. Thus, these technological related issues should be postponed for the final stage when the metamodel will be deployed on a specific platform.

An overview of the approach is shown in Figure 2.11. A detailed description of the approach follows.

[2] www.eclipse.org/epsilon/

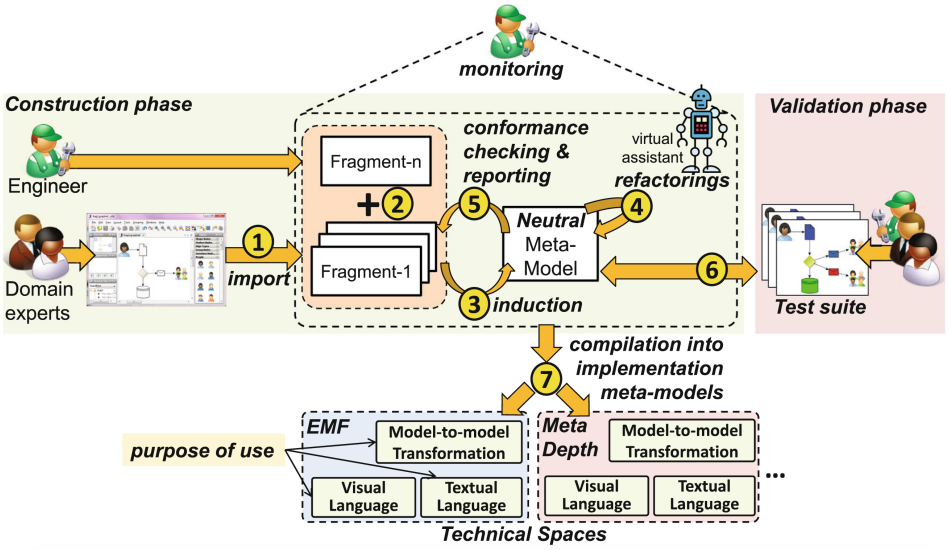


Figure 2.11.: An overview of the metaBUP approach (from [5]).

The process starts (step (1)) by having domain experts expressing example models which in this approach are called *fragments*. MDE engineers can also write their own fragments or annotate those the domain experts have written with more information (step (2)). These fragments are the input for the inference of a first draft metamodel (step (3)). At this point the iterative process starts. Domain experts and language engineers can either improve the draft metamodel by applying the refactorings suggested by the system (step (4)), or write new fragments to expand the metamodel (step (5)). A validator is attached to the tool that checks if the created example models conform to the devised metamodel (step (6)). Wrong models can also be fed to the validator to make sure that the inferred metamodel rejects them. When a final metamodel is obtained, the neutral metamodel is transformed (step (7)) to the platform specific one (e.g., EMF metamodel).

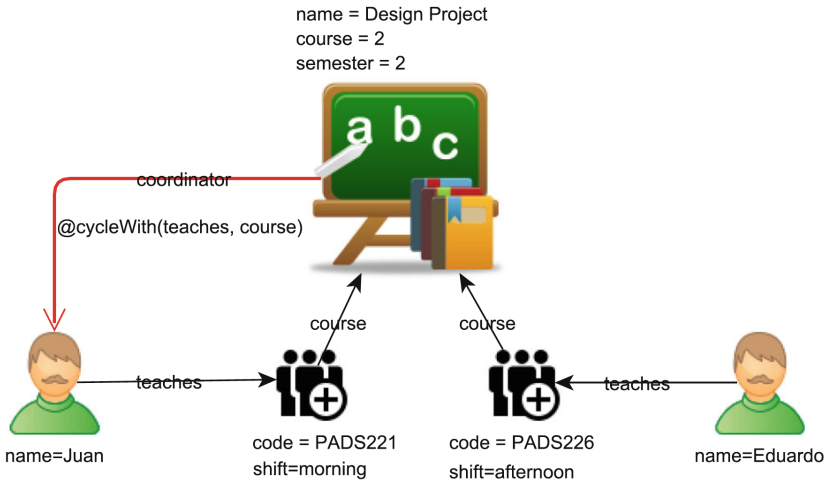


Figure 2.12.: An example visual fragment (from [5]).

Model fragments are the core of this approach. Fragments can be defined graphically, by using a wide variety of drawing editors like yEd [88], Dia [89], Microsoft PowerPoint [90] or Visio [91], or textually. An example for a visual fragment that describes a simple educational DSML is shown in Figure 2.12 and part of its textual equivalent in Listing 2.3. The latter can be generated automatically if the fragments are sketched using either the yEd or Dia editors. The type of each of the nodes in the diagram is defined by using a *legend*: **each symbol (shape) is mapped to one type** while **each concept may be mapped to more than one symbols**.

```
fragment edu1 {
  ...
  g1 : Group {
    attr code = ''PADS221''
    attr shift = ''morning''
    ref course = c
  }
  ...
  p1 : Professor {
    attr name = ''Juan''
    ref teaches = g1
  }
  ...
}
```

Listing 2.3: A fragment expressed using text (from [5]).

Engineers can annotate the fragments with properties that declare either domain or design aspects of the diagram. For example, if the annotation *@unique* is assigned to an attribute that means that the specific attribute acts like an id and thus the same value cannot be reused. A comprehensive list of all the available annotations is provided in [5].

The metamodel is inferred on the fly as new model fragments are added. A detailed description of how metamodels are inferred in metaBUP is given in Section 2.4.

As mentioned before, the approach performs some automatic refactorings based on best practices available in the literature. If these recommendations lead to breaking the conformance of any of the model fragments with the refactored metamodel, then the tool does not proceed but alerts the engineers about the conflict. Otherwise, it proceeds by applying the proposed refactoring and marks the change. If there are more alternative recommendations then one is applied and the change is marked as an "open issue" giving engineers the chance to pick any other alternative if they prefer [5].

The induced metamodel can be validated against example models that domain experts provide. More specifically, in the xUnit style [92], domain experts can provide example models that should or should not conform to the metamodel. The metamodel should accept the former and reject the latter. This validation is done in a straightforward way: if importing the model invokes changes to the metamodel,

then that means that it does not conform to it and thus it is rejected. Otherwise it conforms and thus it is accepted. As a result, validation is not done by writing code but by providing sketches which domain experts are able to write [5].

Finally, the inferred metamodel is platform independent, meaning that it abstracts away technical detail [93]. In order to be used by another MDE tool, this metamodel should be transformed to support that specific platform. Currently there are two platforms supported (i.e., EMF [34] and metaDepth [94]). For example, as EMF requires a root class for the metamodel so that the tree editor can be used, the algorithm automatically adds it to the metamodel [5].

The approach is supported by an Eclipse based tool called metaBUP[3] which offers all the functionality presented above for importing sketches, editing them, inferring the neutral metamodel, proposing and apply refactorings, validating metamodels and compiling a platform specific version of the metamodel.

2.2.3. Flexisketch

Flexisketch [6] is a mobile based drawing editor that allows users to draw models. The approach is based on the assumption that business analysts prefer to use pencil and paper for their modelling activities. Flexisketch mimics this paper and pencil approach by offering freeform drawing of models electronically. The drawn elements can have types annotated to them to allow model management suites run on the drawings. In this way the models written at the initial phases of the product development can be reused throughout the whole software engineering process [6].

As one can extract from the above, the main goal of Flexisketch is to unify freeform modelling with formal structured modelling [6]. The high level requirements for an approach that allows the authors to fulfil this goal are highlighted in [6] and summarised below:

1. *High flexibility*: The approach should offer users an unrestricted environment in terms of what they should be able to draw.

2. *Natural sketching*: The drawing process should look and feel like sketching on a whiteboard. The authors suggest that multi-touch tablet devices are capable to do so.

3. *Formalization capabilities*: The informal models should be (semi-)automatically transformed to structured artefacts that can be managed through model management suites.

4. *Speed*: In order for the process to be used, the tool should offer an experience that will be no slower than the traditional paper and pencil approach.

[3] `http://jesusjlopezf.github.io/metaBup/`

Flexisketch is an iterative approach that is based on three basic phases: *modelling*, *metamodelling* and *sketch recognition*. This iterative manner is depicted in Figure 2.13.

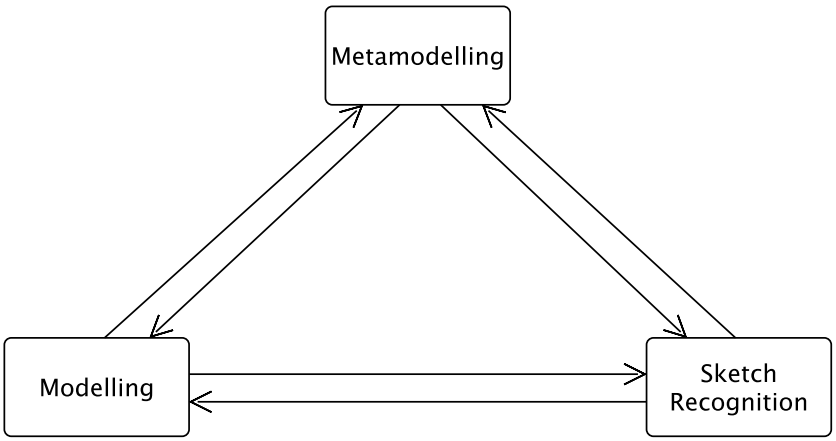


Figure 2.13.: The Flexisketch approach's three basic phases (from [6]).

More specifically, the modelling step consists of two modes. The first is the drawing mode, which allows domain experts to sketch their diagrams and the second is the mode that allows manipulation of the drawn elements (e.g., actions like resize, move, delete, etc.) The metamodelling step is when the untyped elements are annotated with types that define the concept they describe. This is the step where the modelling language definition takes place. In Flexisketch this is done by simply typing the type of the drawn element using the appropriate input field. When the type is given, the tool automatically adds the symbol and its attached type to a library of known symbols and types. In each library the same symbol is attached to a unique type however it is possible to define and use in the same drawing multiple libraries of symbols among which the same symbol might represent a different type offering more flexibility to the users. In the third step, that of sketch recognition, hand drawn symbols are recognised on the fly as they are drawn. This is done in a semi-automatic and interactive way: the system suggests possible alternatives for the type and user picks the correct one. This way the tool is also trained to better recognise future drawn symbols. The opposite is also allowed (i.e., the same type can have different symbols attached to it). The way the types of the sketches are predicted is described in Section 2.4.

A prototype tool supporting the Flexisketch approach was built for mobile and tablet devices that run the Android OS[4]. A screenshot of the application is shown in Figure 2.14 where a node is created and its type is assigned using the input field.

The authors performed two studies to evaluate the approach and the tool. The results of the experiments run among users revealed that firstly the tool is not as

[4] `https://play.google.com/store/apps/details?id=ch.uzh.ifi.rerg.flexisketch`

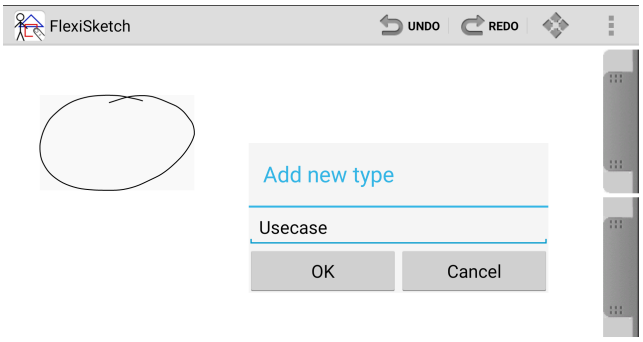


Figure 2.14.: The Flexisketch Android application.

fast as the traditional paper and pencil approaches so changes made to the software according to the feedback improved that. Secondly, users were able to classify the drawn elements (add their type) and finally, the majority of the participants were positive about the idea of using the approach as part of their modelling activities.

2.2.4. Other

The Business Insight Toolkit (BITKit) was proposed in [22, 95] by IBM as an approach to flexible MDE that focuses on filling the gap of modelling the business needs in pre-requirements analysis. As in Flexisketch (see Section 2.2.3), the authors of BITKit claim that from their experience, domain experts and business analysts still prefer to use office tools rather than MDE tools that offer the needed structure to carry the elicited requirements through-out the whole software engineering lifecycle [22]. Business Process Management approaches, like BPMN [46], could be used but due to the abstract and the specific concrete syntax used, they are restricting domain experts from expressing their needs. Thus, they argue that a flexible modelling approach is more suitable [22].

BITKit tool is based on three underlying models: the *visual*, *mapping* and *content* models. The last holds the content that the business analysts have written so far. The first (visual) holds information on how the content is displayed to the users and finally, the mapping model holds the rules on how the content is visualised and the rules on what a specific visual effect means for the content (e.g., red human shapes point to important stakeholders). When these models are constructed a metamodel holding the new types and their features is created in the back-end. However, this metamodel is not enforcing any rules to the front-end: types might violate the semantics of the metamodel. The reason for having this metamodel built is that of being able to import the concepts of a well established domain to future clients so the clients do not need to start from scratch [22].

Kurhmann in [32] suggests the use of drawing editors with free form capabilities for the definition of DSLs and presents the framework under which such a tool should be created. The envisioned approach is based on the *Process Development Environment (PDE)*[5] [96, 97] which is built atop the Microsoft DSL ToolKit [98] and offers the infrastructure for the development of DSLs. In PDE, the *PDE language* is used to define the DSL. In [8], PDE is extended to realise the envisioned idea and support the development of DSLs based on free-form sketches in a bottom-up way.

The process starts by having users drag-and-drop shapes from a palette that represent concepts of the envisioned DSL. Domain experts should assign the type of the concept that this drawn instance represents and assign attributes to it (if any). The metamodel of the DSL is built as new instances are added. A key requirement for this approach is that all the types and the attributes should be provided in the example models. Details on the metamodel inference are presented in Section 2.4. When the metamodel of the DSL is constructed an editor is generated for using the graphical DSL.

The aforementioned approaches focus on the extraction of the metamodel for visual DS(M)Ls. Roth et al. [99] propose an approach to the bottom-up development of textual DSLs. More specifically, their tool can infer a grammar from a set of textual examples. These examples are snippets of free text entered in a dedicated text editor. The grammar inference is based on regular expressions and lexical analysis [99].

The authors in [24] propose an Eclipse plug-in called *Collaboro* that can be used for the *collaborative* definition of DSLs based on flexible MDE principles. More specifically, Collaboro, a DSL itself, can be used by both technical and domain experts during the definition of the abstract syntax of a DSL to propose missing concepts and functionality for the DSL under construction. The proposed features are becoming available to all the stakeholders which can then vote for accepting or rejecting them. The goal of Collaboro is to involve domain experts in all the steps of the definition of a DSL, including the stages of design and implementation, and not restrict their role in the first and last phases, where the requirements are elicited and the DSL is deployed, respectively [24]. Collaboration in the domain of MDE is also used in the development of modelling tools in [100] and models in [101] and [102].

A flexible approach called DSL-maps is used in [103] to represent requirements for DSLs and the automated transformation of them into a metamodel. The requirements of the DSL are expressed using models, the notation for which is inspired by mindmaps [104]. A customisable transformation is used to automatically produce the metamodel based on these models, which is then refactored based on well known DSL patterns [103].

Finally, in [105] an approach to the development of modelling environments for

[5] `https://pde.codeplex.com/`

graphical DSLs based on example models is proposed. The approach is based on the metaBUP [5,29] flexible MDE approach presented in Section 2.2.2.

2.3. Partial Models

In the literature there are different definitions of model partiality. In [106], a partial model is a system model in which uncertainty about an aspect of the system is captured explicitly. In this context, "uncertainty" means "multiple possibilities"; for example a model element *may* be present or not. In contrast to [106], in the context of this thesis model partiality means that a model fragment contains incomplete information. For example element types can be missing. Our notion of model partiality is close to the one of [107] and [108].

Rabbi et al. [107] propose a diagrammatic approach to the completion of partial models based on category theory. Their approach is extending the Diagram Predicate Framework (DPF) [109], a tool that allows the definition of metamodels and models using diagrammatic specifications and constraints. In order to support model completion, the framework was extended with the notion of *completion rules*. Completion rules are extending the built-in functionality of predicates that DPF offers and are used to express the abstract syntax and semantic constraints set by the metamodel. When a rule is not satisfied by a model (thus the model is partial) then model transformations are triggered to fix the problematic part. For example, if a completion rule describes that all the instances of a model should contain an instance of a specific class, then a transformation is invoked to create that instance in case it is missing.

Similarly, in [108] the authors use Constraint Logic Programming (CLP) to assign appropriate values for every missing property in the partial model so that it satisfies the structural requirements imposed by the meta-model. More specifically, the semantics and abstract syntax provided by the metamodel, like source and target types of references and multiplicity constraints are automatically translated to a set of rules expressed in a format of a logic programming predicate. Their approach also supports the translation of Object Constraint Programming (OCL) constraints defined by the modellers. The rules created are expressed in the ECL^{i}PS^e [110] constraint logic programming language which is based on PROLOG [111]. Details on these technologies are given in Section 2.7.

The process proposed in [108] starts by synthesizing the available elements of the partial model into CLP facts. For example, all the nodes and references are listed, attributes and their values are attached to the nodes, etc. Then the process continues by creating what is called the *domain* of each variable in the constraint satisfaction problem. The domain actually includes all the rules that are imposed by the metamodel and the OCL constraints. That includes the definition of the lower and upper bound for each relationship (multiplicities), the possible source

and target types for each relationship, the possible values for the attribute of a type, etc. Having this set of facts and rules the Constraint Satisfaction Problem (CSP) is ready to be solved. The ECL^{i}PS^e engine applies the rules to the facts to prune the search space of possible solutions and comes up with an arrangement of the elements provided in the first set that satisfies the rules. If there is no possible solution this is stated to the engineers.

The aim of both aforementioned approaches is to provide model completion to reduce modelling effort in the same manner that code completion provided by programming language editors reduces coding effort. Moreover, both these approaches rely on a metamodel to produce the rules for the model completion.

Antkiewicz et al. [112] propose an approach to partial model completion based on the *Clafer* [113] language, which is a modelling language with first class support for feature modelling. The main aim of this approach is to use model examples for improving domain comprehension. In this work, partial models are expressed in Clafer while constraints imposed by the metamodels are expressed using first order predicate logic formulas using the Clafer syntax, with a similar syntax to that in which predicates are expressed in Alloy [114]. The partial models expressed in Clafer syntax and the rules imposed by the metamodel are then given to the inference engine to produce a complete model. This work, as the two mentioned above, relies on a metamodel.

2.4. Metamodel and Type Inference

2.4.1. MetaBUP

One of the steps described for the metaBUP bottom-up MDE approach (see Section 2.2.2), is that of *metamodel inference* from example models (called fragments in that approach). When a new fragment is created the algorithm checks if its type exists in the metamodel. If not, the type is created. Then the algorithm checks for the attributes attached to the specific type in the current model fragment. If any attribute does not exist in the metamodel under that specific type, this attribute is added. The same strategy in principle is followed for the creation of references. If a reference bound with a specific type in the example model already exists for that type then it is ignored. If not it is created. In the cases where a reference already exists but it is targeting a different node than the one appearing in the fragment, the algorithm automatically creates an abstract superclass for the two classes and updates the target of the reference to point to the newly created abstract superclass. Cardinalities are created by counting the minimum and the maximum numbers of nodes that a specific reference points to in the different fragments. After the metamodel inference these can be updated manually. The algorithm can also update the cardinalities to many if the name used for the reference is a plural noun [5].

Regarding *type inference*, the metaBUP approach relies only on the shape/icon of a specific element to infer its type. Shapes are mapped to types and thus, when a specific shape is re-used it automatically carries the concept (type) with it.

2.4.2. Flexisketch

As mentioned in Section 2.2.3, Flexisketch [6] has a built in mechanism for *type inference* that is based on sketch recognition. All the symbols are mapped to a type. These mappings are stored in libraries of symbols. When a new symbol is being drawn then the tool automatically searches the library to find the top three most similar symbols. The similarity of the symbols is calculated by using the trainable sketch recognition approach proposed in [7] by Coyette et al.

Coyette et al. use a method that is also used in biometric characteristics recognition [115] and is based on comparison between string vectors. More specifically, each hand drawn shape is translated into a sequence of line segments. Each segment is characterised firstly by its position on a square grid and secondly by a number that declares its direction (see Figure 2.15). This way each hand drawn element is represented by a string. The comparison between these strings is done using the *Levenshtein's string edit distance* measurement [116].

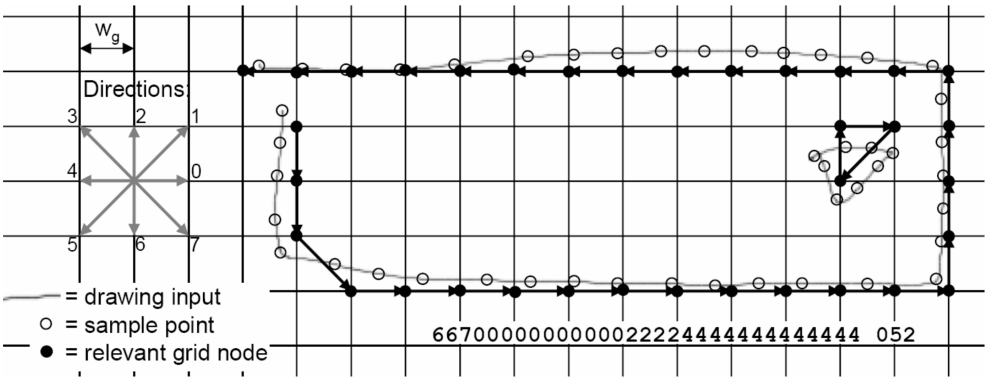


Figure 2.15.: String representation of sketches in the Coyette et al.'s approach (from [7]).

2.4.3. MLCBD

MLCBD [117] is an approach built to support the semi-automatic inference of the abstract and concrete syntax and the semantics of visual DSMLs based on example models. The process starts with the definition of example models of the envisioned DSL using a drawing canvas similar to tools that domain experts are usually familiar with (e.g., Microsoft Visio [91]). These models are then automatically transformed into undirected graph representations using the built-in *Graph Builder* [117].

After all example models are transformed into graphs, the *Concrete Syntax Identifier* parses all the models to identify unique shapes and styles that are used in

representing concepts in the diagram. These become the candidates for the concrete syntax and it is up to the domain expert to pick the desired concrete syntax representation for each concept and give a name to it. Thus, this process of concrete syntax inference is done in a semi-automatic manner. The example models might also include links that connect nodes. The concrete syntax of these is inferred using the same process as for nodes. In addition at this point, the domain expert has to select if the inferred link is directional or not.

In the next step, the undirected graphs are transformed into directed graphs by applying the information given in the previous step. Optimisation is also performed by pruning those nodes that represent the same concepts. This final graph is given to the *Metamodel Inference Engine* to *infer the metamodel* of the DSL.

Firstly, all the graphs generated based on the example models are merged together in one graph. Then, the cardinalities of the references and the dependencies are calculated. Finally, for the induction of the metamodel, the merged graph is tested over a set of graphs, instances of metamodels that define metamodel design patterns [118], in order to check (sub) graph isomorphism [117].

2.4.4. Process Development Environment (PDE)

The *metamodel inference* in the example-based DSL development version of the Process Development Environment (PDE) [8] is done following similar rules with those of metaBUP [5] (see above).

When a new node is created a type and the attributes of the node (if any) should be assigned to it. A new instance of the *DomainClass* type is created for each unique type (see the conceptual model of PDE-based languages in Figure 2.16). In the same manner, instances of the *DomainRelationship* classes are created when a new link is created between two nodes. This gradual construction of the conceptual metamodel is the same that is used when a DSL is created using the textual syntax of the PDE-language. The only difference is that the input method is changed: nodes and links are drawn by the user rather than using textual commands.

PDE, like MetaBUP, offers language optimisation. In traditional DSL development, where language engineers assume they have all the needed information of the domain in hand, the DSL is constructed (or at least it is possible to be constructed) following the best practices and patterns. However, when a DSL is constructed using a flexible modelling approach, this information is gathered on the fly. Thus, the conceptual metamodel that is automatically created is not always the optimal. PDE includes a model optimisation method that automatically refactors the metamodel based on well known defined patterns (based on DSL patterns identified by Fowler et al. [119]). For example, if different types share the same attributes, the application asks the engineer if a base class that includes these attributes should be created.

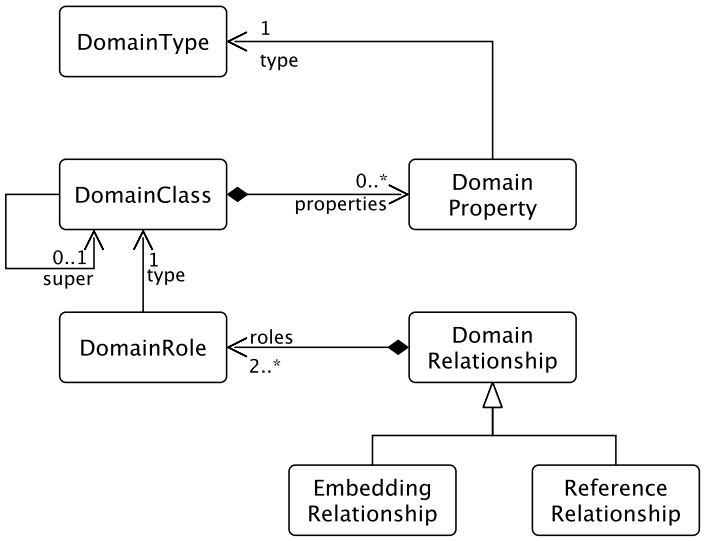


Figure 2.16.: Concept metamodel of PDE-based languages (adapted from [8]).

2.4.5. Metamodel Recovery System (MARS)

In [9], Javed et al. propose an approach to *metamodel inference* from a set of models after migrating or losing their metamodel. Their semi-automatic approach is based on grammar-schema inference approaches. Metamodels evolve to adapt to changing requirements and needs of the domain they represent [9]. The instances that conformed to a legacy version of the metamodel can be transformed to adapt to the new metamodel. In case both the deprecated and the new metamodel exist then approaches like the one proposed in [120] can be used to create the mappings between the concepts in the old and the new metamodel to support the domain models migration. However, when either the old metamodel does not exist or the mappings cannot be created those approaches are not working.

An overview of the MARS approach is shown in Figure 2.17. MARS is based on the Generic Modelling Environment (GME) [121], a modelling tool that stores in a database all the model elements and their relationships that are available in a domain based on a metamodel that is given as an input [122]. The process starts by providing example models expressed in XML. An XSLT [123] translator, which can transform XML documents to any other textual format, is used to prune XML specific content from the models and keep only the domain related information (step (1)). This information is expressed in a DSL developed as part of the approach, called the Model Representation Language (MRL) [9]. MRL generated code is then given as an input to the metamodel inference engine (step (2)), based on LISA [124]. LISA is a grammar inference engine that inducts the grammar of a programming language using context-free grammar (CFG) inputs (step (3)). The metamodel is then constructed based on this grammar.

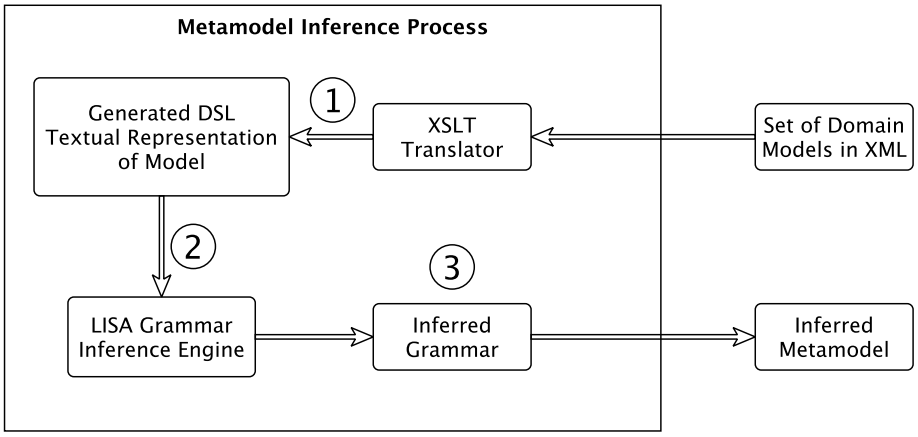


Figure 2.17.: MARS metamodel inference approach (adapted from [9]).

2.5. Summary and Critique of Flexible MDE approaches

As mentioned in Section 2.1.2, one of the weaknesses of MDE is that of the need of training in order to be adopted in practice. Domain experts, whose involvement and collaboration in MDE processes is important, are not usually specialised in using MDE tools which require a good understanding of MDE principles. Flexible MDE offers the needed looseness by not requiring the implicit compliance with rigorous rules that traditional MDE approaches require. However, this comes with a number of drawbacks, one of which is the fact that some of the elements that are drawn using simple editors may be left untyped. If an element is left untyped, it is ignored; an undesirable effect if one thinks that one of the purposes of using flexible MDE approaches is the exploration of the domain and the identification of all the concepts the envisioned metamodel needs to include. Having mechanisms to prevent this from happening is of interest to enhance the usability of flexible MDE approaches. As mentioned in Section 1.1.1, there are at least two ways for achieving that. The first is the execution of querying scripts on the drawn example models which will identify the elements that have left untyped and request engineers to provide the missing information. A second, could be that of type inference, where the types are inferred and filled in either automatically or in a semi-automatic way. In the flexible MDE literature presented in Sections 2.2, 2.3 and 2.4 the vast majority of the approaches require all the elements to have a type assigned to them, or otherwise they are ignored. To the best of our knowledge, the only two approaches that offer type inference are metaBup [5, 29] and Flexisketch [6]. The critique of their type inference approach follows.

In metaBUP [5, 29], the type of each node is explicitly bound to its shape: each type when is instantiated is bound to a specific shape or icon. When this icon is drawn on the canvas, it carries the type with it. That implies that each shape and

icon can represent only and only one type. In contrast, one type can be represented by more than one shapes or icons. For example, one can use a rectangle and a circle to represent a single type, but then rectangles and circles cannot be used to represent any other type. Following this approach metaBUP guarantees that as soon as a type is introduced into the example model then all the elements that represent this notion will not be left untyped. However, we believe that it is a kind of restriction as it forces domain experts to use different shapes for all the types they express while they need to remember which shape is mapped to which type. This can also be seen as a requirement to have the concrete syntax of the graphical DSL in mind when probably is to early to do so. In some cases this can be useful as it can possibly help engineers consider or reject polymorphism requirements (i.e., having different types for concepts that are similar but conceptually belong to the same group or not). For example, should the metamodel include different types for "Lion" and "Tiger" or is an "Animal" concept enough? However, in some scenarios, especially at the exploratory phases of flexible MDE, domain experts are not ready or willing to have the concrete syntax defined before even having all the concepts (abstract syntax) expressed and thus such a strict requirement of attaching the concrete syntax to each type reduces their flexibility.

The Flexisketch [6] approach, follows the spirit of metaBUP in terms of type inference but in a more relaxed manner. Types are bound to shapes and icons but the same shape and icon can be used to represent different types. In addition, the inferred, based on graphical similarity, types are not directly bound to the elements, but are suggested to the engineers as the element is drawn. The suggestion consists of the types of the most similar shapes in the diagram. Flexisketch promotes the use of freehand drawings and thus the definition of the similarity between shapes is less strict; the approach looks for similar but not identical shapes. As done in metaBUP, Flexisketch, bases its type inference approach on the concrete syntax of the models that are drawn, which might not be useful in scenarios where the concrete syntax is not of interest at the phase of the development process the flexible MDE approach is used.

Finally, in Muddles [4] and all the approaches presented in Sections 2.2, 2.3 and 2.4, if a node is left untyped then it is ignored. There is no other way to define or infer the type of the node rather than implicitly filling its type in the appropriate input field. Although this introduces great flexibility in terms of not having a strong requirement to follow any rules while drawing example models, we believe that it reduces the functionality of the approach as critical elements of the envisioned metamodel can be easily ignored or overlooked. Thus, in this thesis we introduce approaches that can be used to infer the types of untyped nodes using algorithms and tools from three different domains, that of classification algorithms, constraint programming and graph matching. The following sections introduce the reader to the basic principles and tools of each of these three domains.

2.6. Classification Algorithms

Classification algorithms are used in this thesis for type inference. An introduction to their basic principles and a presentation of the most important classification algorithms are provided in this section.

Classification algorithms are a form of supervised machine learning for approximating functions mapping input features to a discrete output class from a finite set of possible values [125]. They require a *training dataset* with labelled examples of the output class to process, after which they can generalise from the previous examples to new unseen instances. For example, provided sufficient examples (i.e., diagram elements) of a specific type (e.g., Tiger from Figure 2.9) a classification algorithm can learn to predict the class Tiger when given an unlabelled example with the same properties. These properties are important for classification algorithms as they represent characteristics of the elements that are possibly distinctive features of the type/class each element belongs to. For example, a feature of an element in the context of MDE could be the shape used to represent the node. More examples of such properties will be given in the remaining of this section.

Many classification algorithms exist [125], some of the most established being Classification and Regression Trees (CART) [126], Random Forests (RF) [127], Support Vector Machines (SVM) [128] and Artificial Neural Networks (ANN) [129]. In the following sections these algorithms are presented.

2.6.1. Classification and Regression Trees (CART)

Classification and Regression Trees (CART) classify the elements of a set by producing a decision tree. CART is trained by considering all labelled instances in the training dataset in one single batch. For each input feature the information gain of using that feature to classify the instances in the batch is calculated. The feature with the highest information gain is used as the root node of the decision tree. The dataset is then split based on the values of the feature at the root node, and the process is repeated on each child node with each subset of the dataset until a stop condition (e.g., minimum number of instances in a leaf node, depth or accuracy of tree) is satisfied.

An example decision tree is illustrated in Figure 2.18. Internal nodes represent features (i.e., Outlook, Humidity and Wind), branches are labelled with values of the parent node (e.g., Sunny, Overcast or Rain for the Outlook feature) and leaf nodes represent the final classification given. To classify a new instance, start at the root of the tree and consider the feature specified and take the branch that represents the value of that feature in the new instance. Continue to process each internal node reached in the same manner until a leaf node is reached where the predicted classification of the new instance is the value of that leaf node. For example, given

the tree in Figure 2.18, a new instance with a Sunny Outlook and Normal Humidity would be positively classified (path is highlighted in Figure 2.18).

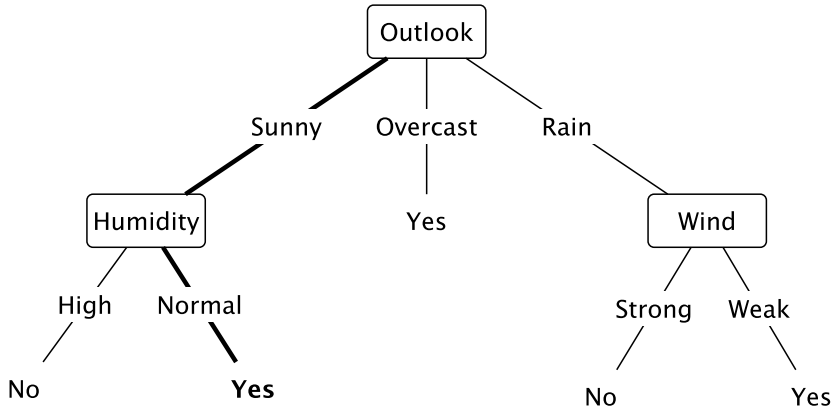


Figure 2.18.: An example of a decision tree in CART (from [10]).

2.6.2. Random Forests (RF)

Random Forests (RF), is a method that typically gives higher accuracy but less interpretable results than CART [130]. An RF is an ensemble of multiple decision trees, each trained on a different set of training instances from the training dataset chosen at random with replacement and often using a random subset of the input features. Once trained, the ensemble classifies new instances by processing each tree in the same manner as an individual decision tree and then choosing a single predicted class by majority vote. Intuitively, this typically increases the accuracy in a manner similar to the wisdom of crowds [131].

More formally, the combined multiple weak hypotheses in an RF will typically outperform the single hypothesis in CART due to each tree containing bias towards the data it observed but the ensemble being able to average out these biases. This advantage, however, is balanced by an increase in the complexity of the resultant model. Whilst it is simple to read a single decision tree and gain an understanding as to which features the model has correlated with a particular class, an ensemble of multiple trees becomes harder to read as many trees must be considered and the classifications of each combined to reach the final prediction of the model.

For both CART and RF, the performance can be evaluated by the accuracy of the resultant model (e.g., the decision tree learnt by CART) on test data not used when training. The performance, also called *accuracy*, of a model is the sum of true positives and negatives (i.e., all correctly classified instances) divided by the total number of instances in the test set. A single measure of accuracy can be artificially inflated due to the learnt model over-fitting bias in the dataset used for training. To overcome this, k-fold classification can be implemented [10]. This approach repeats

the process of training the model and testing the accuracy k times each time with a different split of the data into training and test data sets. The final accuracy using this method is then the mean value generated from the k repeats.

2.6.3. Support Vector Machines (SVM)

Support Vector Machines (SVM) are considered among the best performing classification algorithms [132]. In SVMs, the goal is the identification of the best classification function between the classes in the training set. The best function is realized geometrically and is the separating hyperplane function $f(x)$ that separates the instances of the classes. Having $f(x)$ identified, the classification is done by checking the sign of each element in the testing dataset (an element x_n belongs to the positive class if $f(x_n) > 0$).

The difference between SVMs and other classification algorithms that use a separation hyperplane is that SVMs seek and accept only the function that *maximises* the separating distance among all the possible separation functions. The benefits of doing that is that firstly it improves the performance (accuracy) of the classification in the training dataset and secondly increases the chances for correct classification of future data [132]. However, in order to improve the accuracy, the performance in terms of computational power and time is sacrificed.

By the definition of the hyperplane separation, SVMs imply that the data should be firstly linearly separable (the separation between the classes could be done by drawing a linear function) and secondly belong to two classes (binary classifiers). However, work in other directions is made in order to create kernel functions that are quadratic and exponential to support non-linear separable data that belong to more than two classes [133].

2.6.4. Artificial Neural Networks (ANN)

Artificial Neural Networks (ANN) are inspired by the structure and operations of the biological neural networks [129, 134]. As natural neural networks consist of neurons interconnected with each other, ANNs consist of interconnected simple computing nodes each of which acts like a summing device [135]. The neurons are connected with weighted links which are adjusted during the training process based on values and functions that are set by the engineers who set up the network. Having these computing node layers set and connected, new instances are moving through them until classified.

Initially, ANNs were able to solve only linear problems. However, new advanced methodologies deployed to solve non-linear problems as well [135]. ANNs promise high accuracy in the classification problems used. However, the setup of a well performing ANN requires expertise in the domain of machine learning, a good understanding of the domain the network will be applied to, experimentation with

a variety of parameters and weights and validation of the produced ANN [135].

Until recently, all the artificial neural networks consisted of a small number of layers (most commonly three). This was done for various reasons, mainly the Kolmogorov theorem [136] and the lack of computational efficiency. A new category of ANN was proposed to help in the direction of improving accuracy, especially when the domain involved audio and visual processing and recognition or chemical substances identification [137]. That new approach is called *Deep Artificial Neural Networks*. In contrast to the conventional classification algorithms, the representation learning (i.e., the extraction of a pattern based on input that allows the classification of future elements) in deep learning is done with multiple levels of abstraction [137]. Each layer, starting from the raw input which has no abstraction is one level of higher abstraction of the domain, slightly more abstract from the previous. Having multiple layers of varying abstraction leads to the creation of more complex functions that in principle perform better. For example, in the scenario of a classification problem, the low level abstraction consists of features that are not distinctive for the elements, while higher levels contain all that properties that are important for discrimination [137]. The key aspect of deep learning is that these layers are not identified by engineers but by the algorithm itself, having it trained using a general-purpose learning procedure [137].

2.7. Constraint Logic Programming

The use of logic as a programming language in computer science and not only as part of formal specifications began in the early 1970's [138]. Kowalski [139] and Colmerauer [140, 141] proposed the use of logic statements and PROgramming in LOGic (PROLOG) was born. The first PROLOG interpreter, based on the work of Colmerauer's team was developed in 1972 written in the ALGOL-W [142] language [138]. Since then many distributions of PROLOG were created; among the most used are ECL^{i}PS^e [110] and SWI-Prolog [143,144] A short description of these is given in Sections 2.7.1 and 2.7.1 respectively.

PROLOG is a declarative language. A logic program is split into two components: the *logic* and the *control*. The first defines the problem that needs to be solved and the second the way it should be solved [138]. A PROLOG program consists of commands each of which is called a *term*. Terms can be either one of the datatypes: *atoms* (i.e., names with no inherent meaning), *numbers*, *variables* (which act as placeholders for arbitrary values) or relations: *facts* or *rules* [138]. An example of a rule is the following.

$$woman(mary) : -true$$

The equivalent fact for the above rule is (note the "." at the end for the clause):

$$woman(mary).$$

A basic operation of PROLOG is that of querying. The program is asked a query and returns all the values that satisfy the question. If there is no such a value, the program returns a "No" statement. For example, one can ask the following query to the simple PROLOG program consisting of the above fact:

$$?- woman(X).$$

and it will return the answer

$$X = mary$$

X is working as a variable in this term (i.e., as a placeholder for possible values). That is very useful for solving a common category of problems for which PROLOG is widely used, that of *Constraint Satisfaction Problems (CSP)*. A CSP program usually consists of multiple rules and/or facts (constraints) and variables. Formally, this is denoted by the tuple $\langle V, D, C \rangle$ where V represents the set of variables in the problem, D the possible values that each variable can take (domain) and C the set of constraints on the variables [145]. The built-in $labelling$ predicate is called and returns all the possible sets of values, if any, for the variables that satisfy *all* the constraints.

Eventually, a Logic Programming paradigm can be summarised by the following three principles [110]:

- Each variable can be a number or a string, a structure (e.g., tree) or another program
- Variables are not assigned a value during the runtime but are constrained to a set of possible values by rules and facts
- There are alternative paths for computation of the possible values a variable can hold, each of which leads to a possible solution to the problem. If a value of a variable at a path leads to a state that violates one of the constraints, all the possible states that include this value are discarded as wrong and the program returns back to "open" points where more untested values should be checked; otherwise the value is marked as plausible and the program checks the values of the remaining variables. If a combination of values (a complete state) does not violate any of the rules then it is returned as a solution. This process (of rejecting a set of states and returning back to "open" points), called *backtracking*, is essential in constraint programming as it helps reduce the search space thus improve the performance of the logic program in terms of time. Search can be made more efficient by using optimizations such as

propagation (i.e., use partial assignments to remove unproductive values from domains of unassigned variables) or *backjumping* (i.e., reconsider several decisions in each backtrack). These features are provided in most state-of-the-art constraint solvers and, hence, these optimizations do not need to be implemented manually in the definition of each CSP.

The efficiency of PROLOG in modelling and solving constraint satisfaction problems is demonstrated through a classic example in the literature, that of colouring of maps. More specifically, the problem is that of how the map of a continent should be coloured in a way that none of the neighbouring countries (named "A", "B", "C" and "D") are filled with the same colour. This solution (shown in Listing 2.4) is based on the ECL^{i}PS^e [110] suite presented in the following section.

```
1 coloured(Countries) :-
2   Countries = [A,B,C,D],
3   value(A), value(B), value(C), value(D).
4   ne(A,D), ne(B,D), ne(C,D), ne(A,B), ne(B,C).
5
6 value(red).
7 value(green).
8 value(blue).
```

Listing 2.4: The problem of colouring a map solved using PROLOG (adapted from [14]).

The domain of possible values for the colour is defined in lines 6-8. Five constraints are given in line 4. All define that the variables representing the countries cannot have the same values if they are neighbouring (see Figure 2.19). For example, the values of the variable A and D cannot be equal (*ne* is a function to declare this *not equal* relation) as they are neighbouring. For solving this problem, PROLOG accepts the values in the four variables (A, B, C and D) that are not violating the five rules.

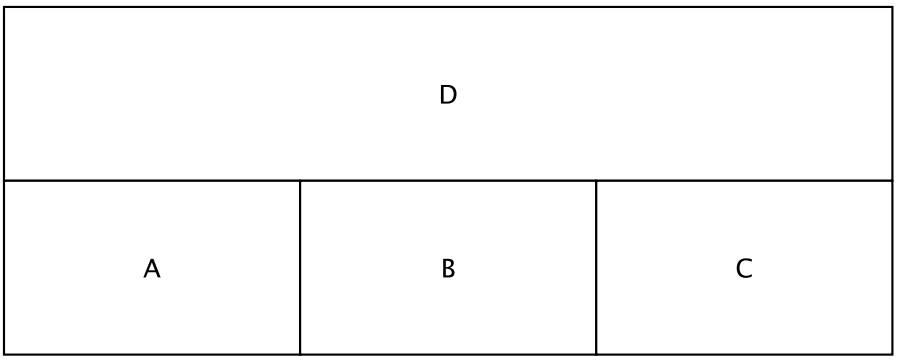


Figure 2.19.: A map for colouring.

2.7.1. Logic Programming Tools & Distributions

ECL^{i}PS^e

ECL^{i}PS^e [110] is a widely used PROLOG distribution that comes with an Integrated Development Environment (IDE). Except the logic programming facilities, ECL^{i}PS^e offers functionality for mathematical and stochastic programming [14].

Typically, a logic programming problem is not restricted to a specific category (i.e., constraint propagation/solving or constraint optimisation) but it is a combination of those [14]. One of the main advantages and features of ECL^{i}PS^e is that of offering support for combining different search algorithms and constraint handlers to more efficiently tackle the aforementioned hybrid problems [14]. More specifically, ECL^{i}PS^e supports different approaches: those that impose constraints (typically in lower levels of search trees) and those that repeatedly repair existing solutions to identify good solutions (stochastic techniques for optimisation) [14]. Regarding constraint handling algorithms, different approaches are supported, like finite domain propagation [146, Chapter 14] or linear constraint solving [147].

SWI-Prolog

SWI-Prolog [143, 144] is a free distribution of the PROLOG language developed at the University of Amsterdam. SWI-Prolog includes the core functionality of PROLOG language along with packages that provides developers with access to rich re-usable functions and procedures, multi-threading and distributed options, testing and more. It is one of the most widely used PROLOG distributions, especially by students and engineers working in the field of semantics web [143].

SWI-Prolog is available for all the major operating systems. There are versions available that support the use through a graphical user interface (GUI) (based on the XPCE suite [148]) or through the terminal.

2.7.2. Combining MDE with Logic Programming

Constraint Logic Programming (CLP) is applied in some MDE scenarios. This section presents these.

In [145], the *EMFtoCSP* tool[6] is proposed, which is used for the verification of EMF models annotated with OCL constraints. The authors claim that as models and model transformations become larger, they also become error-prone. Thus, a fully automatic approach that will be responsible to identify and fix these errors would be useful to improve the reliability of MDE-based processes [145].

The following correctness properties are supported by the tool: *strong satisfiability*, *weak satisfiability*, *lack of constraint subsumptions* and *lack of constraint redundancies*. EMFtoCSP translates the metamodel and the OCL constraints into a CSP

[6] `https://code.google.com/archive/a/eclipselabs.org/p/emftocsp`

and the solver tries to identify at least one instance of that metamodel that satisfies all the constraints. If there is one identified, the tool returns this example solution. Otherwise, the solver returns the "No" statement, meaning that there is no possible instance based on the criteria defined.

EMFtoCSP is an extension of the tool called *UMLtoCSP* [149] which in a similar manner to EMFtoCSP [145] uses CLP for the formal verification of UML class diagrams instead of EMF.

Finally, as described in 2.3, the approach presented in [108] translates the constraints imposed by the metamodel and a partial model into a CSP, and the solver tries to find a solution that creates a complete model based on the partial model and the domain of the variables given.

2.8. Graph Similarity

In the graph similarity domain the main goal is that of identifying matches between the nodes of the source and the target graphs based on different similarity criteria (metrics). In MDE, a model and a metamodel can be seen as two graphs in which the nodes of the first are instances of the nodes of the second. From the perspective of this thesis, finding matches between the nodes of the example models and the nodes of the draft metamodel inferred in one of the iterations that take place in common flexible MDE approaches (see Figure 1.1), could help in identifying the type of the untyped nodes.

In the literature there are different approaches proposed to compare graphs; a classification can be found in [150, 151]. The majority of them are domain specific, meaning that their are developed to work in a specific domain (e.g., database schema matching, XML matching, etc.). In [13, 152], a *generic* graph matching algorithm, called *Similarity Flooding* is proposed which can be used for finding similarities between the nodes of graphs of different domains. The similarity flooding algorithm is presented in detail in Section 2.8.1. In addition, two approaches that use graph similarity in the domain of MDE, the domain of interest of this thesis, are also discussed in Section 2.8.2. The first [11] applies the similarity flooding algorithm for the automatic generation of transformation rules while the second [12] proposes the use of a custom graph similarity algorithm for the automatic generation of trace links in model transformations.

2.8.1. Similarity Flooding

Similarity flooding [13, 152] is a widely used *graph* matching algorithm. It accepts two graphs as input and produces mappings between corresponding nodes of the graphs. This *schema matching* procedure can be applied in different domains like ER diagrams, UML class taxonomies, XML schemas, etc. [13]. The key idea of

the similarity flooding, and thus the name, is the assumption that the similarity between two nodes is affecting the similarity of their neighbouring nodes (i.e., a node is "more" similar to another if their neighbouring nodes are similar). An overview of the similarity flooding approach is shown in Figure 2.20.

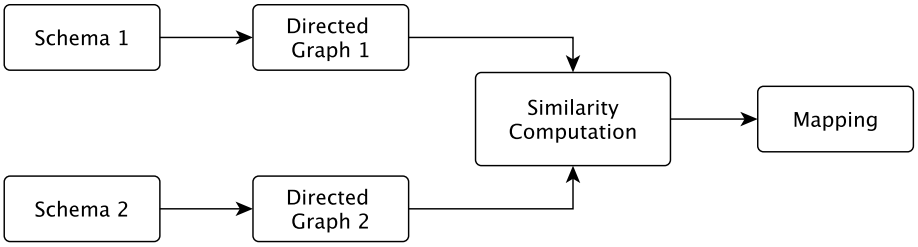


Figure 2.20.: An overview of the similarity flooding approach.

The two schemas or data structures that need to be matched are initially transformed into directed graphs. These graphs are given as input to the similarity computation algorithms (that includes the core similarity flooding algorithm) which calculates the matches between the nodes. Depending on the selected matching goal the algorithm returns the mapping of the nodes.

The matching computation starts by comparing each node of one graph to all the nodes of the second graph using a simple string similarity measure (e.g., the Levenshtein distance). These values are stored in a mapping structure that is called *Initial Map*. At this point the similarity between two nodes has not been propagated to their adjacent nodes. This happens in the next step when the similarity flooding algorithm itself is executed. More specifically, an *iterative* fixpoint computation takes place to add to the values of the initial map the similarity values of their neighbours. Finally, based on the selected mapping goal and constraints, the final mapping is returned. For example, the results will be different if there is a constraint that each of the nodes of the first graph must be mapped to exactly one node of the second graph, or to many.

More details on the similarity flooding algorithm are given in Section 5.4 where it is used as part of a proposed approach for type inference.

2.8.2. Using Similarity Measurements in MDE

In [11], Falleri et al. use the similarity flooding presented above, for the automatic generation of MDE model transformations. More specifically, the source and the target metamodels are translated into directed graphs and are given as input to the matching algorithm in order to identify similarities between their classes. These mappings are then used for the automatic generation of the transformation rules.

As discussed in the presentation of the similarity flooding algorithm, the two schemas (in this context the metamodels) are translated to directed graphs. How-

ever, the way this could happen is not restricted or guided by the similarity flooding algorithm. In their work, Falleri et al. [11] propose 6 different representations, (named *Minimal, Basic, Standard, Full, Flattened and Saturated*) that could be appropriate in the context of MDE and evaluate them though a set of experiments. One of these representations is presented below; the metamodel shown in Figure 2.21 is translated into a directed graph based on the *Minimal* configuration.

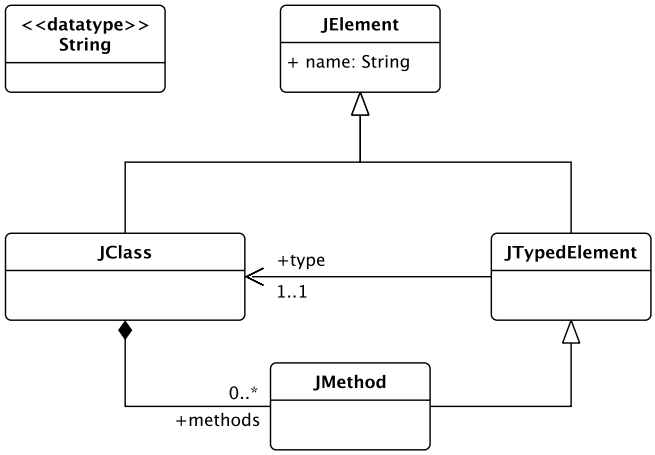


Figure 2.21.: An example metamodel (adapted from [11]).

In this configuration the following information of the metamodels is translated into nodes: classes, non-derived attributes and references, datatypes and enumerations. For example, the metamodel shown in Figure 2.21 will be transformed into the directed graph shown in Figure 2.22.

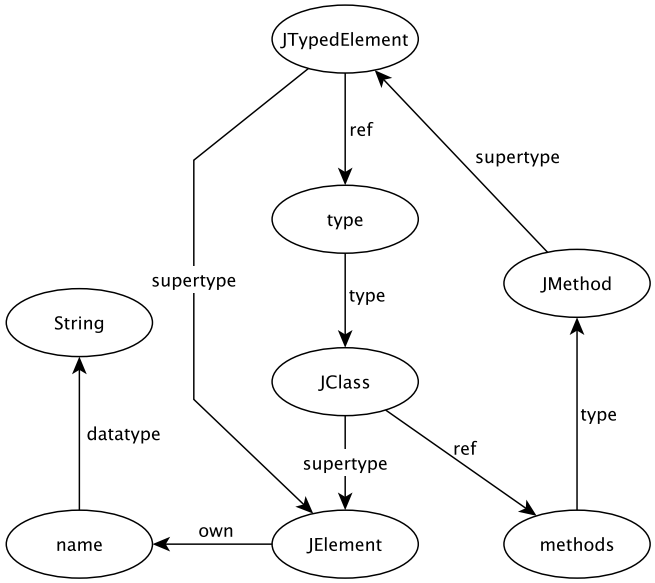


Figure 2.22.: The directed graph of the metamodel shown in Figure 2.21 using the minimal configuration (adapted from [11]).

For edges, the keyword *supertype* is used to name the arc that connects two classes that one inherits from the other. The keyword *own* is used to attach attributes to the classes they belong to and *ref* for the references owned by the class. Finally, *datatype* is used for attaching attributes to their datatypes and *type* to connect each reference name with the class it is typed by.

In Section 5.3, a second configuration (i.e., *flattened*) that is used in our approach to type inference, is presented in detail.

In [12], Grammel et al. propose an approach to trace link generation in MDE based on model matching. The overview of the approach is summarised in Figure 2.23.

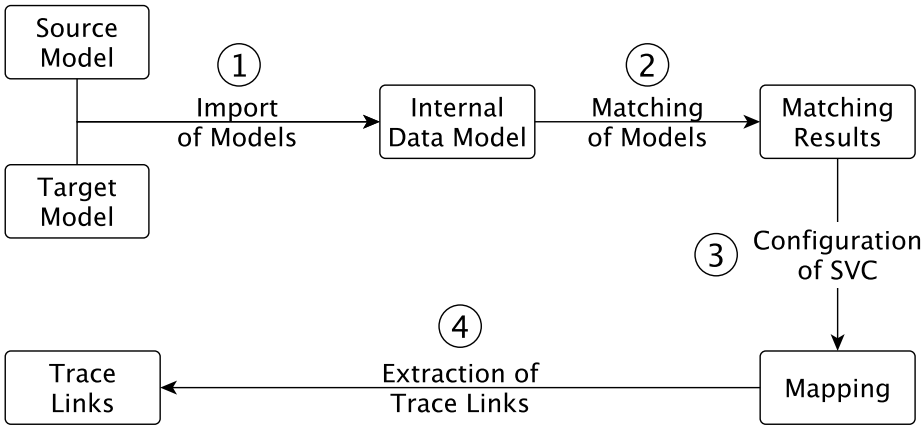
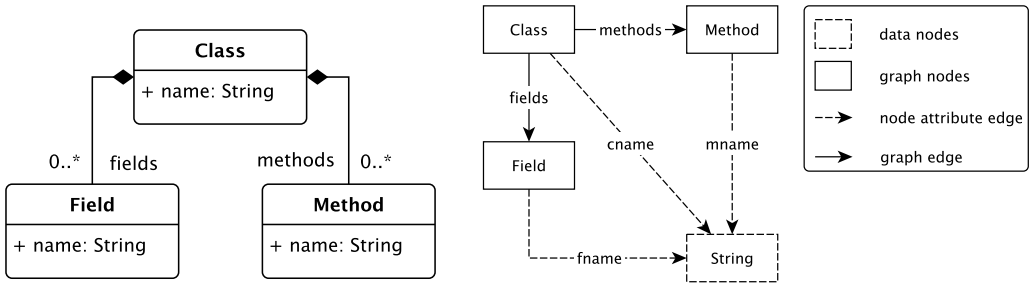


Figure 2.23.: An overview of Grammel et al.' approach [12] to trace link generation (adapted from [12]).

The generation of trace links for the transformation of a source model to the target model starts in step (1) where a custom made importer is used to import and transform the models to a common basis; a graph that is called the *internal data model*. In step (2), matching algorithms are applied to identify similarities between the different concepts of the source and target metamodels. The results of each matching algorithm are stored in a similarity matrix. All the similarity matrices are then arranged to create the *Similarity Value Cube (SVC)*. Using different selection methods (e.g., getting the average of all the similarity values for each pair of elements), the similarity values are then used to create a final mapping between the source and the target model (step (3) in Figure 2.23). Specific heuristics and/or configurations are then used to extract the trace links based on this mapping (step (4)).

For the representation of the source and target models as structured graphs that will allow their comparison, the authors use the *Typed Attributed Graphs* [153] formalism and the E-graphs [153] notation [12]. An example is shown in Figure 2.24 where the example metamodel of Figure 2.24(a) is transformed into the equivalent E-Graph shown in Figure 2.24(b).

For checking the similarity between the model elements, Grammel et al. propose



(a) UML class diagram of example metamodel

(b) Equivalent E-Graph graph (adapted from [12])

Figure 2.24.: An example of how a metamodel is translated to an E-Graph in Grammel et al. [12] model matching approach.

a simple similarity metric. Each element of the source model is checked against each element of the target model. If their labels are identical then a similarity value of 1 is given for this pair [12]. If the label of the source node is a *subset* of the label of the target node, but not identical, then a similarity value of 0.5 is given [12]. Finally, if the labels are different (i.e., neither identical nor the one is subset of the other) a 0 similarity value is given [12]. Users are able to pick among different algorithms for the calculation of the final similarity. The *Attribute Similarity Measure Algorithm* compares each data node from the source model with each node of the target model. The *Connection Similarity Measure Algorithm* propagates the similarity of the children nodes to the parents but in a simplified way comparing to the similarity flooding [13] algorithm presented above.

In our approach we used the similarity flooding [13] algorithm for model matching as it is a widely used technique whose node similarity metric is based on the well established Levenshtein [116] string distance metric, offering a wider granularity than Grammel et al. [12] similarity metric. In addition, using the similarity flooding algorithm in combination with Falleri et al.'s proposal [11] for graph representation, more information from both the metamodel and the example models (e.g., multiplicities, type of reference, etc.) can be taken into account comparing to Grammel et al. [12] model representation.

2.9. Chapter Summary

This chapter discussed the literature related to MDE. The basic terminology and principles of MDE were presented along with a commonly-used architecture for MDE, the Meta Object Facility (MOF). Model management techniques like model transformations were described. Exemplar tools for implementing the MDE operations, like the Eclipse Modelling Framework (EMF) and the Epsilon suite were also presented. A specific branch of MDE approaches, commonly named as *flexible MDE*

or *bottom-up approaches* were introduced. Tools for supporting flexible MDE operations like the Muddles approach, used to develop proofs of concept in this thesis, were described in detail. Finally, literature on metamodel and type inference was presented.

An introduction to the three major fields from which algorithms and tools were used to tackle the research problem and answer the research questions of this thesis was given in the last three sections of this chapter. More specifically, classification algorithms, logic programming techniques and graph similarity were presented.

Although some work was carried out in the way of metamodel inference from example models, there is no significant work done in the direction of type inference for the example models produced as part of flexible MDE approaches. The following three chapters present the work carried out to fill this identified gap.

CHAPTER 3

Type Inference using Classification Algorithms

3.1. Introduction

During the initial phases of flexible MDE approaches, domain experts experiment with example models that are incomplete due to the reasons mentioned in Section 1.1.1. As a result the drawing may be left with untyped nodes. Example models are incomplete and concepts of the domain, which the untyped elements represent, may be ignored. As described in Section 2.5, this is an undesirable effect especially if one takes into consideration that one of the benefits of using flexible MDE approaches is that of having domain experts expressing concepts of the domain; ignoring some of them due to technical errors works in the opposite direction.

This chapter introduces a new approach in type inference of untyped nodes based on machine learning techniques and more specifically *classification algorithms* (*Classification and Regression Trees* and *Random Forests*). The approach presented here can be used in both phases of flexible MDE approaches depicted in Figure 1.1 and discussed in Section 1.1.1 (i.e., having only example models and/or having a draft metamodel inferred). This is the case because the approach relies only on information available in the example models; as a result having a draft metamodel is not a prerequisite. Related to that, the proposed approach differs from those reviewed in Section 2.3, that require the existence of a metamodel for the completion of partial models.

In contrast with the type inference approaches presented in Section 2.4 and summarised in Section 2.5, this approach does not rely on the shapes/icons of the drawn elements to infer the types. A set of characteristics that are not affected by the graphical properties of the drawn elements is proposed. Thus, it can be used in

cases where domain experts are not ready or are unwilling to express the concrete syntax of the concepts they draw in the example models.

A second set of characteristics is also proposed in this approach. This relies on graphical aspects of the drawn models but it takes into account more features and not only their shape (as done in the type inference approaches described in Section 2.4). Additionally to the fact that our approach takes into account more graphical characteristics than those presented in the literature review, it also differs in a second point. The inference is not done using "hardcoded" similarity criteria (i.e., by default, if the shape is the same then the type is the same) but in a more adaptable manner. The approach is trained each time on a specific example model and identifies which graphical characteristic (or characteristics) is (are) the best on this specific example to be used for the classification.

The rest of the chapter is structured as follows. Section 3.2 includes an overview of the approach for both classification algorithms. The two sets of characteristics used to train the classification algorithms are presented in Section 3.3 followed by an analysis of the training performed (Section 3.4). The evaluation of the proposed approach follows in Section 3.5. The results of running the experiments are presented in the same section along with threats to experimental validity. Section 3.6 outlines the limitations of this approach.

Note: The work described in this chapter was published in [15–17]. The version that includes the experiments using the *Random Forests* algorithm was submitted in the *International Journal on Software and Systems Modelling (SoSyM)*[a] and is at present under review. Parts of Section 3.4 are based on text written by Dr. Sam Devlin, a lecturer at the University of York, for the publications mentioned above. His help in understanding the principles of classification algorithms and the guidance in the interpretation of the results are acknowledged.

[a] `http://www.sosym.org/`

3.2. Type Inference

An overview of the approach is given in Figure 3.1. Language engineers initially construct a flexible model using a drawing tool. Each element of this example model can then be annotated with types of the envisioned DSL. However, some nodes may be left untyped. The annotated model is then automatically analysed to extract characteristics of interest, based on the semantics, the abstract and the concrete syntax. A detailed presentation of these characteristics is given in Section 3.3. These characteristics are passed to the classification algorithm of choice (either CART or Random Forests) which performs type inference. We now explain this process in more detail.

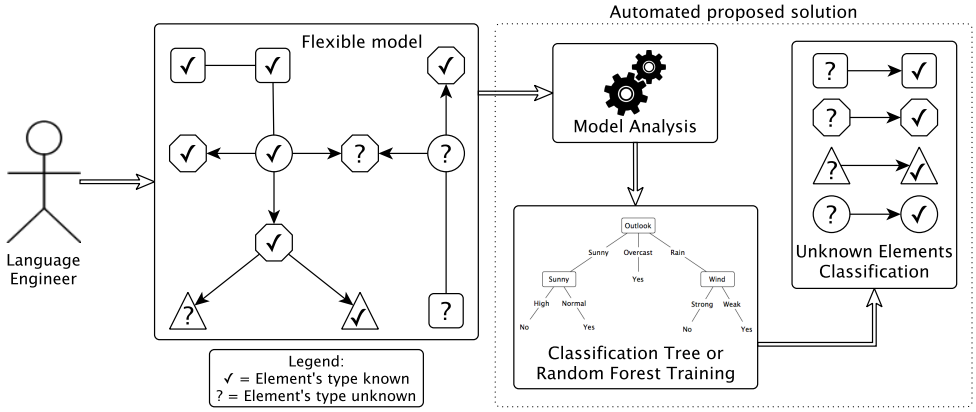


Figure 3.1.: An overview of the proposed approach to type inference using classification algorithms.

A basic requirement for using classification algorithms is having a set of elements whose class (in this chapter the words "class" and "type" will be used interchangeably) is known. This set of known elements is used as input for training the algorithms to be able to classify, or in this context, infer the type, of the unknown elements. In the domain of example models this set of known elements, called *training set* is the nodes that the domain experts have given a type to. The set of the unknown elements that need classification, called the *test set*, are those that were, intentionally or unintentionally, left untyped.

In order to pass the training and the test sets for classification, each element should be represented in a way that it is interpretable by the algorithm. This representation is constructed by extracting specific characteristics from the elements, called *features*. The process of extracting these features is labelled "Model Analysis" in Figure 3.1. A comma-separated string that contains the values of each feature of each element is called a *feature signature* and is unique for each element. At the end of each signature, the type of the element (if known) is also attached. If the type is not known then this field is left empty. As soon as all the elements have their features extracted and placed in a list they can be fed into the classification mechanism to classify the unknown elements.

Both classification algorithms used in this approach get as input the same feature signatures. Before discussing in detail how each of these two works], we present the features selected in this work.

3.3. Feature Signatures

In order to be able to match untyped elements with those that are typed, we need to specify a set of features that describe selected attributes of each element. In this work we focus on two aspects of each example model, the *semantics* and the *concrete* syntax, to extract these characteristics.

3.3.1. Features Based on the Semantics

We use five features to describe example model nodes from the perspective of their semantics. These features were selected because they arguably measure structural and semantic characteristics of the models and are presented below:

- *Number of Attributes (F1):* The number of attributes that the node has.
- *Number of different types of incoming references (F2):* The number of all the types of references that target the specific node. If a node is targeted by more than one references of the same type, only one instance of these is taken into account (unique references).
- *Number of different types of outgoing references (F3):* The number of all the types of references that come from that node. As above, multiple outgoing references of the same type are counted only once.
- *Number of different types of children (F4):* The number of all the unique types that the node contains. Multiple contained elements of the same type are counted once.
- *Number of different types of parents (F5):* The number of all the types that the node is contained in. For environments like the Eclipse Modelling Framework (EMF) and Muddles where one element can be contained in no more than one other node, this value is binary (0: no parents, 1: has parents).

Below we present examples of feature signatures for the elements of the model illustrated in Fig. 3.2, which is an object-diagram-like representation of the muddle of Fig. 2.9.

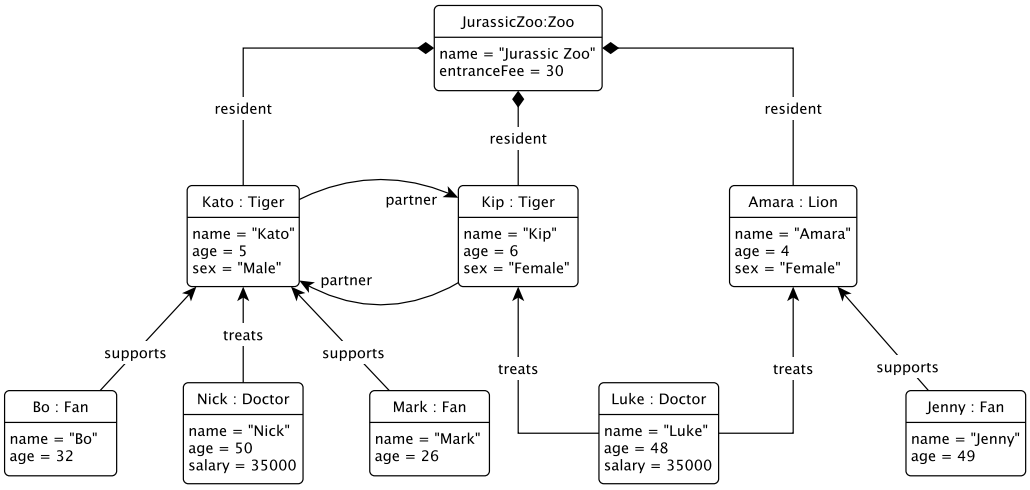


Figure 3.2.: An example model of a zoo configuration.

The feature signature of the node "JurassicZoo : Zoo" is [2,0,0,1,0,Zoo], as it has 2 attributes, no incoming or outgoing references, 3 children which are of the same

type (so 1 unique child) and 0 parents. The last position declares the type of the element, which is useful for training the classification algorithm. Similarly, the feature signature of the node “Kato : Tiger” is [3,3,1,0,1,Tiger] as it has 3 attributes, 3 unique incoming references (supports, partner and treats), 1 unique outgoing reference (partner), 0 children and 1 parent (resident). The class of the node is Tiger and is placed at the end of the signature. Note here that although the element “Kip : Tiger" is also of type Tiger, has not got the “supports” incoming reference instantiated so its signature (i.e., [3,2,1,0,1,Tiger]) is different from the aforementioned “Kato : Tiger" node. This justifies the choice of using a classification algorithm to perform the matching and it is a differentiation point between our approach and those presented in the literature: Classification algorithms, in contrast with simple matching algorithms, do not look for perfect matches based on a specific criterion but are trained to classify elements by using each time those and only those features that are most relevant in the specific set they are trained on, increasing the possibilities of identifying true positives even if two elements have different signatures.

In contrast to the existing approaches in type inference summarised in Section 2.5 these 5 features do not rely on *any* graphical aspect of the example models, and as such they can be used without forcing domain experts to have reached a decision on the concrete syntax of the domain concepts before they are ready to do so.

Algorithm 1 describes how the features based on semantics are calculated from the example models. An empty list that will contain all the signatures is created in line 1. Then the algorithm iterates through all the nodes of the example model (lines 2-38). For each node an empty string is instantiated (line 4) which will host the feature signature of the node. The type of the node (if known) is stored in variable *type* (line 6). If the type is not known then this value is the empty string (line 8). Four empty sets are initialized in line 10 each of which will host the labels of the outgoing and incoming references, children and parents of the node. The number of the attributes that the node has is also stored (line 11). The algorithm iterates through all the edges that are connected with the node. If the node is the source (origin) of the edge then this edge is either an outgoing reference or a containment reference that points to a child. If the edge is a reference its label is added to the set of the outgoing references (lines 15-18). This is done if and only if the label does not already exist in the set (lines 16-18). Otherwise (lines 19-23), it is added to the set of the unique children (if and only if this label does not already exist in the set). If the node under examination is the target node of the edge then the edge is either an incoming reference or an incoming containment from the parent of the node. Following the same logic as described above, the label of the edge is added either in the set of the incoming references (lines 25-28) or the set of the parents (lines 29-33). In line 36, the signature string is constructed by concatenating the number of attributes and the *size* of the sets containing the unique outgoing and incoming references, children and parents followed by the value of th *type* variable.

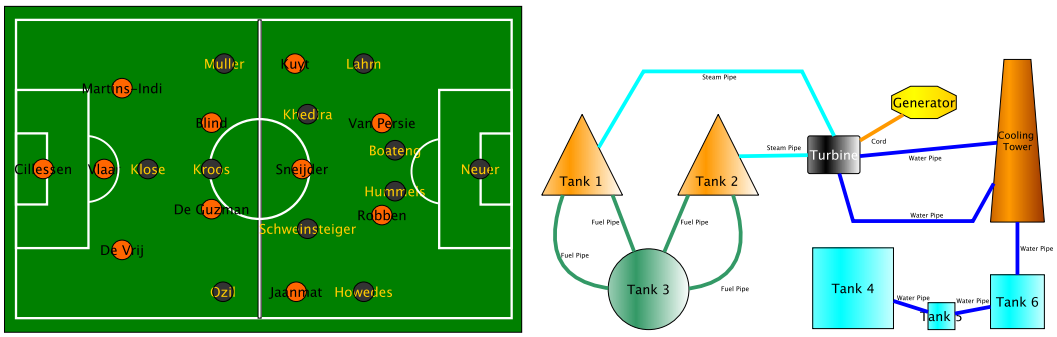
Each value is separated from the other using a ",". The signature is added to the list containing all the signatures for the model (line 37). This process is repeated for all the nodes and eventually the algorithm returns a list containing the signatures for all the nodes in the model (line 39).

Algorithm 1 Computing signatures based on semantic features.

1: $SignaturesList \leftarrow$ Empty list
2: $N \leftarrow$ all nodes in Model
3: **for all** $n \in N$ **do**
4: $signature \leftarrow$ Empty String
5: **if** type of n is known **then**
6: $type \leftarrow$ type of n
7: **else if** type of n in unknown **then**
8: $type \leftarrow$ Empty String
9: **end if**
10: $UniqueOutgoingReferences, UniqueIncomingReferences,$ $UniqueChildren, UniqueParents \leftarrow \emptyset$
11: $NumberOfAttributes \leftarrow$ number of the attributes node n has
12: $E \leftarrow$ all edges connected with n
13: **for all** $r \in E$ **do**
14: **if** n is the origin node of e **then**
15: **if** e is a reference **then**
16: **if** label of n $\notin UniqueOutgoingReferences$ **then**
17: add label of r in $UniqueOutgoingReferences$
18: **end if**
19: **else if** e is a containment **then**
20: **if** label of n $\notin UniqueChildren$ **then**
21: add label of r in $UniqueChildren$
22: **end if**
23: **end if**
24: **else if** n is the target node of e **then**
25: **if** e is a reference **then**
26: **if** label of n $\notin UniqueIncomingReferences$ **then**
27: add label of r in $UniqueIncomingReferences$
28: **end if**
29: **else if** e is a containment **then**
30: **if** label of n $\notin UniqueParents$ **then**
31: add label of r in $UniqueParents$
32: **end if**
33: **end if**
34: **end if**
35: **end for**
36: $signature \leftarrow$ "$NumberOfAttributes, size(UniqueOutgoingReferences),$ $size(UniqueIncomingReferences), size(UniqueChildren),$ $size(UniqueParents), type''$
37: $SignaturesList \leftarrow SignaturesList + signature$
38: **end for**
39: **return** $SignaturesList$

3.3.2. Features Based on Concrete Syntax

In this section we present a second set of features that is based on four graphical characteristics of each node. As presented in [15], graphical properties of the diagram can be used to extract semantics for the elements of a graph. For example, in Figure 3.3(a) the colours of the circle nodes represent the team each footballer plays for while in Figure 3.3(b) the shapes and sizes denote the type of the tank in a nuclear reactor building plan.



(a) Team Formations - Color

(b) Nuclear Reactor - Shapes and Sizes

Figure 3.3.: Colours and shapes are used to define semantics on graphical models.

We use the following proposed properties to construct our feature signatures. The selected characteristics are described below:

- *Shape (F6):* The shape of the node (e.g., rectangle, ellipse, etc.).
- *Colour (F7):* The colour of the filling of the node in HEX (e.g., #FFCC00, etc.).
- *Width (F8):* The width of node expressed in pixels.
- *Height (F9):* The height of the node expressed in pixels.

We now present examples of feature signatures for the elements of the muddle shown in Figure 2.9. The feature signature for the node named “Jurassic Zoo” and annotated with the type “Zoo” is [roundRectangle, #CCFFCC, 418, 196, Zoo]. The first feature represents the shape of the node which in this example is a rounded rectangle, the second the color of the filling which is expressed in hex code. The third and the fourth features are the width and the height dimensions expressed in pixels, respectively. The last part of the signature is the annotated type for this element. Similarly, the feature signature for the node named “Tiger Kato" is [hexagon, #FFCC00, 100, 43, Tiger]. Although the node “Tiger Kip” is also of type “Tiger” its signature will be [hexagon, #FFCC00, 78, 35, Tiger] as it has smaller dimensions in the drawn diagram. It is clear from these two signatures that elements of the same type may or may not have the same signatures. As mentioned in the previous section, this justifies the use of a classification algorithm and not a simple matching

algorithm, as classification algorithms do not look for perfect matches but classify the items by picking each time the features that are more relevant in the set that they are trained on.

Algorithm 2 presents how the features signatures based on the concrete syntax of models are created.

Algorithm 2 Computing concrete signatures.

1: $SignaturesList \leftarrow$ Empty list
2: $N \leftarrow$ set of all nodes in Model
3: **for all** $n \in N$ **do**
4: $id \leftarrow$ n's unique id
5: $Shape(n) \leftarrow getShape(id)$
6: $Color(n) \leftarrow getColor(id)$
7: $Width(n) \leftarrow getWidth(id)$
8: $Shape(n) \leftarrow getHeight(id)$
9: $SignaturesList \leftarrow SignaturesList + Shape(n), Color(n), Width(n), Height(n), Type(n)$
10: **end for**
11: **return** $SignaturesList$

The Muddle implementation presented in Section 2.2.1 does not offer all the needed functionality to extract the desired features. An extension was implemented as part of this work and is described in the following section.

3.3.3. Extending Muddles

The current Muddles metamodel and the implementation of the EMC driver (see Figure 2.7) for muddles discard information regarding graphical and spatial properties of each element. These properties are the: *x* and *y coordinates*, the *width* and *height*, the *shape* and the *colour* of each Muddle Element.

Firstly, we extended the Muddles metamodel to allow muddle elements to hold the above information. The changes are shown in Figure 3.4. In the previous version, the Muddle metamodel (see Figure 2.10) held only the "id" of each element. The "MuddleElement" class was extended to provide holders for the graphical and spatial properties of each element as shown in Figure 3.4.

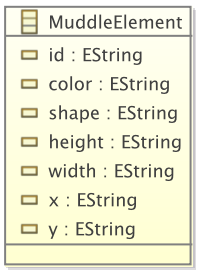


Figure 3.4.: The muddles extension for type inference using concrete syntax properties.

As these attributes are now available to store the information needed for each element, we implemented the required functionality in the EMC Muddles driver to be able to parse the GraphML file (the drawing), retrieve the information from it and store them in these attributes.

3.4. Training and Classification

In our approach, the feature signatures list that contains the signatures of the known elements of the model is the input to the CART and RF algorithms. A trained decision tree or ensemble of trees is produced depending on the algorithm used. These can then be used to classify (identify the type of) the untyped nodes using their feature signatures. A decision tree (or an ensemble of trees in the case of RF) is generated based on the training data that were given to the algorithm (in this context the nodes that are typed). An example decision tree for the Zoo domain described in this chapter is illustrated in Figure 3.5. Internal nodes represent conditions based on features (e.g., number of attributes, unique children, etc.), branches are labelled with "TRUE" or "FALSE" values for the condition of the parent node and leaf nodes represent the final classification given.

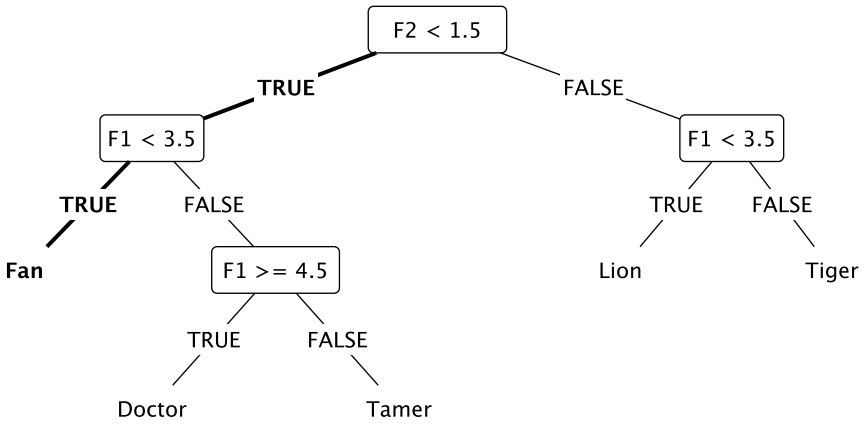


Figure 3.5.: Example decision tree for the features based on semantics. F1 represents the number of attributes of a node and F2 the number of unique incoming references.

To classify a new instance, the algorithm starts at the root of the tree and takes the branch that satisfies the condition of this node. The algorithm continues to process each internal node reached in the same manner until a leaf node is reached where the predicted classification of the new instance is the value of that leaf node. For example, given the tree in Figure 3.5, a new instance with fewer than 1.5 unique incoming references (F2) and less than 3.5 attributes (F1) is classified as "Fan" (path is highlighted in Figure 3.5).

3.5. Experimental Evaluation

In this work we use the *Muddles* approach [4] for running the experiments. The proposed approach can be applied to any example models produced as part of a flexible MDE approach which fulfils the following minimal set of requirements:

- provide a mechanism to extract the *types* of the nodes in the example models
- provide a mechanism to extract the *names* of the *references/containments* in the example models (for the features based on *semantics*)
- provide a mechanism to extract the *shape, color and dimensions* of the nodes in the example models (for the features based on the *concrete syntax*)

The following are the research questions addressed by the experiments presented in the following sections for both the set of features:

- **RQ1:** What is the accuracy of the proposed approach in predicting the types of the untyped nodes using the set of features based on the semantics of the example models?
- **RQ2:** What is the accuracy of the proposed approach in predicting the types of the untyped nodes using the set of features based on the concrete syntax of the example models?

Related to the hypothesis presented in Section 1.2, both these research questions will reveal if the approaches offer an acceptable accuracy to be feasibly applied to a flexible MDE approach. Regarding **RQ1**, more details on the experiment are presented in Section 3.5.1 that follows while in Section 3.5.3 the experiment related to the second research question (**RQ2**) is presented.

3.5.1. Experiment for Features Based on Semantics

In this section, the experiments ran to evaluate the proposed approach using the features based on the *semantics* are presented. An overview of the experiment is shown in Figure 3.6. Details about each step follow.

To evaluate our approach we applied it to 100 randomly generated models, instances of publicly available metamodels that were collected as part of the work presented in [154]. The 10 metamodels, which were selected without having any criterion in mind other than that of having a variation in size (number of concrete meta-classes), are presented in Appendix A. For each of these metamodels we produced 10 random instances using the Crepe model generator tool [155] (step (1) in Figure 3.6). Crepe uses a genetic algorithm to produce random models. In the majority of the cases the Crepe random model generator had the tendency to instantiate all the "0..n" (optional) references and containment relationships appeared in

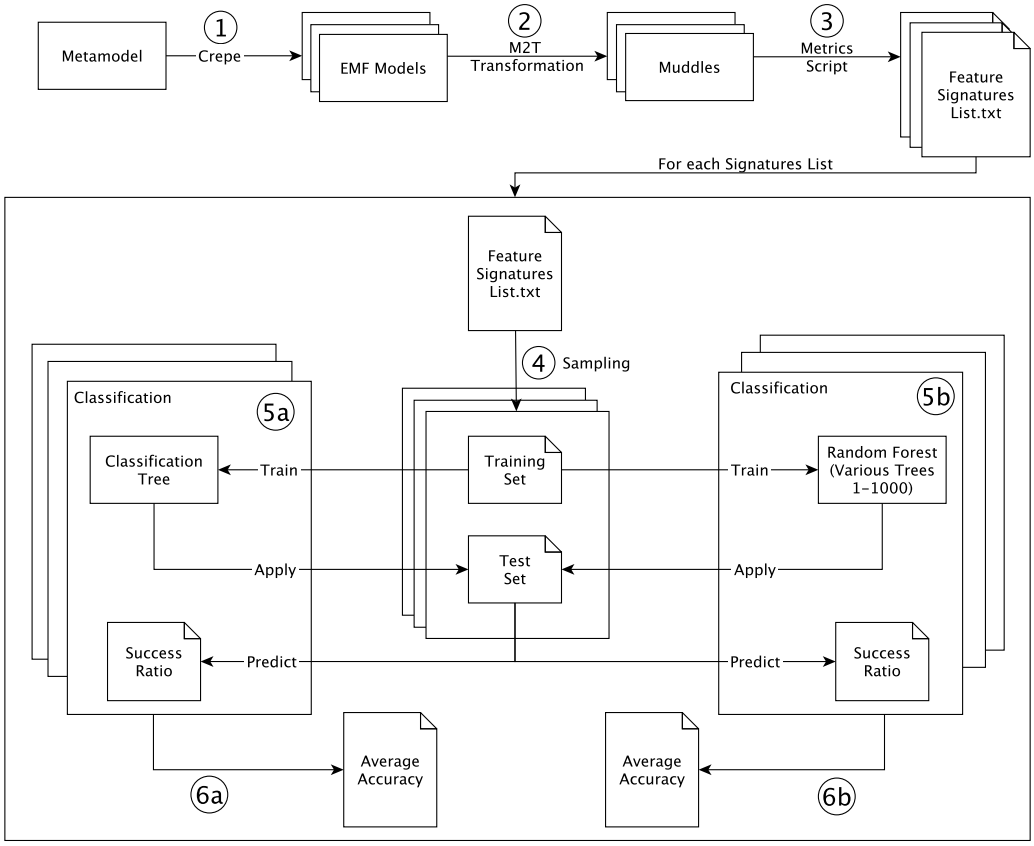


Figure 3.6.: The experimentation process using the features based on semantics.

the metamodels. This might be a bias in the experiment as features F2-F5 are relying on the unique appearance of references and containments: if a type of reference is not instantiated then it is not counted; in contrast if it is instantiated *at least once* then it is counted. This way we create cases like those presented in the example feature signatures in Section 3.3.1 between two "Tiger" nodes that have different signatures due to the absence of the "support" incoming relationship in one of them. To include noise in the signatures in this experiment we decided to create an extra set of random models modifying the generator to be less keen in instantiating the aforementioned relationships. This second set, consisting of 10 models for each of the metamodels, is called "Sparse" set while the original is called "Normal" in this work. This is done by changing a couple of parameters in the Crepe random model generator. Listings 3.1 and 3.2 shows the values of these two parameters.

```
<?xml version="1.0" encoding="ASCII"?>
<conf:Configuration xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:conf="http://crepe.core/meta/conf">
  ...
  <properties name="population.segments.featurePairs.quantity.max" value="40"/>
  <properties name="population.segments.featurePairs.quantity.min" value="20"/>
  ...
</conf:Configuration>
```

Listing 3.1: Crepe configuration parameters for the *Normal* set generation.

```
<?xml version="1.0" encoding="ASCII"?>
<conf:Configuration xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:conf="http://crepe.core/meta/conf">
  ...
  <properties name="population.segments.featurePairs.quantity.max" value="10"/>
  <properties name="population.segments.featurePairs.quantity.min" value="5"/>
  ...
</conf:Configuration>
```

Listing 3.2: Crepe configuration parameters for the *Sparse* set generation.

For our approach, the values of the attributes of each node in the example models were randomly selected, as these do not affect the final feature signature of the element. We discuss threats to validity introduced by using randomly generated models instead of muddles in Section 3.6. Having the models generated, we then transform them into muddles. A model-to-text (M2T) transformation was implemented to transform instances of EMF models to GraphML files that conform to the Muddles metamodel (step (2)). Algorithm 3 describes the transformation.

These two steps ((1) and (2)) could be skipped if there was a corpus of muddles available to test our approach on. However, to our knowledge such a repository of flexible models does not exist. A second approach, that of drawing example muddles on our own to experiment with, was also rejected because it could introduce bias to the process. We decided to follow the 2-step process instead firstly because we would be able to have a bigger number of test muddles and secondly because these muddles are randomly generated and are not biased to fit our approach. Moreover, by introducing the second set of models (i.e., "Sparse") we inject noise in the features signatures that works against the proposed approach.

```
3,1,0,0,0,Project
...
3,1,0,1,0,Project
5,1,2,0,1,Target
...
11,0,0,0,1,Property
...
5,1,2,0,1,Target
3,1,0,0,0,Project
3,1,0,2,0,Project
...
2,0,0,0,1,Task
2,0,0,0,0,Attribute
2,0,0,2,0,Task
...
2,0,0,1,0,TaskParameter
2,0,0,0,1,TaskParameter
2,0,0,3,0,TaskParameter
...
2,0,0,0,1,TaskParameter
```

Listing 3.3: An example of a features signature list based on the semantics of the example model.

After the generation of the muddles from the random models, we extract the feature signature of each node. We implemented a script (see Algorithm 1) that queries each muddle to collect the information needed for each node (i.e., number of attributes, unique outgoing and incoming references, children and parents). By following this process a text file containing a list with signatures is created for each muddle (step (3)). An example is shown in Listing 3.3

At this point the types of all the nodes are known and saved in the feature signatures list. However, in order to test the proposed approach we had to simulate the scenario where some nodes were left untyped. For that reason, each feature signature file is split into two sets (step (4)): the training set which includes all the nodes that their type is known and will be used to train the classification algorithm and the test set which includes all the nodes left untyped and will be used to test the prediction capabilities of the algorithm. Of course, in this experiment all the nodes have types assigned to them as the muddles were generated from typed models. Thus, the simulation of a realistic scenario was done by randomly sampling the feature signatures lists and placing elements in the training and testing sets to simulate a scenario in which some of them were left untyped. It is of interest to evaluate if and how the proportion of untyped nodes affects the prediction capabilities of the approach. For that reason we experimented with 7 different sampling rates, from 30% to 90% (with a step of 10%). A 30% sampling rate means that only 30% of the nodes have a type assigned to them. For the rest of this thesis, the term "sampling rate" will be used to denote the *percentage of typed nodes in the model*. In order to conform to the standard 10-fold cross-validation in the domain of classifications algorithms, described in Section 2.6, we did this random sampling 10 times for each of the sampling rates for each example model, ending with 700 different pairs of training and test sets for each metamodel in the experiment.

The same process was done for both the "Normal" and "Sparse" sets of models in this work. It is important to highlight that each time the classification algorithm was trained using one training set and was tested using the associated test set. After that the algorithm was reset and trained/tested with the next pair of sets. We tested two different classification algorithms, CART (step (5a)) and Random Forests (step (5b)). CART was used initially as it offers more interpretable results than other classification algorithms, making the debugging of the approach easier. RFs were then introduced as, in principle, they offer better accuracy but less interpretable results.

In order to check if the number of trees used for the classification in Random Forest affects the accuracy of the prediction we performed the same experiment for seven different values for the number of trees variable: 1, 5, 10, 50, 250, 500 and 1000.

For classification and regression trees, we used the *rpart* package (version 4.1-

9) [156] that implements the functionality of CART [126] in R [157]. For RF we used the *randomForest* R package (version 4.6-12) [158]. For both algorithms we used the default parameters of each package except the *minbucket* parameter which value was set in 1. This parameter denotes the minimum amount of instances that a leaf node can have.

At the end of each train/test run, the success ratio was calculated (step (6)). The success ratio (also referred as accuracy) is defined as the total number of correct predictions to the total number of untyped nodes. Before presenting the results, a description of the algorithm that is used to transform models to muddles is given.

Models to Muddles Transformation

Step (2) of the experiment (see Figure 3.6) consists of the transformation of EMF models to the flexible modelling approach used as proof of concept in this work, called *Muddles*. Algorithm 3 describes the transformation of EMF models to muddles. The transformation consists of two steps; the first transforms classes in the EMF model to nodes in muddle while the second transforms references to edges.

In Muddles, each node and edge has a unique id. In lines 2-3 of Algorithm 3 the id counter and a mapping that stores the instance class and the id it points to are instantiated. The algorithm iterates through all the model's class instances (lines 5-18) and creates the unique id of the node; in yEd this should be a unique number led by the letter "n" (line 6). The node mapping is updated in line 7. GraphML tags and contents are created to draw the node based on the id (line 8). The property's field that stores the type of the node is populated in line 9. In Muddles, the attributes are stored in a separate yEd property field. Each has a primitive type (e.g., String, Integer, etc.), a name and a value. The algorithm iterates through all the attributes of this instance class (lines 12-14) and constructs the attributes string that holds all the attributes (line 13). Then the attributes' field is populated in line 15. Static content related with graphical properties of the GraphML file is then printed.

In the second step, the edges between the nodes are created. Again, a unique id for each edge should be created, thus an edge counter is instantiated in line 21. The algorithm iterates again through all the model's class instances (lines 23-40). For each reference of each class instance, the string containing the lower and upper bounds is built (line 26). For technical reason in our approach containments are differentiated by attaching the string "VAL" at the end of their name. This is done in lines 28-30. The ids of the source and target node for the specific edge are retrieved and stored in lines 31-33 while the GraphML statement to instantiate the edge is printed in line 34. The fields for the name of the reference and its multiplicities are populated in lines 35-36. Static content related with graphical properties of the GraphML file is then printed.

Algorithm 3 Algorithm to transform EMF models to muddles.

```
1: {Step 1: Create nodes}
2: nodesCounter ← 0
3: nodesMapping ← {}
4: E ← all elements in model
5: for all e ∈ E do
6:    nodeId ← "n" + nodesCounter
7:    nodesMapping ← (e, nodeId)
8:    print <node id="nodeId">
9:    print <data key="d4"><![CDATA[[e.type.name]]]></data>
10:   A ← all attributes of e
11:   attributesString ← ""
12:   for all a ∈ A do
13:      attributeString ← attributeString + a.EType + a.name + a.value
14:   end for
15:   print <data key="d5"><![CDATA[[attributesString]]]></data>
16:   print GraphML static content
17:   nodesCounter + +
18: end for
19:
20: {Step 2: Create edges}
21: edgesCounter ← 0
22: E ← all instances in model
23: for all e ∈ E do
24:   R ← all instantiated references of e
25:   for all r ∈ R do
26:      bound ← "lowerBound" + "upperBound"
27:      name ← r.name
28:      if r is Containment then
29:         name ← name + "VAL"
30:      end if
31:      t ← r's target instance class
32:      tId ← nodesMapping.get(t)
33:      sId ← nodesMapping.get(e)
34:      print <edge id="e[edgesCounter]" source="[sId]" target="[tId]">
35:      print <data key="d12"><![CDATA[[name]]]></data>
36:      print <data key="d16"><![CDATA[outgoing [bound]]]></data>
37:      print GraphML static content
38:      edgesCounter + +
39:   end for
40: end for
```

3.5.2. Results and Discussion

In this section, the results of the experiment using the features signatures based on the *semantics* are presented. As described in Section 3.5.1, the experiment can be split into four sub-experiments based on two variables (see Table 3.1); the type of classification algorithm used in the experiment (CART vs. RF) and the density of

the models ("Normal" set vs "Sparse" set). The results for the CART algorithm are initially presented followed by the results for the RF experiment. A comparison of the results for the CART vs. RF and the "Normal" set vs "Sparse" set experiments follow. The results of the experiments on the importance of the variables used in the feature signatures are also discussed in this section along with a qualitative analysis.

Table 3.1.: The IDs of the experiments.

	Normal	**Sparse**
CART	N-CART	S-CART
Random Forest	N-RF	S-RF

Before going into the presentation and the discussion of the results, a summary of the models used as input in the experiments is given (see Table 3.2). The smallest of the metamodels consists of only 2 types. The largest is the one that describes Wordpress Content Management System websites with 19 different types of classes. On average the test metamodels had 6.5 types with a median of 6. The number of classes exclude the abstract classes in the metamodels as it takes into account only those that can be instantiated in models. For each metamodel, 10 models were generated for the "Normal" set and 10 different for the "Sparse" set. The sizes of the smallest (Min) and the largest (Max) instance model for each metamodel are shown in the respective columns of Table 3.2. The average number of elements for the instances of each metamodel is also given for both sets.

Table 3.2.: Input data summary table for the classification algorithms experiment.

		Normal			**Sparse**		
Model Name	#Types	Min Elements	Max Elements	Average #Elements in instances	Min Elements	Max Elements	Average #Elements in instances
Chess	2	17	26	21.3	18	33	25.5
Conference	4	30	61	42.5	21	48	36.7
Profesor	4	25	36	29.2	19	37	27.7
Zoo	5	47	73	57	22	35	26.2
Ant	6	53	78	65.3	39	77	61.1
Usecase	6	35	71	54.2	42	70	52
Bugzilla	7	21	56	39.9	10	30	21.4
Bibtex	8	56	106	78.8	66	122	92.9
Cobol	11	33	92	63.7	13	62	39.1
Wordpress	19	42	71	58.6	40	88	64.2
Muddle	20	105	105	105	-	-	-

We also provide the values for a muddle drawing we examined. This muddle was part of a side project and was created before commencing this work. It was used to describe requirements of a hotel booking system. We only provide this

muddle as an indication that the performance of the algorithm on the random generated muddles from Ecore metamodels does not differ from that of applying it to real muddles.

Quantitative Analysis for CART

As discussed in Section 3.5.1, 10 random models were instantiated from each of the metamodels. Seven different sampling rates (30%-90%) were applied to each of these models. The classification algorithms were run 10 times (10-fold) for each sampling rate of each model. That sums up to 700 experiments for each of the 10 metamodels (7,000 runs in total). In this work, the exact same experiments were executed on the "Sparse" set. The results are summarised in Tables 3.3 and 3.4 respectively. In the table, we also include the calculated values for the muddle drawing. However, we do not include it in the results' analysis as we only have one instance available in contrast with the 10 random instances of the other metamodels and thus not the variability needed to extract statistical relevant conclusions from it.

Table 3.3.: Results summary table for N-CART

		Average Accuracy for Different Sampling Rates (N-CART)								
Model Name	**#Types**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Chess	2	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
Profesor	4	0.97	0.98	0.98	0.99	0.99	1.00	1.00	0.985	1.00
Zoo	5	0.96	0.98	0.99	1.00	1.00	1.00	1.00	0.990	0.99
Ant	6	0.66	0.69	0.72	0.74	0.74	0.73	0.76	0.723	0.89
Conference	6	0.87	0.91	0.93	0.96	0.96	0.97	0.99	0.940	1.00
Usecase	6	0.74	0.76	0.80	0.81	0.80	0.80	0.78	0.783	0.5
Bugzilla	7	0.46	0.52	0.55	0.55	0.55	0.55	0.55	0.531	0.75
Bibtex	8	0.66	0.67	0.67	0.68	0.66	0.67	0.69	0.673	0.46
Cobol	11	0.59	0.63	0.68	**0.71**	0.75	0.75	0.74	0.692	0.89
Wordpress	19	0.44	0.53	0.63	0.69	0.75	0.77	0.81	0.658	1.00
Muddle	*20*	*0.55*	*0.60*	*0.63*	*0.65*	*0.66*	*0.66*	*0.66*	*0.630*	*0.89*
	Avg.	0.70	0.77	0.79	0.81	0.82	0.82	0.83		
	Cor. 2	-0.88	-0.90	-0.89	-0.87	-0.74	-0.73	-0.72		

In Tables 3.3 and 3.4, the average accuracy is given for all the models of each metamodel. The results are separated into columns based on the sampling rate that was used each time. For instance, the highlighted value **0.71** in Table 3.3 indicates that for the Cobol metamodel, on average (between the 10 random models and 10 random sampling simulations), 71% of the missing types were successfully predicted, using 60% sampling rate. The respective value for the "Sparse" case was 59% (highlighted in Table 3.4)

Considering the raw values, the average successful prediction varied from 53.1%

Table 3.4.: Results summary table for S-CART

		Average Accuracy for Different Sampling Rates (S-CART)								
Model Name	**#Types**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Chess	2	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
Profesor	4	0.85	0.90	0.93	0.95	0.95	0.95	0.94	0.924	0.75
Zoo	5	0.72	0.81	0.89	0.94	0.96	0.97	0.99	0.899	1.00
Ant	6	0.69	0.74	0.75	0.76	0.78	0.78	0.80	0.758	0.96
Conference	6	0.88	0.92	0.94	0.95	0.96	0.97	0.98	0.943	1.00
Usecase	6	0.74	0.78	0.82	0.82	0.83	0.83	0.82	0.806	0.79
Bugzilla	7	0.53	0.57	0.61	0.62	0.63	0.67	0.56	0.599	0.46
Bibtex	8	0.68	0.68	0.68	0.67	0.67	0.68	0.69	0.679	0.04
Cobol	11	0.44	0.51	0.54	**0.59**	0.62	0.64	0.68	0.573	1.00
Wordpress	19	0.41	0.48	0.56	0.62	0.64	0.66	0.66	0.576	0.96
	Avg.	0.66	0.74	0.77	0.79	0.80	0.82	0.81		
	Cor. 2	-0.85	-0.91	-0.90	-0.87	-0.85	-0.91	-0.87		

to 100% for the "Normal" dataset and from 57.3% to 100% for the "Sparse" dataset. By checking the values for the "Normal" experiments, there are some small models (i.e., their metamodel has fewer than 5 types) that the predictive mechanism performs quite well (success ratio of 96% to 100%) even for low samples like 30% and 40%. The same outcome is noticed at the smaller metamodels (fewer than 5 types) of the "Sparse" experiments. In both, the average accuracy drops (some times significantly) for models of more types. However, these values are affected by the fact that in the relatively large metamodels, the prediction scores are lower in small sampling rates, but they keep increasing as the sampling rate (which equals to the amount of knowledge that the CART algorithm is trained with) is increased.

These two observations lead us investigate the following questions:

Cor. 1: How strong is the dependency between the sampling rate and the success score?

Cor. 2: How strong is the dependency between the number of types in a metamodel (size of metamodel) and the success score?

The answers to these questions are given by the values of the correlation measures that are calculated in column named **"Cor. 1"** and the row named **"Cor. 2"** respectively.

We use Spearman's rank correlation coefficient (Spearman's ρ) [159] for all the experiments in this thesis. Spearman's ρ is the non-parametric equivalent to Pearson's product-moment correlation coefficient [160, 161] and we use it as we do not have evidence to assume that our data belong to a normal distribution. For the rest of this thesis, we will interchangeably use the term correlation and correlation coefficient to refer to Spearman's rank correlation coefficient.

As expected, the correlation coefficient values for "Cor. 1" indicate a strong or

perfect dependency for all the metamodels (except BibTeX), for the "Normal" and except two (i.e., BibTeX and Bugzilla) for the "Sparse" experiments. Regarding the second correlation ("Cor. 2") we observe a strong (negative) correlation between the number of types in a metamodel and the success score for all the samples in the "Normal" and the "Sparse" experiments. One can extract the following 2 observations by checking these trends:

1. Fewer types, lead to better results.
2. Lower proportion of untyped nodes results to higher accuracy of the approach.

Both these outcomes are expected. For the first, having fewer types implies that the classification algorithm has to pick the correct type among less candidates. Regarding the second observation, classification algorithms perform based on the training they have received on elements which class is known. The more the known nodes, the more the knowledge that the algorithm is trained on.

Quantitative Analysis for RF

The results of running the experiments using the Random Forest algorithm as the prediction mechanism for both the "Normal" and "Sparse" experiments are summarised in Tables 3.5 and 3.6 respectively. Random Forests typically perform better than individual classifiers like CART [130] and as that testing their performance is of interest. A comparison between the results using CART and RF follows after the presentation of the quantitative results.

The RF mechanism has the same prediction characteristics identified in the CART experiments: for models with few types the accuracy is higher. It drops as the number of types is increased. The same occurs when it comes to the sampling rate: the higher the number of typed elements in the graph, the best the prediction. This behaviour is identical for the "Sparse" set as well.

More specifically, for the "Normal" set the accuracy prediction varies from 84.0% to 100.0% for models with less than 5 types (based on the 50 trees classification). The same values for the "Sparse" set are 66.0% to 100.0%. The lowest prediction value in the "Normal" set is for the Bugzilla metamodel, with 44.8% of the untyped nodes predicted correctly. The lower value in the "Sparse" set is 51% and that is for the Wordpress metamodel.

Table 3.5.: Results summary table for N-RF

			Average Accuracy for Different Sampling Rates (N-RF)								
Model Name	**#Types**	**#Trees**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Chess	2	1	-	0.97	0.96	0.97	0.98	0.97	0.98	0.970	0.41
		5	-	0.99	1.00	0.99	1.00	1.00	0.99	0.995	-0.06
		10	-	1.00	1.00	1.00	1.00	1.00	1.00	0.999	0.78
		50	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
		250	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
		500	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
		1000	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
Profesor	4	1	0.85	0.87	0.91	0.90	0.91	0.92	0.93	0.898	0.93
		5	0.93	0.95	0.94	0.95	0.96	0.96	0.95	0.948	0.86
		10	0.93	0.96	0.96	0.95	0.96	0.96	0.98	0.958	0.82
		50	0.95	0.97	0.97	0.97	0.97	0.97	0.99	0.971	0.75
		250	0.95	0.97	0.96	0.98	0.98	0.98	0.99	0.974	0.93
		500	0.95	0.98	0.97	0.98	0.98	0.98	0.99	0.975	0.93
		1000	0.95	0.98	0.96	0.98	0.98	0.98	0.99	0.975	0.93
Zoo	5	1	0.59	0.61	0.60	0.69	0.70	0.70	0.72	0.658	0.92
		5	0.74	0.78	0.82	0.84	0.84	0.85	0.87	0.819	0.96
		10	0.77	0.83	0.87	0.87	0.93	0.92	0.94	0.875	0.96
		50	0.84	0.90	0.92	0.95	0.97	0.98	0.98	0.932	0.96
		250	0.83	0.91	0.94	0.96	0.98	0.98	0.99	0.942	1.00
		500	0.85	0.91	0.94	0.96	0.98	0.99	0.98	0.944	0.96
		1000	0.85	0.91	0.94	0.96	0.98	0.99	0.98	0.944	0.96
Ant	6	1	0.52	0.58	0.61	0.66	0.67	0.65	0.67	0.625	0.89
		5	0.59	0.66	0.68	0.71	0.72	0.70	0.72	0.682	0.89
		10	0.62	0.67	0.70	0.72	0.73	0.72	0.74	0.701	0.96
		50	0.65	0.68	0.71	0.73	0.74	0.73	0.75	0.713	0.89
		250	0.66	0.69	0.71	0.73	0.74	0.73	0.75	0.717	0.89
		500	0.66	0.69	0.72	0.73	0.74	0.73	0.77	0.718	0.89
		1000	0.66	0.69	0.71	0.73	0.74	0.73	0.77	0.718	0.89
Conference	6	1	0.68	0.74	0.76	0.79	0.79	0.81	0.84	0.771	0.96
		5	0.75	0.80	0.81	0.85	0.84	0.87	0.86	0.824	0.93
		10	0.76	0.81	0.82	0.86	0.85	0.85	0.87	0.831	0.89
		50	0.79	0.84	0.85	0.88	0.86	0.87	0.92	0.858	0.89
		250	0.78	0.83	0.86	0.87	0.87	0.87	0.92	0.857	0.89
		500	0.78	0.83	0.87	0.87	0.87	0.88	0.92	0.860	1.00
		1000	0.78	0.83	0.86	0.87	0.88	0.88	0.92	0.859	1.00

Table 3.5.: Results summary table for N-RF

			Average Accuracy for Different Sampling Rates (N-RF)								
Model Name	**#Types**	**#Trees**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Usecase	6	1	0.59	0.63	0.67	0.67	0.66	0.70	0.73	0.663	0.91
		5	0.67	0.71	0.74	0.76	0.76	0.77	0.74	0.736	0.61
		10	0.70	0.73	0.76	0.78	0.78	0.79	0.78	0.759	0.75
		50	0.72	0.74	0.78	0.79	0.79	0.79	0.78	0.771	0.54
		250	0.73	0.75	0.78	0.79	0.79	0.79	0.78	0.772	0.75
		500	0.73	0.75	0.78	0.79	0.79	0.79	0.78	0.774	0.75
		1000	0.73	0.74	0.78	0.79	0.79	0.79	0.78	0.774	0.68
Bugzilla	7	1	0.36	0.37	0.39	0.40	0.38	0.42	0.38	0.386	0.54
		5	0.41	0.43	0.44	0.43	0.44	0.44	0.41	0.427	0.43
		10	0.42	0.43	0.44	0.44	0.42	0.42	0.42	0.430	-0.57
		50	0.44	0.44	0.47	0.46	0.44	0.45	0.46	0.448	0.43
		250	0.44	0.45	0.47	0.47	0.43	0.43	0.46	0.451	-0.18
		500	0.44	0.45	0.48	0.47	0.44	0.43	0.45	0.451	-0.18
		1000	0.44	0.45	0.47	0.47	0.43	0.44	0.43	0.447	-0.57
Bibtex	8	1	0.57	0.58	0.57	0.57	0.59	0.56	0.57	0.572	-0.18
		5	0.61	0.61	0.62	0.61	0.61	0.58	0.61	0.606	-0.43
		10	0.62	0.62	0.63	0.63	0.63	0.60	0.63	0.623	0.04
		50	0.64	0.64	0.65	0.64	0.64	0.62	0.65	0.640	0.21
		250	0.64	0.64	0.65	0.64	0.64	0.62	0.66	0.641	-0.11
		500	0.63	0.64	0.65	0.64	0.63	0.62	0.65	0.640	0.14
		1000	0.63	0.65	0.65	0.65	0.64	0.63	0.65	0.641	-0.04
Cobol	11	1	0.45	0.48	0.52	0.57	0.58	0.58	0.59	0.539	0.93
		5	0.52	0.56	0.61	0.65	0.66	0.66	0.70	0.622	0.96
		10	0.55	0.60	0.63	0.67	0.70	0.69	0.74	0.654	0.96
		50	0.58	0.64	0.67	0.70	0.71	0.72	0.76	0.683	1.00
		250	0.59	0.65	0.67	0.71	0.72	0.72	0.75	0.685	0.96
		500	0.59	0.65	0.67	0.71	0.72	0.72	0.75	0.688	1.00
		1000	0.59	0.65	0.67	0.71	0.72	0.73	0.75	0.687	1.00
Wordpress	19	1	0.25	0.32	0.35	0.39	0.42	0.44	0.51	0.384	0.99
		5	0.35	0.42	0.48	0.54	0.56	0.59	0.66	0.515	1.00
		10	0.38	0.47	0.54	0.59	0.62	0.64	0.70	0.563	1.00
		50	0.44	0.53	0.59	0.65	0.68	0.68	0.77	0.618	0.96
		250	0.44	0.53	0.61	0.67	0.70	0.71	0.80	0.637	1.00
		500	0.44	0.53	0.61	0.68	0.70	0.71	0.80	0.638	1.00
		1000	0.44	0.53	0.61	0.67	0.71	0.72	0.79	0.638	1.00

Table 3.5.: Results summary table for N-RF

			Average Accuracy for Different Sampling Rates (N-RF)								
Model Name	**#Types**	**#Trees**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Muddle	*20*	*1*	*0.44*	*0.45*	*0.45*	*0.48*	*0.46*	*0.51*	*0.52*	*0.473*	*0.91*
		5	*0.46*	*0.50*	*0.48*	*0.52*	*0.52*	*0.56*	*0.54*	*0.511*	*0.89*
		10	*0.48*	*0.49*	*0.52*	*0.54*	*0.55*	*0.58*	*0.53*	*0.527*	*0.85*
		50	*0.48*	*0.54*	*0.55*	*0.56*	*0.58*	*0.60*	*0.58*	*0.556*	*0.89*
		250	*0.49*	*0.56*	*0.55*	*0.58*	*0.60*	*0.60*	*0.53*	*0.559*	*0.80*
		500	*0.50*	*0.56*	*0.55*	*0.58*	*0.60*	*0.60*	*0.52*	*0.559*	*0.99*
		1000	*0.50*	*0.56*	*0.55*	*0.58*	*0.59*	*0.60*	*0.56*	*0.563*	*0.91*
		Avg.	0.64	0.71	0.73	0.75	0.76	0.76	0.78		
		Cor. 2[1]	-0.88	-0.91	-0.90	-0.88	-0.88	-0.87	-0.75		

Table 3.6.: Results summary table for S-RF

			Average Accuracy for Different Sampling Rates (S-RF)								
Model Name	**#Types**	**#Trees**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Chess	2	1	-	0.98	0.97	0.95	0.98	0.99	0.94	0.969	-0.46
		5	-	1.00	1.00	1.00	1.00	1.00	1.00	0.998	-0.21
		10	-	1.00	1.00	1.00	1.00	1.00	1.00	0.999	0.65
		50	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
		250	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
		500	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
		1000	-	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
Profesor	4	1	0.75	0.77	0.84	0.86	0.87	0.87	0.87	0.830	0.89
		5	0.79	0.86	0.91	0.92	0.92	0.92	0.91	0.890	0.68
		10	0.83	0.88	0.92	0.93	0.92	0.93	0.93	0.906	0.82
		50	0.87	0.90	0.92	0.94	0.94	0.95	0.95	0.925	0.96
		250	0.88	0.90	0.93	0.93	0.94	0.95	0.94	0.924	0.96
		500	0.88	0.91	0.93	0.94	0.94	0.95	0.95	0.926	0.96
		1000	0.88	0.90	0.92	0.94	0.94	0.95	0.95	0.926	1.00

[1]Calculated based on the 50 trees values

Table 3.6.: Results summary table for S-RF

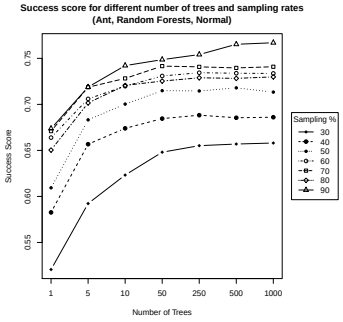
			Average Accuracy for Different Sampling Rates (S-RF)								
Model Name	**#Types**	**#Trees**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Zoo	5	1	0.48	0.50	0.58	0.61	0.64	0.60	0.65	0.578	0.89
		5	0.57	0.61	0.67	0.70	0.75	0.74	0.77	0.689	0.96
		10	0.62	0.65	0.72	0.74	0.79	0.79	0.81	0.731	1.00
		50	0.66	0.71	0.76	0.80	0.84	0.83	0.88	0.780	0.96
		250	0.66	0.72	0.76	0.80	0.84	0.84	0.90	0.786	0.96
		500	0.66	0.71	0.77	0.80	0.83	0.83	0.89	0.784	0.96
		1000	0.67	0.71	0.77	0.80	0.83	0.83	0.89	0.786	0.96
Ant	6	1	0.48	0.53	0.57	0.60	0.61	0.62	0.62	0.575	0.93
		5	0.56	0.63	0.66	0.68	0.69	0.69	0.73	0.662	1.00
		10	0.60	0.65	0.69	0.70	0.72	0.72	0.74	0.689	1.00
		50	0.64	0.68	0.71	0.72	0.74	0.74	0.76	0.712	1.00
		250	0.64	0.68	0.71	0.72	0.74	0.75	0.76	0.716	1.00
		500	0.64	0.68	0.71	0.72	0.74	0.75	0.76	0.714	1.00
		1000	0.64	0.68	0.71	0.72	0.74	0.74	0.77	0.716	1.00
Conference	6	1	0.76	0.79	0.79	0.82	0.83	0.84	0.83	0.809	0.92
		5	0.82	0.84	0.85	0.85	0.87	0.85	0.88	0.852	0.96
		10	0.84	0.84	0.87	0.87	0.88	0.86	0.88	0.862	0.75
		50	0.85	0.85	0.88	0.88	0.88	0.88	0.88	0.871	0.68
		250	0.85	0.85	0.88	0.88	0.89	0.88	0.88	0.873	0.71
		500	0.85	0.86	0.88	0.88	0.88	0.88	0.89	0.875	0.96
		1000	0.85	0.86	0.88	0.88	0.89	0.88	0.88	0.874	0.89
Usecase	6	1	0.56	0.62	0.67	0.69	0.73	0.72	0.73	0.674	0.92
		5	0.66	0.69	0.75	0.76	0.81	0.80	0.79	0.751	0.86
		10	0.69	0.71	0.76	0.79	0.82	0.81	0.79	0.767	0.86
		50	0.72	0.75	0.79	0.79	0.83	0.82	0.81	0.787	0.86
		250	0.73	0.76	0.79	0.81	0.83	0.82	0.82	0.794	0.86
		500	0.73	0.76	0.80	0.81	0.83	0.82	0.82	0.794	0.89
		1000	0.73	0.76	0.79	0.81	0.83	0.82	0.82	0.793	0.86
Bugzilla	7	1	0.47	0.48	0.53	0.53	0.53	0.58	0.48	0.513	0.41
		5	0.52	0.53	0.58	0.56	0.59	0.60	0.54	0.560	0.61
		10	0.53	0.56	0.59	0.57	0.59	0.62	0.53	0.571	0.43
		50	0.55	0.57	0.60	0.60	0.59	0.65	0.55	0.588	0.32
		250	0.55	0.58	0.60	0.60	0.59	0.64	0.55	0.588	0.11
		500	0.55	0.57	0.60	0.61	0.59	0.64	0.55	0.588	0.36
		1000	0.55	0.57	0.59	0.61	0.59	0.64	0.55	0.588	0.21

Table 3.6.: Results summary table for S-RF

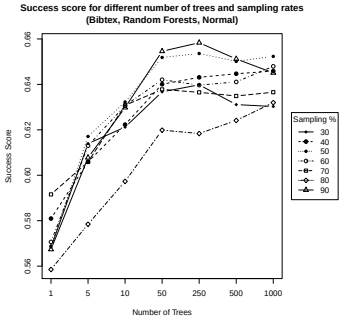
			Average Accuracy for Different Sampling Rates (S-RF)								
Model Name	**#Types**	**#Trees**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Bibtex	8	1	0.58	0.60	0.59	0.59	0.59	0.59	0.63	0.596	0.62
		5	0.62	0.62	0.62	0.61	0.61	0.61	0.65	0.622	0.00
		10	0.63	0.63	0.64	0.63	0.62	0.63	0.67	0.635	0.43
		50	0.63	0.65	0.64	0.63	0.63	0.64	0.66	0.641	0.29
		250	0.64	0.65	0.64	0.63	0.63	0.64	0.66	0.641	0.00
		500	0.64	0.65	0.64	0.63	0.63	0.64	0.66	0.642	-0.11
		1000	0.64	0.65	0.64	0.63	0.63	0.64	0.67	0.642	-0.11
Cobol	11	1	0.32	0.39	0.39	0.45	0.46	0.48	0.47	0.424	0.93
		5	0.39	0.45	0.48	0.52	0.55	0.57	0.55	0.501	0.96
		10	0.42	0.48	0.51	0.55	0.58	0.59	0.56	0.528	0.89
		50	0.45	0.51	0.53	0.58	0.61	0.63	0.63	0.562	1.00
		250	0.45	0.51	0.55	0.58	0.61	0.64	0.63	0.567	0.96
		500	0.45	0.51	0.54	0.58	0.62	0.65	0.63	0.570	0.96
		1000	0.45	0.51	0.55	0.58	0.62	0.65	0.64	0.573	0.96
Wordpress	19	1	0.20	0.25	0.31	0.33	0.36	0.38	0.38	0.314	0.95
		5	0.29	0.35	0.40	0.42	0.45	0.47	0.49	0.410	1.00
		10	0.33	0.39	0.44	0.47	0.51	0.53	0.56	0.462	1.00
		50	0.38	0.43	0.49	0.53	0.56	0.59	0.59	0.510	0.96
		250	0.39	0.44	0.50	0.55	0.57	0.60	0.60	0.520	0.96
		500	0.39	0.44	0.50	0.55	0.56	0.60	0.60	0.521	0.96
		1000	0.39	0.44	0.50	0.55	0.56	0.60	0.60	0.520	1.00
		Avg.	0.61	0.68	0.71	0.72	0.74	0.75	0.75		
		Cor. 2[2]	-0.90	-0.93	-0.93	-0.95	-0.93	-0.96	-0.91		

From both tables is clear that the number of trees affects the accuracy: more trees lead to better accuracy. There is a point after which there is no improvement in the accuracy. To make this more clear we present the accuracy line graphs based on the number of trees used, for all the metamodels for both the "Normal" and "Sparse" sets in Figures 3.7 and 3.8, respectively. From $1 - 50$ trees there is a rapid increase in the accuracy. After 50 trees the accuracy does not improve but the computation required continues to increase. This pattern of diminishing returns is typical when increasing the number of trees in a RF. Given the common occurrence of convergence in the accuracy across multiple metamodels, these results would suggest that if deploying RF as the classification algorithm for type inference, 50 **is a suitable parameter setting** using the set of the five features that are based on the semantics.

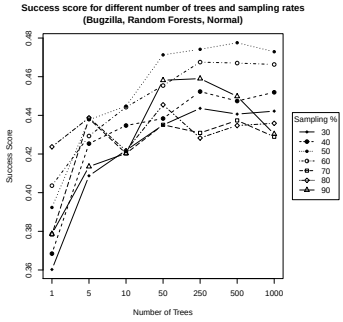
[2]Calculated based on the 50 trees values



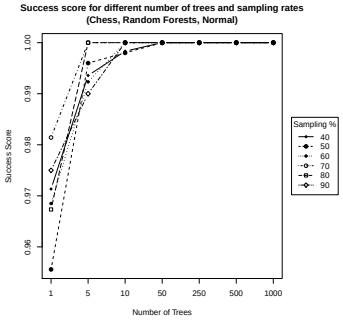
(a) Ant - Normal



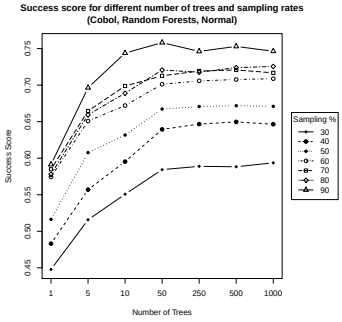
(b) Bibtex - Normal



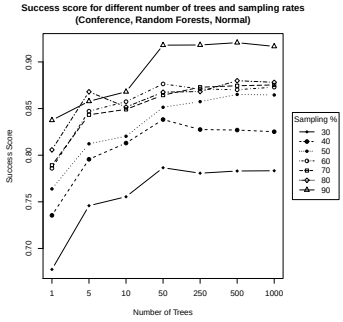
(c) Bugzilla - Normal



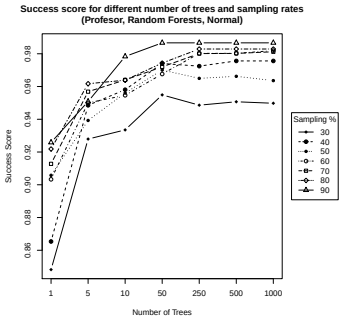
(d) Chess - Normal



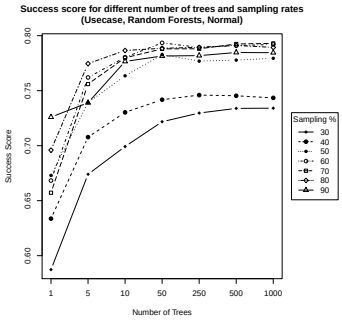
(e) Cobol - Normal



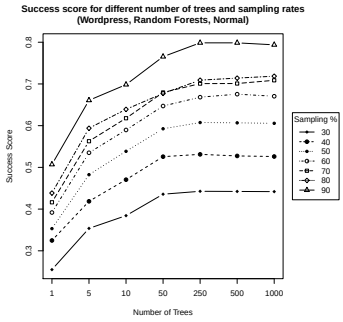
(f) Conference - Normal



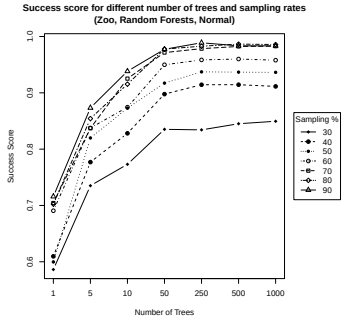
(g) Profesor - Normal



(h) Usecase - Normal

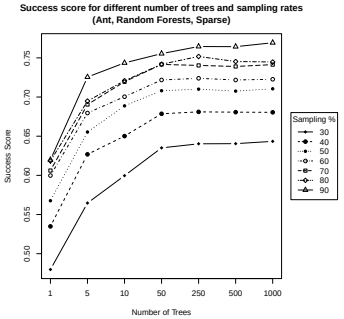


(i) Wordpress - Normal

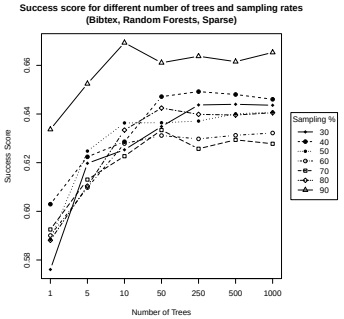


(j) Zoo - Normal

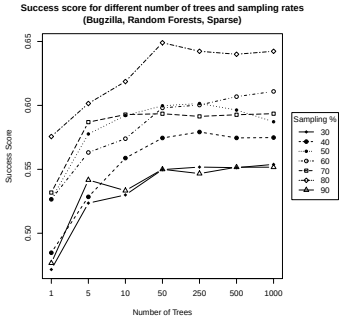
Figure 3.7.: Accuracy for different sampling rates and number of trees (“Normal”, Random Forests).



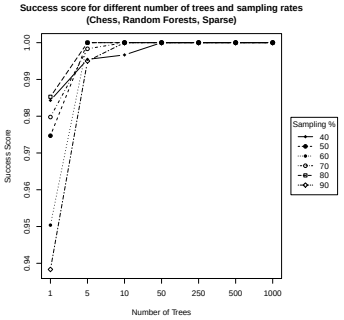
(a) Ant - Sparse



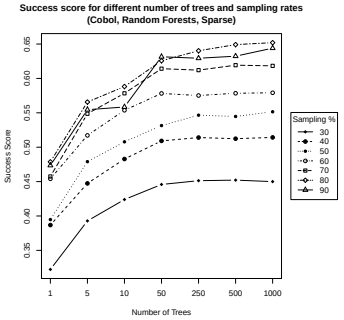
(b) Bibtex - Sparse



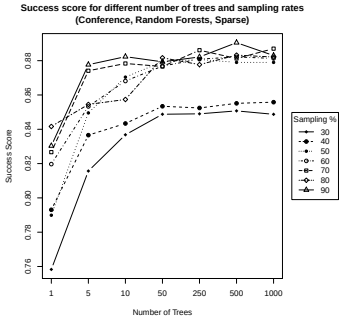
(c) Bugzilla - Sparse



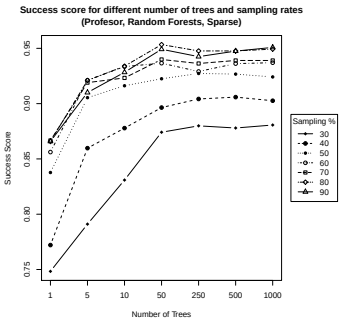
(d) Chess - Sparse



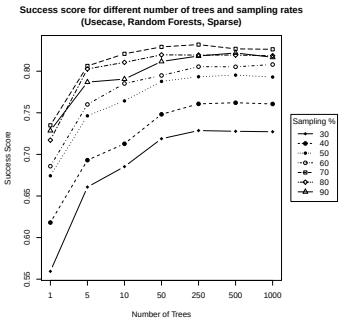
(e) Cobol - Sparse



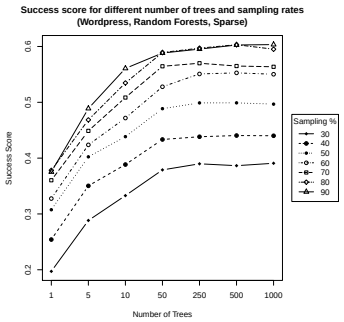
(f) Conference - Sparse



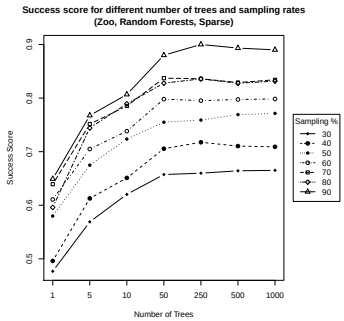
(g) Profesor - Sparse



(h) Usecase - Sparse



(i) Wordpress - Sparse



(j) Zoo - Sparse

Figure 3.8.: Accuracy for different sampling rates and number of trees (“Sparse”, Random Forests).

Finally, regarding the correlation coefficients **"Cor. 1"** and **"Cor. 2"** defined in the CART experiment section, the calculated values for the RF experiment are given in the columns labelled "Cor. 1" and "Cor. 2" respectively for both the "Normal" and "Sparse" experiment (see Tables 3.5 and 3.6 respectively). We observed the same behaviour with the CART experiment, as summarised below:

1. Fewer types, lead to better results.
2. Lower proportion of untyped nodes results to higher accuracy of the approach.

The results for both algorithms answer the research question, related with the set of features based on the semantics of example models **(RQ1)**, set at the beginning of this experiment. For both the approaches, regardless the density set ("Normal" or "Sparse"), the prediction accuracy is high and acceptable for almost all the scenarios. This is not the case with some exceptional metamodels (the reasons for this behaviour are investigated in the qualitative analysis section of this chapter) and scenarios where relatively large metamodels have less than 50% of the nodes typed.

Comparison

Normal vs Sparse

In this experiment, we adjusted the random model generator to produce less dense models by reducing the number of times that optional references (references with multiplicities set to "0..n") are instantiated. As discussed in Section 3.5.1 this behaviour affects features F2-F5 (i.e., the number of unique incoming and outgoing references, unique children and parents) and this way we try to minimise any bias in favour of the approach. We summarise the results of the comparison in Table 3.7. In the table, the 2nd column presents the number of optional references in each model out of the total references. The 3rd column shows the trend in the prediction accuracy between the "Normal" and the "Sparse" set in CART. For example, in the Ant metamodel the average accuracy was higher ($\nearrow$) in the "Sparse" set than the "Normal" set. In the same manner, the last column hosts the trend for the RF equivalent.

As one can see from the table, in 9 cases the accuracy was not affected at all, while in 8 others the accuracy dropped, sometimes significantly. The were 3 cases were the accuracy was increased. There are models which have all their relationships marked as optional (e.g., Bugzilla, Usecase, Wordpress) and have different trends in their prediction scores ($\nearrow$, $\sim$, $\searrow$ respectively). That does not allow us to reach a definitive conclusion if the density of the models affects the prediction accuracy.

In addition, we believe that the following, *unavoidable*, side effect of the "Sparse"

Table 3.7.: Accuracy difference trends between "Normal" and "Sparse" experiments.

Model Name	# of optional relationships out of total	Difference from N-CART to S-CART	Difference from N-RF to S-RF (50 trees)
Ant	6/6	↗	∼
Bibtex	0/1	∼	∼
Bugzilla	6/6	↗	↗
Chess	1/1	∼	∼
Cobol	6/13	↘	↘
Conference	2/6	∼	∼
Profesor	3/5	↘	↘
Usecase	7/7	∼	∼
Wordpress	32/33	↘	↘
Zoo	2/3	↘	↘

model generation also affects the results: The elements that are created only through one optional relationship will be instantiated fewer times in the "Sparse" experiment. If these elements are in turn responsible to instantiate other types that are only hosted by them, then the latter have significantly decreased chances of appearing in the set. For example, in a naive example model that could potentially conform to the metamodel shown in Figure 3.9 with 3 types (Grandparent, Parent, Children) with 2 optional relationships, the "Sparse" set will have fewer "Parent" nodes and even fewer (maybe 0) "Children" nodes than the "Normal" set. In the scenario, where the "Children" node is a distinctive one, and the prediction algorithm has high accuracy in predicting this specific type, the absence of this type in the model will be the reason why the total average accuracy is dropped and not the fact that the model is less dense (and thus because of the fact that the feature signatures of the "Grandparent" and "Parent" nodes were affected).

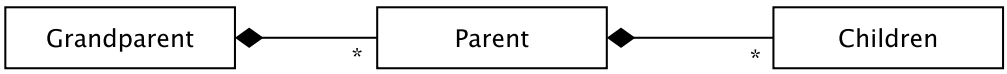


Figure 3.9.: A metamodel from which instances of "Children" nodes may never be instantiated if the random model generator forces optional composition relationships to be instantiated less frequently ("Sparse" scenario).

Finally, CART and RF, dynamically pick each time the feature that is distinctive among the different types. Thus, it is possible that between two types that have one of their features affected by the noise injection (e.g., F2), the algorithm will pick any other from the remaining four features (i.e., F1, F3, F4 or F5) to differentiate these types. This way, the noise injection has no effect in the accuracy of the prediction mechanism.

CART vs RF

By comparing the accuracy of RF and CART given the same metamodel and sampling rate, our results show that the accuracy of our implementation of RF is at best equivalent to CART and often worse (see Table 3.8 for the trends in the average scores for each metamodel). This was an unexpected outcome for the study, given that RF typically outperforms CART and more generally that ensembles of classifiers typically outperform individual classifiers [130].

Table 3.8.: Accuracy difference trends between CART and RF.

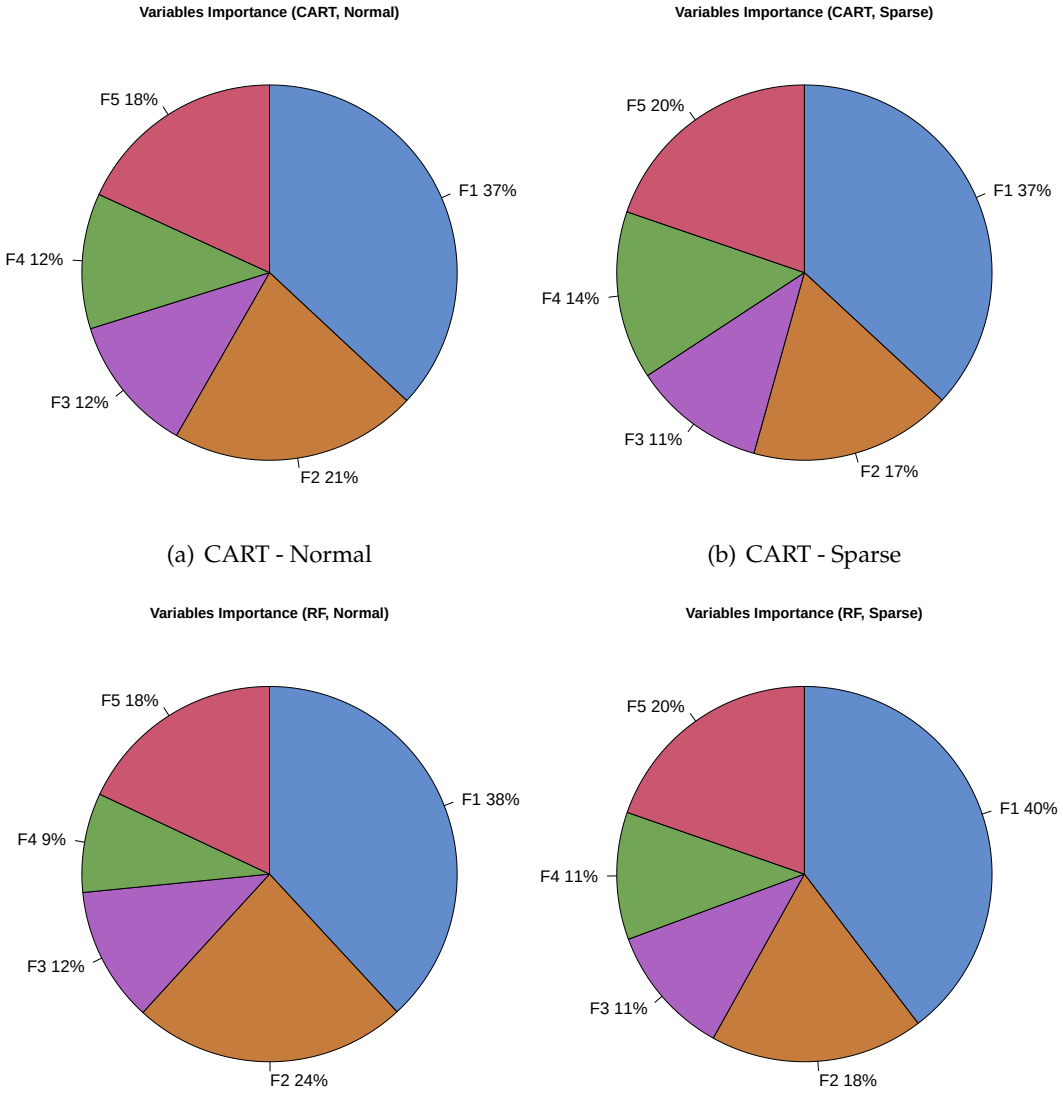
Model Name	Difference from N-CART to N-RF	Difference from S-CART to S-RF (50 trees)
Ant	$\sim$	$\searrow$
Bibtex	$\searrow$	$\searrow$
Bugzilla	$\searrow$	$\sim$
Chess	$\sim$	$\sim$
Cobol	$\sim$	$\sim$
Conference	$\searrow$	$\searrow$
Profesor	$\sim$	$\sim$
Usecase	$\sim$	$\sim$
Wordpress	$\searrow$	$\searrow$
Zoo	$\searrow$	$\searrow$

Our possible explanation is that this result has occurred due to the reduction in features used by each tree in the RF. By default, the randomForest package used chooses $\sqrt{p}$ features where p is the number of features in the input. Given that our feature signature contains only five features, the package chose only two features to train each tree in the ensemble potentially harming the accuracy achievable by the resultant models. Furthermore, considering the high accuracy of CART on almost all metamodels (particularly those with 5 or less types) this may also have occurred because CART is able to achieve an upper bound on the accuracy achievable. Therefore, the extra predictive ability of RF may become more apparent if we increased the number of features in our feature signature, removed the sampling of features used by each tree in a RF or increased the complexity of the metamodels by including more types. However, assuming the metamodels tested are representative of those that this method may be applied to, we conclude that for this application to type inference **CART is** both **sufficiently accurate** and preferable to more complex classification algorithms due to its interpretable output.

Variables Importance

The importance of each variable is a value that signifies how important that variable is in classifying the elements of the test set. In experiments with large sets of features (variables) such a process is important as it helps eliminate those that

do not play a significant (or any) role in creating the split decision nodes in each tree and thus reduce the time needed for training. As this is the first time, to our knowledge, that classification algorithms are used for type inference, it is of interest to assess if any of the five proposed features based on the semantics is redundant and/or which features are more important in this domain. To measure the importance of different variables in the experiments we used the built-in functions available in the same packages (rpart and randomForest) used for the classification.



(a) CART - Normal

(b) CART - Sparse

(c) RF - Normal

(d) RF - Sparse

Figure 3.10.: Variables importance of features based on semantics. F1 represents the number of attributes, F2 and F3 represent the number of unique incoming and outgoing references respectively and F4 and F5 the number of unique children and parents respectively.

A summary of the results for the four experiments is shown in the pie charts of Figure 3.10. These values are the average importance of different variables for all the runs of the experiments expressed as percentages. We also include the tables

with the variable importance values of each feature for each metamodel in the four experiments (Table 3.9, 3.10, 3.11 and 3.12). The charts suggest that feature F1, that of *Number of Attributes*, is the most important one in creating the decision nodes in the classification trees. The second most important is either feature 2 (*Number of Incoming References*) or feature 5 (*Number of Parents*). The last 2 positions are occupied by features 3 or 4 (*Number of Outgoing* or *Number of Children* respectively).

The fact that F1 is the most important feature is an expected outcome. This is because in all the metamodels used, there are types that have attributes assigned to them, thus at some point this becomes a distinctive point between some types. In contrast, there are metamodels which have no containment relationships or references significantly. Thus this specific feature value (i.e., F2 and F3 if there are no references, F4 and F5 if there are no containments) is always 0 between **all** elements. This way features 2 to 4 are sometimes absolutely ignored and thus their average importance value shown in the pie is decreased. For example, as is shown in Table 3.9, in the Chess metamodel (also see Figure A.4(a)), which has no reference relationships, the values of F2 and F3 are 0.

Table 3.9.: Variable importance table for the N-CART experiment.

Model Name	**F1**	**F2**	**F3**	**F4**	**F5**
Ant	16.93 (39.29%)	9.26 (21.49%)	6.88 (15.97%)	3.18 (7.38%)	6.84 (15.87%)
Bibtex	21.99 (46.12%)	0.00 (0.00%)	0.00 (0.00%)	12.36 (25.92%)	13.33 (27.96%)
Bugzilla	10.49 (40.41%)	5.35 (20.61%)	0.00 (0.00%)	4.77 (18.37%)	5.35 (20.61%)
Chess	6.24 (46.46%)	0.00 (0.00%)	0.00 (0.00%)	0.97 (7.22%)	6.22 (46.31%)
Cobol	12.85 (28.71%)	11.73 (26.21%)	7.98 (17.83%)	4.71 (10.52%)	7.49 (16.73%)
Conference	8.14 (25.53%)	8.55 (26.81%)	0.28 (0.88%)	7.58 (23.77%)	7.34 (23.02%)
Profesor	7.64 (41.16%)	1.99 (10.72%)	1.84 (9.91%)	0.66 (3.56%)	6.43 (34.64%)
Usecase	7.32 (17.81%)	10.83 (26.36%)	11.69 (28.45%)	5.40 (13.14%)	5.85 (14.24%)
Wordpress	16.02 (39.43%)	13.74 (33.82%)	8.00 (19.69%)	0.46 (1.13%)	2.41 (5.93%)
Zoo	17.43 (55.49%)	9.76 (31.07%)	4.22 (13.44%)	0.00 (0.00%)	0.00 (0.00%)

Table 3.10.: Variable importance table for the S-CART experiment.

Model Name	**F1**	**F2**	**F3**	**F4**	**F5**
Ant	17.66 (44.48%)	5.49 (13.83%)	4.80 (12.09%)	5.24 (13.20%)	6.51 (16.40%)
Bibtex	25.77 (45.84%)	0.00 (0.00%)	0.00 (0.00%)	14.72 (26.18%)	15.73 (27.98%)
Bugzilla	6.09 (49.15%)	0.00 (0.00%)	0.00 (0.00%)	2.55 (20.58%)	3.75 (30.27%)
Chess	6.85 (47.77%)	0.00 (0.00%)	0.00 (0.00%)	0.66 (4.60%)	6.83 (47.63%)
Cobol	7.96 (30.72%)	6.63 (25.59%)	4.43 (17.10%)	3.01 (11.62%)	3.88 (14.97%)
Conference	5.18 (17.61%)	8.15 (27.70%)	0.89 (3.03%)	7.70 (26.17%)	7.50 (25.49%)
Profesor	6.72 (33.02%)	4.29 (21.08%)	2.48 (12.19%)	1.14 (5.60%)	5.72 (28.11%)
Usecase	7.69 (18.75%)	10.39 (25.33%)	11.03 (26.89%)	6.71 (16.36%)	5.20 (12.68%)
Wordpress	16.48 (41.85%)	11.64 (29.56%)	7.78 (19.76%)	0.78 (1.98%)	2.70 (6.86%)
Zoo	8.09 (53.90%)	4.63 (30.85%)	2.29 (15.26%)	0.00 (0.00%)	0.00 (0.00%)

Table 3.11.: Variable importance table for the N-RF experiment.

Model Name	F1	F2	F3	F4	F5
Ant	7.61 (41.18%)	4.77 (25.81%)	2.13 (11.53%)	1.24 (6.71%)	2.73 (14.77%)
Bibtex	8.56 (47.29%)	0.00 (0.00%)	0.00 (0.00%)	3.55 (19.61%)	5.99 (33.09%)
Bugzilla	3.23 (44.80%)	1.55 (21.50%)	0.00 (0.00%)	0.92 (12.76%)	1.51 (20.94%)
Chess	2.53 (45.34%)	0.00 (0.00%)	0.00 (0.00%)	0.37 (6.63%)	2.68 (48.03%)
Cobol	7.02 (29.16%)	7.38 (30.66%)	3.68 (15.29%)	2.25 (9.35%)	3.74 (15.54%)
Conference	4.00 (28.72%)	3.30 (23.69%)	0.39 (2.80%)	3.09 (22.18%)	3.15 (22.61%)
Profesor	3.24 (42.30%)	0.69 (9.01%)	0.52 (6.79%)	0.18 (2.35%)	3.03 (39.56%)
Usecase	3.61 (19.75%)	5.32 (29.10%)	5.20 (28.45%)	1.36 (7.44%)	2.79 (15.26%)
Wordpress	8.56 (38.20%)	7.12 (31.77%)	4.44 (19.81%)	0.47 (2.10%)	1.82 (8.12%)
Zoo	9.07 (53.26%)	6.08 (35.70%)	1.88 (11.04%)	0.00 (0.00%)	0.00 (0.00%)

Table 3.12.: Variable importance table for the S-RF experiment.

Model Name	F1	F2	F3	F4	F5
Ant	9.36 (52.67%)	1.74 (9.79%)	1.72 (9.68%)	2.34 (13.17%)	2.61 (14.69%)
Bibtex	9.98 (47.12%)	0.00 (0.00%)	0.00 (0.00%)	4.27 (20.16%)	6.93 (32.72%)
Bugzilla	2.32 (50.33%)	0.00 (0.00%)	0.00 (0.00%)	0.67 (14.53%)	1.62 (35.14%)
Chess	2.82 (46.08%)	0.00 (0.00%)	0.00 (0.00%)	0.30 (4.90%)	3.00 (49.02%)
Cobol	4.50 (30.38%)	4.39 (29.64%)	2.32 (15.67%)	1.65 (11.14%)	1.95 (13.17%)
Conference	2.37 (20.97%)	2.83 (25.04%)	0.44 (3.89%)	2.94 (26.02%)	2.72 (24.07%)
Profesor	2.77 (33.74%)	1.43 (17.42%)	0.58 (7.06%)	0.27 (3.29%)	3.16 (38.49%)
Usecase	4.06 (22.80%)	5.18 (29.08%)	4.51 (25.32%)	1.94 (10.89%)	2.12 (11.90%)
Wordpress	9.82 (41.95%)	6.36 (27.17%)	4.44 (18.97%)	0.65 (2.78%)	2.14 (9.14%)
Zoo	3.91 (49.49%)	2.77 (35.06%)	1.22 (15.44%)	0.00 (0.00%)	0.00 (0.00%)

Qualitative Analysis

We now examine the results from a qualitative perspective in order to identify patterns that may occur in the models that affect the prediction accuracy.

By assessing the Bugzilla metamodel (see Figure A.3) we found that all the wrong predictions were done among four classes that were extending the same abstract superclass. More specifically, the types DependsOn, Keywords, Blocks and CC (which extend the class named StringElt) were all identified as being of the same type, the one with the greatest presence in the training data. By looking at the metamodel, we identified that these types follow the structure of modelling inheritance with no concrete differentiating characteristics [162] (i.e., no differentiating point with the parent class as they have no extra attributes, no extra containment relations assigned to them but one extra incoming reference going to each of them). As a result, the feature signature is identical for all of the four and thus the classification algorithm is unable to find a distinctive characteristic to split them into different classes. A similar behaviour was also discovered in the BibTeX metamodel; the types had one differentiating point which was of the same category (i.e., an extra

attribute each). Again, the feature signature was identical.

A way to address this problem could be to introduce other features, atop the five used in this study, which are not calculated based on semantic characteristics, like the features based on *concrete syntax* presented in Section 3.3.2. In addition, including string similarity measurements (like checking the name of the extra added attribute in the BibTeX example) will help, too.

However, that behaviour is not always undesirable: more specifically if the goal is that of metamodel inference, this behaviour will help in identifying possible unnecessary inheritance introduced in the language. Both algorithms used in this approach have built-in mechanisms to group classes/types that are very similar by using the notion of "buckets" in the leaf nodes.

Performance Analysis

Table 3.13 summarises the average execution time for each of the two prediction algorithms (CART and RF) and the two model sets ("Normal" and "Sparse") used in this approach. The specification of the machine used to run the experiments is the following:

- Architecture: x64 (64-bits)
- Processor: Intel(R) Core(TM) i5-4288U CPU @ 2.60GHz
- RAM: 2x8GB DDR3 @ 1600 MHz
- Hard Disk Drive: 256GB PCIe SSD
- Operating System: Mac OS X 10.11.6

Table 3.13.: Average execution time for each metamodel in the classification algorithms approach

		Average Execution Time for Each Metamodel (in seconds)			
		CART		**RF (50 trees)**	
Model Name	**#Types**	**Normal**	**Sparse**	**Normal**	**Sparse**
Chess	2	0.06713	0.06635	0.00720	0.00751
Profesor	4	0.07610	0.09403	0.00751	0.00729
Zoo	5	0.07708	0.08369	0.00800	0.00755
Ant	6	0.10216	0.09426	0.00903	0.00946
Conference	6	0.08496	0.08558	0.00984	0.00721
Usecase	6	0.09611	0.10264	0.00839	0.00755
Bugzilla	7	0.08242	0.07704	0.00734	0.00718
Bibtex	8	0.08319	0.08244	0.00822	0.00843
Cobol	11	0.16032	0.14863	0.00938	0.00703
Wordpress	19	0.15565	0.17360	0.00874	0.00916

As one can see from Table 3.13, the execution times for this approach are very low for all the four experiments. More specifically, times vary from 7.03 milliseconds up to 17.36 milliseconds. There is some raise in execution times as the number of types increases however this is not affecting the applicability of the approach as the execution times are minor. There is also difference between the CART and RF approach as RF performs significantly faster (the measurement is for the 50 trees for which RF reaches its maximum performance). This is expected as in RF a subset of the nodes and the features is taken into account when training is performed.

The results of the average execution times let us believe that this approach has no scalability issues, as the execution times are minor even with large metamodels.

3.5.3. Experiment for Concrete Syntax Features

In this section the experimentation process to evaluate the performance of the proposed approach using features based on graphical characteristics of the example models is presented. An overview is given in Figure 3.11.

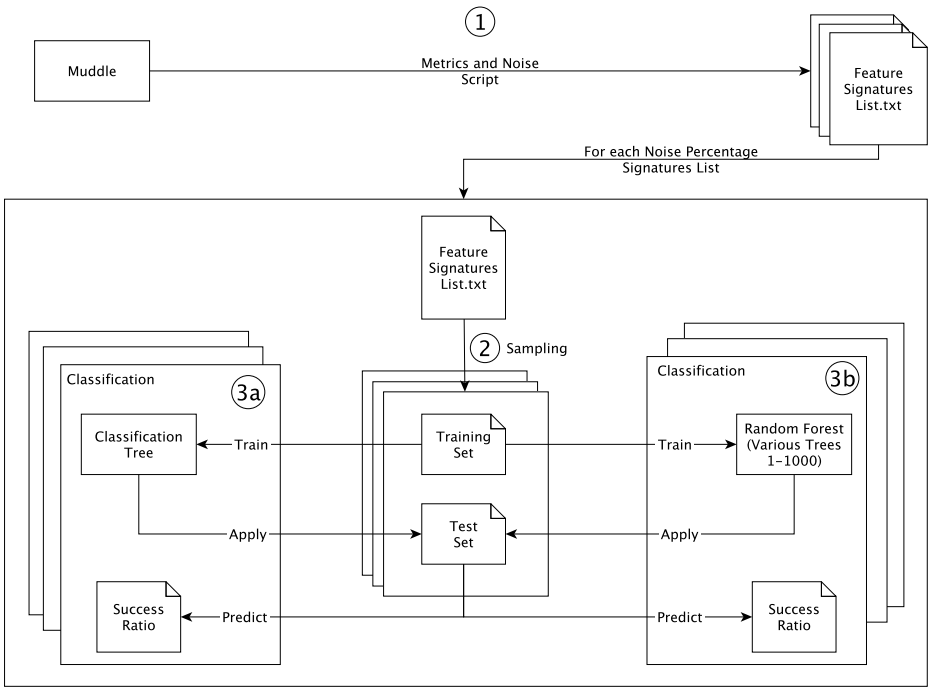


Figure 3.11.: The concrete features experimentation process.

In order to test the proposed approach we applied both classification algorithms to a muddle. This muddle was created before commencing this research as part of a side project to express requirements for a web application [18]. Our experience working with Muddles suggests that it is a fairly complicated example as it consists of more than 100 elements of 20 different types. A comparison with the 10 other models used in the experiment of the previous approach is not possible as these were automatically generated using mechanisms that are not unbiased in the

selection of the four features that we assess in this work: all the nodes were of the same shape, colour and size.

In addition, we tested the reluctance of the proposed approach to human error and the bias that our muddling habits may introduce: we tend to use the same shape when we express a specific type. We need to highlight here, that this does not mean that all elements of the same type have the same features or that each feature value (e.g., rectangle) was used only in one type. Regarding the latter, elements of different types share common features in our experimentation example. For instance, rectangles were used to represent elements of many different types. The same holds with the colours. Arguably, this is the case with any other relatively large muddle as the number of available shapes and colours is in practice limited. However, in order to check if adhering to some basic conventions when drawing an example model is important for the accuracy of the prediction, we performed a second experiment by adding noise to some of the elements by explicitly changing some of their features. We did that gradually by altering randomly *one* feature of none (0%) up to all (100%) of the elements of the muddle using a step of 20% (0%, 20%, ..., 80%, 100%). 40% noise addition means that 40 per cent of the nodes in the diagram have *one* feature (randomly selected) changed to something else (e.g., shape is changed from rectangle to ellipse). A detailed step-by-step description of the experimentation process follows.

Initially a script (see Algorithm 2) is run to collect the features from the muddle and place them into a list that includes the feature signatures for each element (step (1)). As the example has 105 nodes, there are 105 feature signatures in this signatures list. An extract of the signatures list is shown in Listing 3.4.

```
roundrectangle,#F5F5F5,934.8791503906252,995.6377243193067,TR
roundrectangle,#F5F5F5,904.0000000000002,943.9717086943067,HomePage
roundrectangle,#F5F5F5,862.0000000000002,110.66354037747524,Menu
rectangle,#FFCC00,130.17326732673268,30.0,MenuItem
...
roundrectangle,#F5F5F5,862.0,377.666015625,Frame
rectangle,#FFCC00,98.0,78.0,Image
roundrectangle,#F5F5F5,854.9999999999995,182.91890279771678,Frame
rectangle,#FFCC00,228.7499999999999,30.0,Heading
...
hexagon,#FFCC00,372.6980198019801,173.34003712871288,Requirement
roundrectangle,#F5F5F5,1024.897485014718,1032.8898611540842,Step
roundrectangle,#F5F5F5,862.0000000000002,110.66354037747521,Menu
...
rectangle,#FFCC00,136.81786872407747,43.180693069306926,Button
rectangle,null,994.8974850147181,981.2238455290842,Page
```

Listing 3.4: An example of a features signature list.

The same process is repeated 6 times; in each iteration a new signatures list is created for each noise level. At the end of step (1) there are 6 signature list files generated. In step (2), each of these lists is randomly separated into a *training*

and a *test set*. The training set contains the nodes which, in a realistic scenario, are typed by the engineer while the test set contains those left untyped. Following this random sampling we simulate that scenario. In order to reach unbiased results we perform the sampling process 10 times for each file (10-Fold). It is also of interest to identify if the amount of knowledge that the algorithm has on each diagram is of importance to the success ratio. For that reason we use 7 different sampling rates; from 30% to 90%. For example, a 40% sampling rate means that in the simulation, 40% of the nodes are thought to be of known type (and thus the training set consists of them) while the rest (60%) are the nodes for which the type is unknown.

The generated pairs of training and test sets are then fed one after the other to the CART (step (3a)) or the Random Forests (step (3b)) algorithm. The algorithm is trained on the training set and predicts the types of elements of the test set. The prediction accuracy is then calculated by dividing the total number of correct predictions to the number of all untyped elements. Type predictions are checked against the correct types of these nodes to calculate the success ratio. The correct types are known to us, as these were originally defined to the example muddle and kept before splitting it into sets of nodes with known and unknown types. The number of trees that the Random Forest algorithm will be trained with is of importance so we run the same experiment for 7 different values of trees: 1, 5, 10, 50, 250, 500 and 1000. For this experiment, we used the same R [157] packages used in the previous experiment described in Section 3.5.1.

After having noise injected for 6 levels (0%-100%), for 7 different sampling rates (30%-90%), 10 different times each to avoid a lucky (or unlucky) sample we end up having 420 pairs of training and test sets. For the Random Forest each of these 420 pairs are tested for the 7 different values of the tree variables ending up having 2,940 runs in total.

3.5.4. Results and Discussion

CART

Table 3.14 summarises the results for the 420 runs in the CART experiment. Each cell contains the average accuracy of the classification for the 10 runs for the specific added-noise level and sampling rate. For instance, the highlighted value **0.59** indicates that on average, 59% of the missing types were successfully predicted for the 10 samples of the 40% added-noise model, using a 70% sampling rate.

The average accuracy for the CART prediction is 64.6% in the case where there is no added noise. As expected, when the sampling rate is increased, which means that more nodes are fed to the algorithm for training, the accuracy is improved. This is verified by the line charts given in Figure 3.12. The lines follow an upward inclination for all the noise levels. The opposite trend is visible when the level of noise is increased: higher noise results in worse accuracy. The average accuracy

Table 3.14.: Results summary table for CART (concrete syntax features)

	Average Accuracy for Different Sampling Rates (N-CART)								
Noise Level	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
0%	0.52	0.59	0.68	0.65	0.68	0.70	0.70	0.646	0.89
20%	0.50	0.55	0.63	0.63	0.68	0.70	0.66	0.622	0.89
40%	0.48	0.54	0.59	0.56	**0.59**	0.60	0.63	0.571	0.89
60%	0.40	0.47	0.48	0.48	0.49	0.50	0.60	0.488	0.96
80%	0.33	0.35	0.46	0.45	0.53	0.57	0.49	0.454	0.86
100%	0.32	0.35	0.37	0.41	0.40	0.41	0.45	0.388	0.96
Cor. 2	-1.00	-0.94	-1.00	-1.00	-0.93	-0.89	-1.00		

drops to 38.8% in the case where all the elements have a randomly selected feature altered. The following correlation coefficients confirm both these visual observations.

Cor. 1: How strong is the dependency between the sampling rate and the success score?

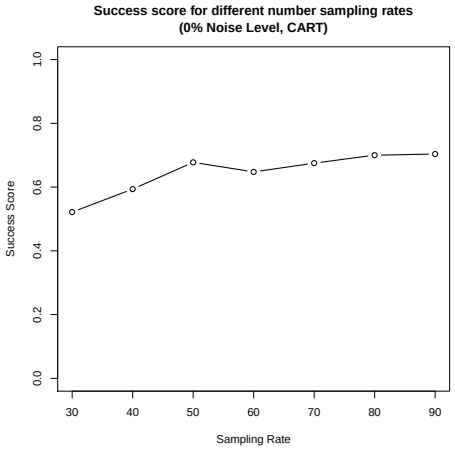
Cor. 2: How strong is the dependency between the added-noise level and the success score?

As expected, the correlation coefficient values for "Cor. 1" indicate strong dependency for all the added-noise levels. This means that prediction scores increase as training sets become larger or in other words the chances of correctly predicting the type are significantly increased if the number of untyped nodes is decreased. The same behaviour was also observed in the type inference approach presented in 3.5.1 where the features are based on the semantics of the example models.

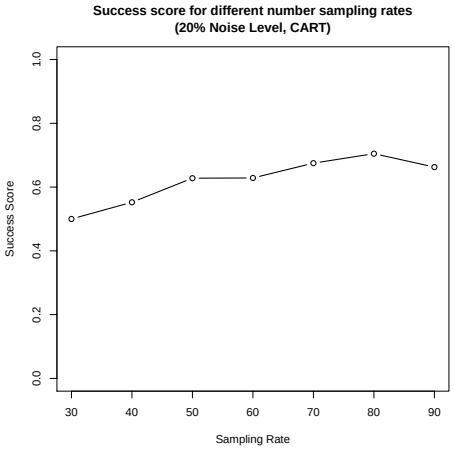
Regarding the second correlation ("Cor. 2") we observe a perfect (negative) correlation between the number of nodes in a drawing that have altered features and the success score across all the sampling rates. This is evidence that following specific rules in the concrete syntax of the drawing, increases the chances for correct type inference. By the term "specific rules" it is not implied that these rules should be strict. As discussed in Section 3.5.3, in the 0% added-noise example the authors use the same shapes to express the same concepts or in other cases the same color but not in a rigorous manner: same graphical properties are used in different concepts while the same concepts may have different graphical properties.

These two observations are summarised in the following:

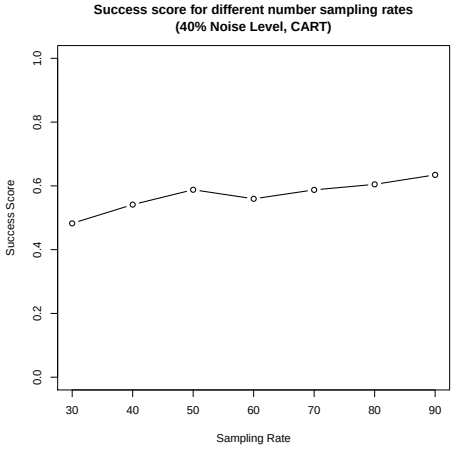
1. Fewer untyped nodes results to higher accuracy of the approach.
2. Following drawing conventions results in a higher accuracy.



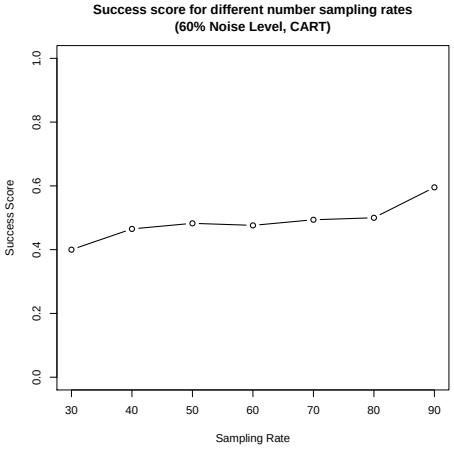
(a) 0% Noise Level



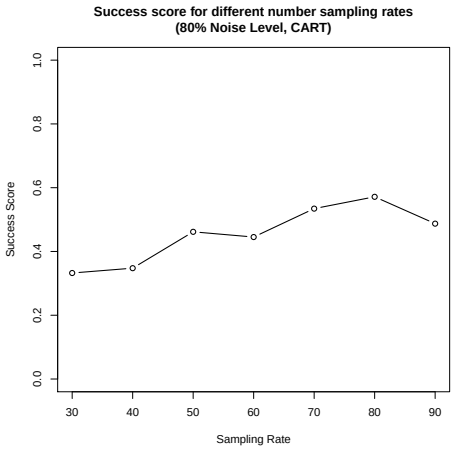
(b) 20% Noise Level



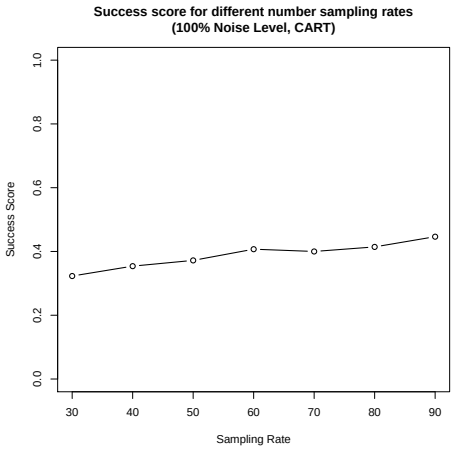
(c) 40% Noise Level



(d) 60% Noise Level



(e) 80% Noise Level



(f) 100% Noise Level

Figure 3.12.: Accuracy for different sampling rates (CART - Concrete).

Random Forests

The same set of four features was used as an input to the Random Forests algorithm to identify if this algorithm performs better than CART. Looking at the raw results summarised in Table 3.15 the same trends identified for the CART algorithm appear in RF as well. More specifically, as the sampling rate is increasing (thus, fewer nodes are left untyped) the accuracy is improved. The same inverse trend is noticed regarding the added noise: as the noise level is increased the prediction accuracy drops. These two observations are verified by the correlation coefficient values, named **"Cor. 1"** and **"Cor. 2"** in Table 3.15. Their description is repeated here:

Cor. 1: How strong is the dependency between the sampling rate and the success score?

Cor. 2: How strong is the dependency between the added-noise level and the success score?

Table 3.15.: Results summary table for RF (concrete syntax features)

		Average Accuracy for Different Sampling Rates (RF Concrete)								
Noise Level	**#Trees**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
0%	1	0.47	0.48	0.54	0.58	0.62	0.64	0.64	0.567	1.00
	5	0.52	0.59	0.66	0.64	0.68	0.70	0.71	0.643	0.96
	10	0.56	0.61	0.66	0.67	0.70	0.71	0.73	0.663	1.00
	50	0.59	0.63	0.70	0.67	0.71	0.74	0.75	0.684	0.96
	250	0.58	0.63	0.72	0.68	0.71	0.76	0.75	0.691	0.86
	500	0.59	0.63	0.71	0.68	0.71	0.76	0.73	0.687	0.86
	1000	0.59	0.63	0.71	0.68	0.71	0.74	0.75	0.688	0.89
20%	1	0.42	0.46	0.52	0.55	0.57	0.60	0.57	0.528	0.89
	5	0.50	0.54	0.58	0.61	0.64	0.63	0.64	0.593	0.96
	10	0.52	0.57	0.62	0.63	0.68	0.65	0.69	0.623	0.96
	50	0.54	0.60	0.66	0.65	0.67	0.70	0.74	0.652	0.96
	250	0.55	0.60	0.66	0.65	0.68	0.70	0.74	0.655	0.96
	500	0.55	0.61	0.66	0.65	0.69	0.71	0.73	0.657	0.96
	1000	0.55	0.60	0.66	0.65	0.69	0.71	0.73	0.656	0.96
40%	1	0.41	0.41	0.50	0.52	0.52	0.52	0.52	0.487	0.89
	5	0.46	0.52	0.55	0.56	0.58	0.57	0.55	0.543	0.75
	10	0.49	0.53	0.57	0.57	0.60	0.62	0.58	0.567	0.86
	50	0.53	0.57	0.60	0.60	0.62	0.61	0.62	0.593	0.86
	250	0.53	0.57	0.61	0.61	0.63	0.63	0.61	0.599	0.82
	500	0.53	0.57	0.61	0.60	0.63	0.62	0.62	0.599	0.86
	1000	0.53	0.56	0.61	0.60	0.63	0.63	0.61	0.598	0.82

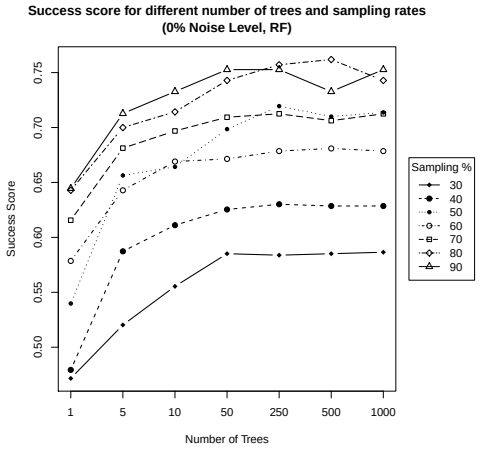
Table 3.15.: Results summary table for RF (concrete syntax features)

		Average Accuracy for Different Sampling Rates (RF Concrete)								
Noise Level	**#Trees**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
60%	1	0.34	0.36	0.39	0.37	0.42	0.43	0.51	0.402	0.96
	5	0.42	0.44	0.47	0.48	0.47	0.49	0.56	0.476	0.89
	10	0.44	0.46	0.48	0.48	0.51	0.52	0.58	0.498	0.96
	50	0.47	0.49	0.50	0.50	0.52	0.52	0.61	0.516	0.93
	250	0.46	0.49	0.53	0.50	0.51	0.55	0.61	0.521	0.89
	500	0.46	0.49	0.53	0.49	0.51	0.55	0.61	0.519	0.89
	1000	0.47	0.50	0.53	0.49	0.52	0.54	0.61	0.522	0.82
80%	1	0.33	0.28	0.38	0.37	0.40	0.43	0.34	0.363	0.57
	5	0.36	0.38	0.45	0.44	0.51	0.48	0.45	0.437	0.68
	10	0.39	0.41	0.45	0.46	0.51	0.51	0.51	0.462	1.00
	50	0.40	0.45	0.51	0.50	0.55	0.59	0.53	0.504	0.86
	250	0.41	0.46	0.51	0.51	0.58	0.57	0.57	0.516	0.89
	500	0.41	0.46	0.51	0.50	0.59	0.56	0.57	0.515	0.86
	1000	0.41	0.46	0.52	0.51	0.58	0.57	0.57	0.517	0.86
100%	1	0.28	0.30	0.32	0.31	0.38	0.40	0.41	0.342	0.96
	5	0.34	0.37	0.37	0.39	0.40	0.48	0.44	0.397	0.96
	10	0.35	0.38	0.40	0.41	0.44	0.50	0.46	0.420	0.96
	50	0.37	0.42	0.45	0.44	0.46	0.50	0.50	0.447	0.96
	250	0.38	0.43	0.45	0.44	0.44	0.49	0.53	0.451	0.86
	500	0.38	0.43	0.46	0.46	0.44	0.50	0.53	0.455	0.86
	1000	0.38	0.43	0.45	0.44	0.45	0.50	0.54	0.455	0.89
	Cor. 2[3]	-0.99	-1.00	-1.00	-0.94	-0.94	-0.94	-1.00		

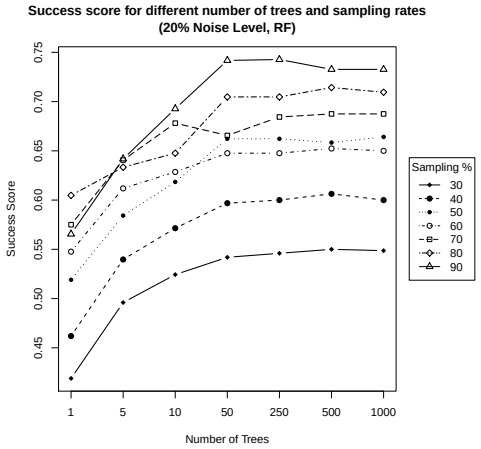
Regarding "Cor. 1", the values reveal a very strong or strong correlation for all the number of trees and noise levels. Regarding "Cor. 2", the values based on the 250 trees, show a very strong correlation between the added noise level and the accuracy for all the sampling rates.

The results shown in Table 3.15 are also given as line charts in Figure 3.13. From the six graphs for the different added noise levels one can see that the accuracy is increased steadily and hits a plateau at either 50 trees value or at 250 trees value (this depends on a mixture of the added noise level and the sampling rate). After these values there is no improvement in the accuracy while the time needed for the prediction continues to increase. Thus, as a conclusion, setting **250 trees** as a parameter for this algorithm **is a value that maximises the accuracy to time ratio** when the four features based on the concrete syntax are used.

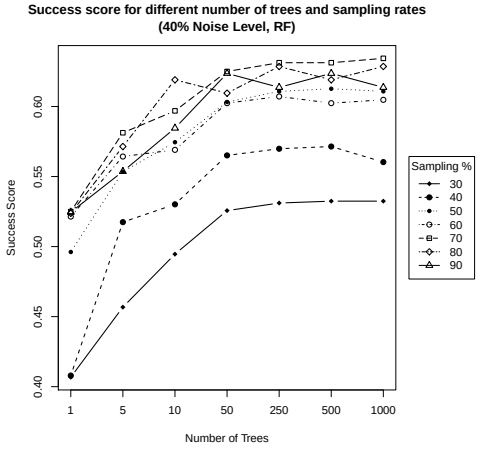
[3]Calculated based on the 250 trees values



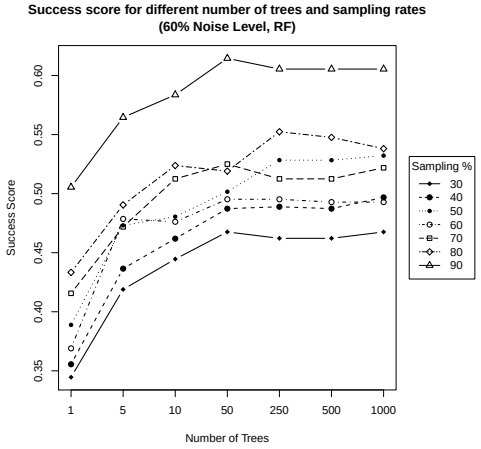
(a) 0% Noise Level



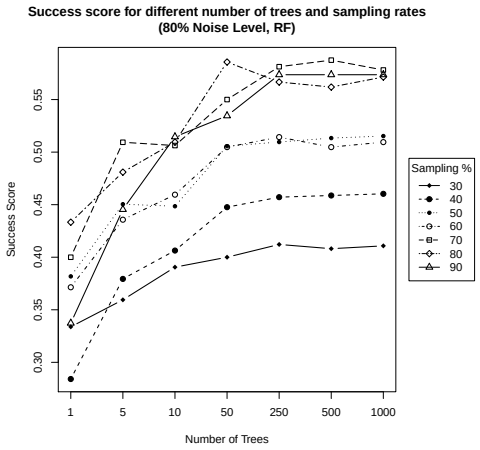
(b) 20% Noise Level



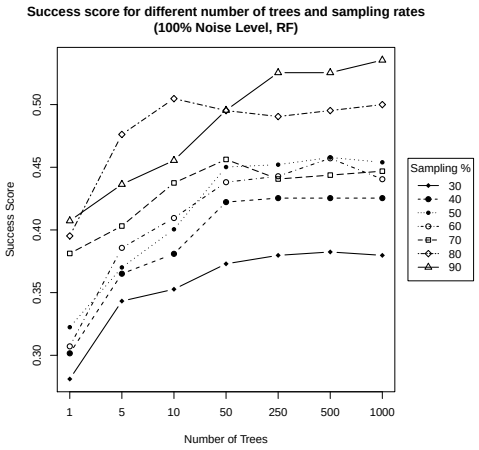
(c) 40% Noise Level



(d) 60% Noise Level



(e) 80% Noise Level



(f) 100% Noise Level

Figure 3.13.: Accuracy for different sampling rates and number of trees (RF - Concrete).

The results for both algorithms answer the research question **(RQ2)** set at the beginning of the experimental evaluation. For both approaches, the prediction accuracy is high and acceptable for all the scenarios where the added-noise in less than 40%. When the added-noise percentage is increased, then approach has satisfactory accuracy for higher sampling rates (above 70-80%).

Comparison: CART vs RF

Looking at the averages for all the sampling percentages for each noise level one can see that the Random Forests algorithm has *better* accuracy than CART in all the scenarios. Table 3.16 summarises this. Based on the literature [130] this is an expected outcome as normally Random Forests outperform CART algorithms, however it is different from the outcome of the experiment with the features based on semantics. There are at least two conditions that are different between these two experiments and could possibly explain this behaviour. Firstly, the different nature of the features used (concrete syntax vs. semantics) and secondly the fact that in this experiment the approach is applied to a single example model. Suggestions for future work on investigating this behaviour are provided in Section 6.2.

Table 3.16.: Accuracy difference trends between CART and RF experiments.

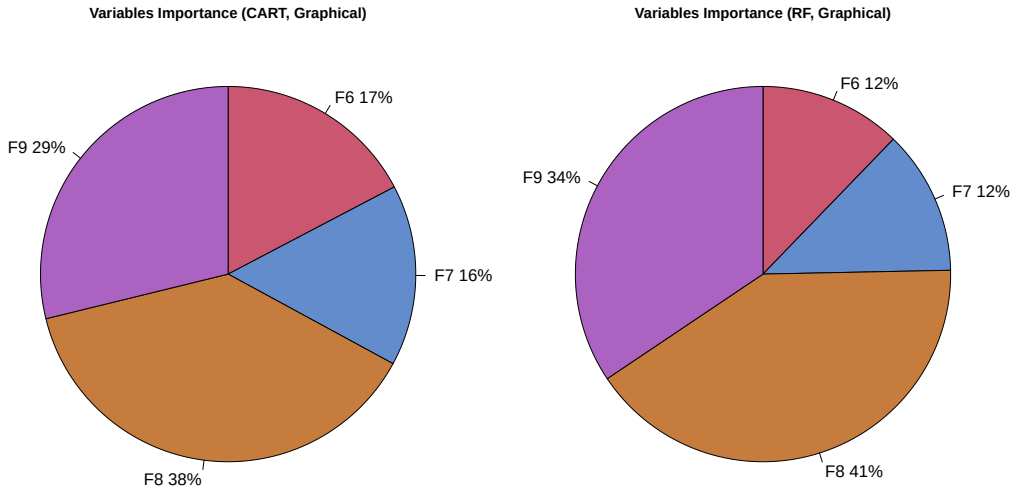
Noise Level	Difference from CART to RF (250 trees)
0%	↗
20%	↗
40%	↗
60%	↗
80%	↗
100%	↗

Variables Importance

The level of significance of the features in the decision making for each algorithm is of interest as it will help identify which characteristics of the diagram are more important for the algorithm to split the nodes into the different classes (types in this domain).

The significance of each of the four features which are based on the concrete syntax is shown in the pie charts of Figure 3.14. More specifically, Figure 3.14(a) shows that in the scenario where the CART algorithm was used, the most important variable was F8, which is that of the width of the node. The second most important is related to the size as well and it is that of height of the node. The last two features have only half the importance of the top two features combined as they have a score of 33% in total (17% for F6 which is the shape of the node and 16% for F7 which is the colour of the node).

Regarding Random Forests, the results follow the same pattern but this time the first top two features, which are again F8 and F9, accumulate three quarters of the total importance. More specifically, the width of the node (F8) scores a 41% in importance and the height (F9) a 34%. The other two variables (colour and shape) tie in the last place and share the remaining 24% (12%) each.



(a) CART (b) Random Forests

Figure 3.14.: Variables importance of concrete features.

Tables 3.17 and 3.18 show raw values of the importance of each variable for each added noise level in the experiment for the CART and RF algorithms respectively.

Table 3.17.: Variable importance table (CART - Concrete).

Model Name	F6	F7	F8	F9
0%	13.96 (16.51%)	12.26 (14.50%)	34.37 (40.66%)	23.94 (28.32%)
20%	14.41 (17.81%)	10.30 (12.73%)	31.73 (39.21%)	24.49 (30.26%)
40%	13.24 (16.21%)	12.82 (15.69%)	31.85 (38.99%)	23.78 (29.11%)
60%	13.46 (18.79%)	10.94 (15.28%)	25.92 (36.19%)	21.30 (29.74%)
80%	11.67 (16.67%)	12.73 (18.18%)	27.68 (39.53%)	17.94 (25.62%)
100%	11.59 (18.22%)	11.55 (18.16%)	21.53 (33.85%)	18.94 (29.78%)

Table 3.18.: Variable importance table (RF - Concrete).

Model Name	F6	F7	F8	F9
0%	4.88 (10.34%)	5.35 (11.34%)	19.74 (41.83%)	17.22 (36.49%)
20%	5.81 (11.88%)	5.25 (10.73%)	21.13 (43.20%)	16.72 (34.19%)
40%	5.71 (11.16%)	5.69 (11.12%)	22.69 (44.34%)	17.08 (33.38%)
60%	6.84 (13.49%)	6.32 (12.46%)	18.81 (37.09%)	18.74 (36.96%)
80%	6.46 (12.41%)	7.67 (14.73%)	22.08 (42.40%)	15.86 (30.46%)
100%	7.19 (13.93%)	7.27 (14.08%)	18.98 (36.76%)	18.19 (35.23%)

3.6. Limitations

The data used to evaluate the performance of the first proposed approach were generated using a random model generator. An issue of that is that we are using models which conform to a metamodel, and not real muddles. This was done for pragmatic reasons: we have a model generator that uses genetic algorithms to produce random models (no random muddle generator currently exists) and we wanted to evaluate the feasibility of the proposed approach for type inference on a large number of example models.

Regarding the approach that is based on the features extracted from the semantics of the models, we do not believe that the use of models instead of muddles has significant impact on the experimental results. The accuracy of our classification algorithm depends only on the features described in Section 3.3.1; randomly generated models and muddles will not be significantly different in terms of these features. To support this argument, we ran the prediction on a real muddle and the results suggest that the performance of the predictions is not affected by this parameter. In addition, we injected noise to the models that affect 4 out of 5 features by generating a second "Sparse" set. This added noise sometimes affected the prediction accuracy sometimes positively, sometimes negatively and sometimes had no effect. However, other user-defined models and muddles *may* differ - and as such future work on experiments with more user-created muddles is needed.

A second issue related to the use of this generator is that although it generates random models, the number of attributes that each node has is always the same for nodes of the same type. However, this does not always work in favour of our approach, because in cases where two different types have the same number of attributes, all instances will have the same value in the attributes feature in their signature. A work-around for this would be the injection of noise in the number of attributes that each node has by running a post-generation script that randomly deletes attributes from elements. Interesting research directions for future work are described in Section 6.2.

An additional limitation is the way in which the feature signatures are currently calculated. As shown in Algorithm 1, the uniqueness of a reference is decided based on its name, thus this current experiment relies on the assumption that the references will have a name assigned to them.

For both the algorithms used, the outcome that fewer types lead to better results was observed. However, we need to mention that this might be a result of having higher nodes:types ratio in metamodels with fewer types than in those with more types, especially in such approaches where learning is used. Further experimentation would be required to additionally check the impact of having more or fewer nodes per type.

In this experiment, 10 metamodels were used in total from which a number of

muddles were generated. The metamodels were picked randomly from a repository of 500 metamodels with no specific criteria other than that of having a variation in size (number of concrete meta-classes). The number of types in these varied from 2 up to 19 as shown in Table 3.2. It would be of interest to experiment with even larger metamodels, although our experience with working on muddles suggests that having a flexible model with more than 20 different types is a marginally realistic scenario.

Finally, the number of instances that the experiment was ran on is sufficient as it complies with the standard 10-fold methodology used in the domain of classification algorithms.

Regarding the approach using features based on the concrete syntax it was evaluated using one real muddle created as part of a side project. The fact that only one muddle was examined presents a threat to validity, thus we experimented with the intentional addition of noise in the diagram to check how this affects the prediction accuracy. A strong correlation between the percentage of altered nodes and the accuracy was identified providing evidence that this approach is more successful if it is used under the assumption that modellers tend to use, to the extent possible, the same graphical notation for elements of the same concept. We believe that this behaviour can be “unintentionally” replicated because of the “copy-paste" nature of muddling (e.g., create an animal node once and then copy & paste the node when you need it again). This way the same graphic notation is used for all the elements of the same type reducing the effect of this threat.

3.7. Chapter Summary

In this chapter we proposed the use of classification algorithms for the type inference in the initial steps of flexible MDE, providing support for moving from partially typed example models to more complete ones. More specifically, we proposed two different sets of features that can be used in classifying the elements, based on the semantics and concrete syntax of example models. Both feature sets were tested with two classification algorithms, Classification and Regression Trees and Random Forests.

In order to test the proposed approach we ran experiments on a large number of example models. The results suggest that even in large models with more than 10 concrete types, the prediction accuracy is significant. It varies based on the number of nodes that are left untyped, the number of total candidate types and the classification algorithm used. Regarding the algorithm, the results show that CART has already maximized the prediction performance for the feature signatures which are based on the semantics and the use of an algorithm that belongs to the same category does not improve the results. In contrast, using the concrete features signatures we have seen an improvement in the accuracy.

For the Random Forests algorithm we used 7 different values for the number of trees that the algorithm is trained with, identifying a point (i.e., 50 trees for features based on semantics and 250 trees for concrete features) after which the prediction accuracy reaches a plateau. Finally, we calculated the importance of each variable in both algorithms for all the 9 features proposed as part of this work.

CHAPTER 4

Type Inference using Constraint Programming

4.1. Introduction

In flexible MDE approaches, the language engineering process starts with the definition of example models [5,23,29]. These example models help language engineers better understand the concepts of the envisioned DSL and can be used to infer *draft metamodels* manually or (semi-)automatically. This can eventually lead to the definition of the final metamodel. In this fashion, a richer understanding of the domain can be developed *incrementally*, while concrete insights (e.g., type information) pertaining to the envisioned metamodel are discovered.

In Chapter 3, an approach to tackle type omissions from nodes of example models was proposed. The algorithm is trained on the elements of the example models without requiring the existence of a draft metamodel. In this chapter a novel approach to addressing the challenges associated with type omissions, but this time taking into account the *draft metamodel* is proposed based on Constraint Programming (CP) principles. Thus, this approach can be applied in scenarios where a draft metamodel is already inferred. As described in Section 1.1.1, the inference of draft metamodels based on the knowledge of the domain acquired till that point from the example models is possible. This iterative and incremental process, depicted in Figure 1.1, helps in the development of a richer understanding of the domain. In the approach presented here, the syntax and the semantics defined in the draft metamodel are transformed into constraints that are then applied to the example models to reduce the number of possible types of untyped nodes.

The requirement of a draft metamodel is a restriction of our approach comparing with the type inference approaches used in metaBUP [5] and Flexisketch [6]. How-

ever, our approach does not rely on the concrete syntax of the example models as the aforementioned approaches do. In that way, language engineers and domain experts are not required to have the concrete syntax in mind when expressing the concepts of the domain. In addition, our approach *guarantees* that the correct type will be in the set of possible types returned to the user; the Flexisketch [6] approach returns the most similar types, but not necessarily the correct ones.

The approach proposed here is also close to the one presented in [108] that uses CSP principles in partial models. Compared to [108] our approach is different in two important points. The first is that [108] requires the type of each element to be defined, otherwise the element is ignored. Secondly, their approach produces only one possible solution that fulfils the rules of the CSP. Our approach calculates and returns *all* the possible solutions to the CSP and the language engineer picks the correct one that represents the envisioned DSL. By calculating all the possible solutions, we actually identify all the possible values (types) from the domain (all the available types) each element can take without violating the rules imposed by the metamodel. The same differences exist with the approach proposed in [145] which uses CSP principles to validate EMF models. Their approach requires the types of the elements to be defined and returns only one solution to verify that the model is valid.

The rest of the chapter is structured as follows. Section 4.2 includes an overview of the proposed approach. In Section 4.3 the CP algorithm used for type inference is described in detail followed by the algorithm used to transform the draft metamodel and the example models into a CSP. In Section 4.4, an empirical evaluation of the performance of the proposed approach is conducted. The results of running the experiments are discussed in Section 4.4.2 from both a quantitative and qualitative perspective. Any threats to experimental validity and limitations of the proposed approach are discussed in Section 4.5.

Note: The technical work described in Section 4.3.1 was carried out in collaboration with Dr. Robert Clarisó from the IT, Multimedia and Telecommunication Department, Universitat Oberta de Catalunya, Barcelona, Spain. Parts of Section 4.3.1 of this chapter were written by Dr. Robert Clarisó for [19] . My contributions in the work described in Section 4.3.1 are the definition of the problem representation, the variables, the domain and the representation of the results. In addition, a working version of the constraints and the solver was implemented in SWI-PROLOG [144] and handed to Dr. Robert Clarisó who produced a refined version that avoids redundant computations using ECL^{i}PS^e [14], and is the one used in the evaluation of this work.

4.2. Type Inference

In this section the proposed approach for type inference in flexible MDE using CP principles is presented. An overview is given in Figure 4.1.

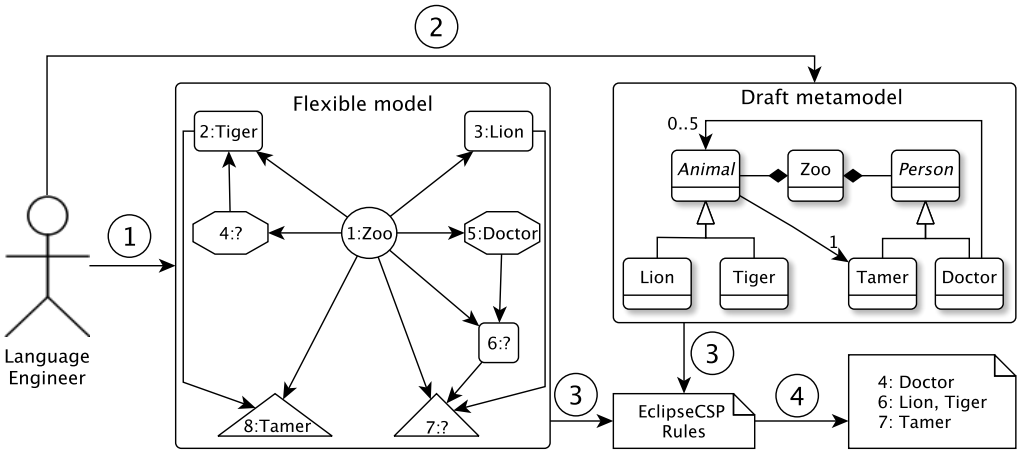


Figure 4.1.: An overview of the proposed approach for type inference for example models defined as part of flexible MDE approaches using constraint programming principles.

The approach starts with the language engineers having example models drawn using a flexible modelling approach (step (1)). This may involve iterative changes and updates to the example models, after which enough knowledge is acquired for the production of a first draft version of the DSL's metamodel (step (2)). This can be done either manually, or automatically using one of the approaches presented in Section 2.4. As described in Section 1.1.1, the example models may have nodes that are left untyped. At this stage, the draft metamodel might be a partial one, which only describes the concepts that are defined in the example models. Thus this approach works under the *Closed World Assumption (CWA)* [163]. The language engineer may want to continue working on the example models by introducing new or evolved concepts. This is an *iterative* process: new concepts can be introduced in the example models, and the metamodel could be updated until a final version is ready. During each iteration, when a stable but incomplete metamodel is defined, the proposed approach can be used to automatically assess the example models and the metamodel to provide suggestions for the nodes that were left untyped to facilitate the engineers having complete models so they can easier proceed to the next iteration. This is done in step (3) (Figure 4.1). More specifically, a custom-made script analyses the example models and the draft metamodel and produces a set of constraints using model-to-text transformations. This auto-generated file can be consumed by a constraint solver (e.g., ECL^{i}PS^e [110]) which suggests the possible types for each node (step (4)). More details about the CSP algorithm and the scripts responsible for the automatic transformation of the example models and metamodels to a CSP are given in Sections 4.3.1 and 4.3.2 respectively. Once the

type suggestions have been generated, language engineers can pick the correct type from those suggested (if there is more than one) and assign it to the node.

The approach relies on the assumption that language engineers have acquired enough knowledge from the example models to come up with a draft metamodel that describes the envisioned DSL. Thus, it is important to highlight that in contrast with our approach presented in Chapter 3, this work requires the existence of a draft version of a metamodel. Beyond the requirement for a metamodel, another important difference from the approach presented in Chapter 3 is the fact that in this approach, the correct type for each node is **always** included in the set of suggested types. In the previous work, the suggested type is not guaranteed to be the correct one. A trade-off for that is the fact that there might be more than one possible type suggested for each node while in the previous there was always one type returned, not always the correct one though. In addition, although the approach guarantees that the correct type will be included in the set of possible types returned, this does not mean that all the types returned for a node can be combined validly with all the types returned for another node as constraints may induce conflicting typing configurations.

Currently the errors of interest (i.e., type omissions) are eliminated manually by engineers by selecting an appropriate type from a set of possible types. That means, that if in the draft metamodel there are N concrete types, the language engineer has N options for each untyped element. Following this practice one does not benefit from information that exists in the draft metamodel and that could possibly help in reducing the number of possible types for a specific node. For example, if in the metamodel it is defined that nodes of "Type 1" can only be connected with nodes of "Type 2", then if an untyped node is connected with a node of "Type 2", it can be inferred that the type of the missing node is "Type 1". An advantage of that second approach is that the search space for the possible types suggested to the language engineer can be reduced from N possible types to M, where $M \in [1, N]$.

4.3. The Constraint Satisfaction Problem

4.3.1. CSP Formalisation

In this section we describe how the type assignment problem is formalized as a CSP. A CSP is characterized by three elements:

1. The set of *variables* involved in the problem.
2. The *domain* of each variable (i.e., the set of potential values it can take).
3. The *constraints* over the variables that define valid value assignments.

A *solution* to a CSP is an assignment of values to variables such that (a) each variable is given a value within its domain and (b) all constraints are satisfied by the

assigned values [164]. Depending on the CSP, there may be no solution (unsatisfiable problem [165]), a single solution or more than one.

Algorithm 4 Computing feasible types.

```
1: {Step 1: Construct the CSP}
2: N ← set of untyped nodes in Model
3: T ← set of non-abstract types in MetaModel
4: Vars ← N                                              {Define variables}
5: for all v ∈ Vars do
6:    Domain(v) ← T                                      {Define domains}
7: end for
8: Constraints ← ∅                                       {Define constraints}
9: for all edge ∈ Model do
10:   Constraints ← Constraints ∪ compatibleAssociation(edge, MetaModel)
11: end for
12: for all node ∈ Model do
13:   for all association ∈ MetaModel do
14:      Constraints = Constraints ∪ multiplicityBounds(node, association)
15:   end for
16: end for
17:
18: {Step 2: Find feasible types by iteratively solving the CSP}
19: for all v ∈ Vars do
20:   for all d ∈ T do
21:      Feasible[v,d] ← false
22:   end for
23: end for
24: for all v ∈ Vars do
25:   for all d ∈ Domain(v) such that Feasible[v,d] = false do
26:      solution ← solveCSP(Vars, Domain, Constraints ∪ (v = d))
27:      if solution exists then
28:         for all v′ ∈ Vars do
29:            Feasible[v′, value(v′, solution)] ← true
30:         end for
31:      end if
32:   end for
33: end for
34: return Feasible
```

Considering these, type assignment in flexible modelling approaches can be formalized as the following CSP, as described in Step 1 of Algorithm 4:

1. *Variables*: There is one variable per untyped node in the model, representing the type of that node (line 4).

2. *Domain*: Untyped objects may be assigned any non-abstract type in the metamodel. Thus, the domain of each variable is all the concrete types (line 6).

3. *Constraints*: Edges among nodes define some restriction on the valid type assignments (lines 8-16):

   a) All edges must belong to an association defined in the metamodel (line 10), i.e., the types of source and target nodes must be compatible with some association.

   Formalization: Let $\langle obj_1, obj_2 \rangle$ be an edge between two objects obj_1 and obj_2 in the model M. Any object obj is an instance of class $type(obj)$ in the metamodel MM. Pairs of classes may be related through associations, e.g., $\langle t_A, t_B \rangle$, or inheritance hierarchies. Let $super(t)$ denote the set of direct superclasses of a class t and let $ancestors(t)$ denote the set defined inductively as follows: $t \cup ancestors(super(t))$.
   Given an edge e and an association as, the edge is type-compatible with the association if the following holds:

   $$\begin{aligned} &compatible\big(e = \langle obj_1, obj_2 \rangle, as = \langle t_A, t_B \rangle\big) := \\ &\qquad (t_1 = type(obj_1)) \ \wedge \ (t_2 = type(obj_2)) \wedge \\ &\qquad (t_A \in ancestors(t_1)) \ \wedge \ (t_B \in ancestors(t_2)) \end{aligned}$$

   Then, the constraint can be expressed as follows:

   $$\forall edge \in M : \exists assoc \in MM : compatible(edge, assoc)$$

   b) Edges must respect the multiplicity constraints of associations defined in the metamodel (line 14), i.e., the number of edges corresponding to a given association must be between the lower and upper bound.

   Formalization: Let l^t_{as} (respectively u^t_{as}) denote the lower (upper) bound on the multiplicity of role t in association as.
   Let $from(obj, as)$ (respectively, to) denote the number of edges in the model M that have object obj as a source (resp. target) and are compatible with association as:

   $$\begin{aligned} from(obj, as) &:= \big(\#e = \langle obj, obj' \rangle \in M : compatible(e, as)\big) \\ to(obj, as) &:= \big(\#e = \langle obj', obj \rangle \in M : compatible(e, as)\big) \end{aligned}$$

   Then, the constraint can be expressed as follows:

   $$\begin{aligned} &\forall obj \in M : \forall as = \langle t_A, t_B \rangle \in MM : \\ &\qquad (l^{t_A}_{as} \leq from(obj, as) \leq u^{t_A}_{as}) \ \wedge \ (l^{t_B}_{as} \leq to(obj, as) \leq u^{t_B}_{as}) \end{aligned}$$

A solution to this CSP is a type assignment that conforms to the metamodel. We are not interested in a single type assignment but rather the set of potential types for each object for which there is a valid type assignment. Therefore, we will need to solve this CSP several times, once per each pair ⟨variable, type⟩. The existence of a solution to this CSP means that the type can be assigned to that variable without violating any constraints. Step 2 of Algorithm 4 describes this procedure.

To avoid redundant computations, if a pair ⟨variable, type⟩ has already appeared in the solution to any of the previous CSPs (line 29), then it can be skipped (line 25) as we already know that this type can be assigned to that variable. Thus, considering a model with n untyped objects and a metamodel with m non-abstract types, the number of CSPs that need to be solved in the worst-case can be calculated as follows. The total number of ⟨variable, type⟩ pairs is $n \cdot m$. The solution to the first CSP will yield one potential type assignment per variable, i.e., n pairs. Hence, Algorithm 4 will require solving at most $(n \cdot m) - n + 1$ CSPs. The search space of each CSP has n^m potential solutions, even though in practice the majority of CSPs can be solved without exploring the entire search space.

This algorithm was implemented in the ECL^{i}PS^e [14,110] which uses a PROLOG-based syntax. In principle any other PROLOG-based CSP solver could be used. The finite domain (fd) library [1] has been used as the underlying constraint solver.

Listing 4.1 shows a (partial) example PROLOG file for the draft metamodel and the flexible example model that appear in Figure 4.1. In lines 3-5, the relationships (references and aggregations are treated the same way) appearing in the metamodel are listed with their cardinalities using the *can_have()* fact. Its signature consists of the name of the class that owns the reference followed by the type of the class the reference points to. Finally, the upper and lower bounds of the reference are passed. For technical reasons, the many (*) upper limit is set to the value 500 but this could change to anything thought to be a large enough number for each domain. In lines 8-10, all the classes (both abstract and concrete) are instantiated as objects in the problem by creating an *object()* fact for each of them. In lines 13-15, concrete classes are defined by using a *concrete()* fact. In lines 17-19, the inheritance relationships between the classes are defined. The fact for the inheritance relationship is *direct()* in our solution. Its parameters are the name of the subclass and the name of the first-level superclass. If the type extends more than one other classes, separate *direct()* facts are created. This concludes all the information needed from the metamodel to construct the CSP.

The rest of the file includes facts about the example model. In lines 24-26, each node is assigned with a type using its distinctive id (in muddles each node by default gets a unique id). This is done by using a fact which we name as *is_type()*. Its arguments are the id of the node and its type. If the type is unknown, meaning that the node has been left untyped, then an "_" underscore is used, prompting the algo-

[1] `http://eclipseclp.org/doc/bips/lib/fd/index.html`

rithm to assign any valid type to this node. Finally, the edges between the nodes in the muddle are defined in lines 29-31 using the unique ids of the source and target nodes as parameters in the fact *has_a()*.

```
1 // Information from the metamodel
2 // Relations and cardinalities between types
3 can_have(zoo,animal,0,500).
4 ...
5 can_have(lion,tamer,1,1).
6
7 // Every class (abstract and concrete) is an object in the problem
8 object(zoo).
9 ...
10 object(lion).
11
12 // Define which classes are concrete
13 concrete(zoo).
14 ...
15 concrete(lion).
16
17 // Inhertiance relationships
18 direct(tamer,person).
19 ...
20 direct(tiger,animal).
21
22 // Information from the example model
23 // The type of each node. If not known then "_" is used
24 is_type(1,zoo).
25 ...
26 is_type(4,_).
27
28 // Links between the nodes
29 has_a(1,2).
30 ...
31 has_a(5,6).
```

Listing 4.1: An example PROLOG file automatically generated based on the draft metamodel and the flexible example model shown in Figure 4.1.

A straightforward approach that calculates the possible types that each type can be connected with could also be used to solved the same problem. This approach would be faster however it would have less predictive power. Consider the example presented in Figure 4.2. In the example model (see Figure 4.2b) the set of possible types using the aforementioned straightforward way for the node labelled as “4:?" would be “C”, “D” as the metamodel (see Figure 4.2a) indicates that nodes of type “B” should be connected with nodes of type “C” or of type “D”. However,

using the CSP method, the algorithm will identify that the node labelled “3:?" could only be of type “C” as nodes of type “A” can be connected only with nodes of type “C”. This results to the propagation of this decision to the suggested types for node “4:?", forcing prediction “C” to be removed from the set of the possible types as this would violate the rule that nodes of type “B” should be connected with no more than one nodes of type “C" and thus “D" becomes the only possible solution for node “4:?". This propagation of solutions in the scenario where a CSP is formulated leads to improved prediction performance and was selected over the straight forward approach.

(a) Example Metamodel (b) Example Model

Figure 4.2.: Example where a simple direct computation method of the possible connected types has less performance compared to the CSP approach.

4.3.2. Model and Metamodel to CSP Transformation

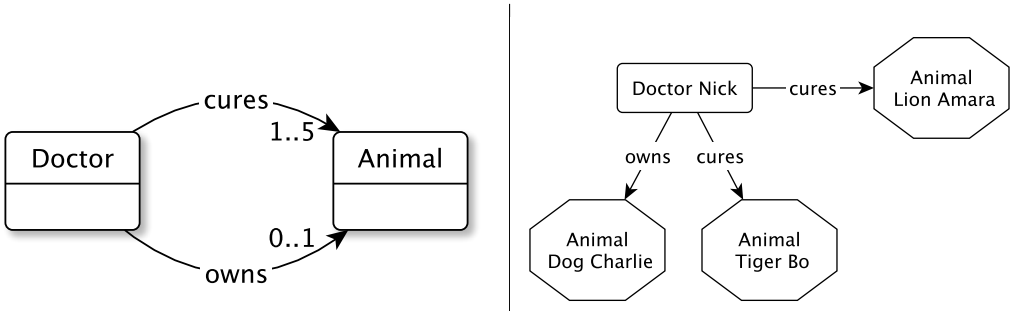
In this section, the algorithms that are used to transform the draft metamodel and the example model into the CSP are presented. In our approach, the algorithms were implemented in the Epsilon Generation Language (EGL) [65]. Using EGL, the CSP code is automatically generated and printed to a PROLOG file. Algorithm 5 presents the transformation of the rules contained in the draft metamodel to the necessary CSP rules while Algorithm 6 is the equivalent transformation for the example model.

In step 1 of Algorithm 5 (lines 1-25), all the data from the metamodel, like the concrete and abstract classes and references with their multiplicities, are collected and stored into sets and maps. These sets and maps are instantiated (are empty at this point) in lines 2-3. Then the algorithm iterates through all the classes (abstract and concrete) of the draft metamodel (lines 5-25). The set containing the names of all the types is populated with the name of the current class (line 6). If a class is not abstract, the set containing the types of the concrete types is amended with the type of the current class (line 8). Then the algorithm iterates through the references of the current class (lines 11-24). The name of the current class and the type the reference points to are stored to two variables in lines 12 and 13, respectively. The lower and upper bounds of the reference are also stored (line 14).

Algorithm 5 Transforming a metamodel to a CSP.

```
1: {Step 1: Collect concrete and abstract classes names and their multiplicities}
2: AllTypes, AllConcreteTypes ← {}
3: ReferenceLowerBoundsMap, ReferenceUpperBoundsMap ← {} ↦ {}
4: N ← set of all classes in metamodel
5: for all n ∈ N do
6:    AllTypes ← n
7:    if n not abstract then
8:       AllConcreteTypes ← n
9:    end if
10:   R ← set of all references of this class
11:   for all r ∈ R do
12:      SourceName ← n
13:      TargetName ← Name of the class this reference points to
14:      LowerBound, UpperBound ← Lower and upper bounds of the reference
15:      AlternativeName ← SourceName_2_TargetName
16:      if AlternativeName ∉ ReferenceLowerBoundsMap then
17:         ReferenceLowerBoundsMap ← (AlternativeName, LowerBound)
18:         ReferenceUpperBoundsMap ← (AlternativeName, UpperBound)
19:      else
20:         OldLowerBound, OldUpperBound ← Already stored bounds
21:         NewLowerBound ← OldLowerBound + LowerBound
22:         NewUpperBound ← OldUpperBound + UpperBound
23:      end if
24:   end for
25: end for
26:
27: {Step 2: Produce the CSP rules and facts based on the above collected data}
28: for all r ∈ ReferenceLowerBoundsMap do
29:   SourceType ← String before "_2_"
30:   TargetType ← String after "_2_"
31:   LowerBound, UpperBoung ← Stored bounds
32:   if n ∈ AllConcreteTypes then
33:      Print can_have(SourceType, TargetType, LowerBound, UpperBound)
34:      Print concrete(n)
35:      S ← Super types of n
36:      for all s ∈ S do
37:         Print direct(n, s)
38:      end for
39:   end if
40: end for
41: for all t ∈ AllTypes do
42:   Print object(t)
43: end for
```

In line 15, an alternative name for the reference is created. This is done to include scenarios where a class points to another class two or more times. Consider the example shown in Figure 4.3. A reference named "cures" and a reference named "owns" might be part of a "Doctor" class pointing to a "Animal" class in a meta-



(a) Draft metamodel

(b) Example model

Figure 4.3.: An example on amended multiplicities for the construction of the appropriate CSP rules.

model (see Figure 4.3(a)). In the example model (see Figure 4.3(b)), the domain expert has created one "Doctor" node and three "Animal" nodes, one of which is for the "owns" relationship. There are at least two approaches for constructing a CSP in this case. The first requires the CSP to relay on the *types* on the edges of the example model and construct different rules for the two different references (i.e., one rule for the "owns" and one rule for the "cures" references) with their appropriate multiplicities. The second does not require the types of the edges to be set in the example model as they are ignored in the CSP. In this second approach, one rule is created for denoting that "Doctors" are connected with "Animals". Thus, all the references in the draft metamodel between the same classes are combined to one. We construct this "super-reference" by amending the name of the source class with the name of the target class separated by the "_2_" string. For example, both the "owns" and "cures" references are now represented as one with the name "Doctor_2_Animal" reference. The lower bound (or upper bound) of this "super-reference" is calculated by aggregating the lower bounds (or upper bounds) for the two (or more) references. This is done in lines 16-23 of Algorithm 5. A unique rule is created for this "super-reference" in the CSP and stored in the available references. The sub-references consisting the "super-references" are discarded.

The rules of the CSP are printed in the PROLOG file in step 2 of the Algorithm 5 (lines 28-43) based on the data collected in step 1. The algorithm iterates though all the created references and extracts the source and target types (lines 29-30), while it gets the lower and upper bounds (line 31) previously calculated. A "can_have()" rule is printed for each reference (line 33). For the example of Figure 4.3 a "can_have(doctor, animal, 1, 6)" rule is created. If the class is concrete, the *concrete()* fact is created for this class (line 34). Inheritance relationships are modelled using the *direct()* fact (line 37). Finally, for implementation related reasons, each type in the metamodel (concrete or abstract) is modelled as an object by creating an *object()* fact in the CSP (lines 41-43).

At this point the algorithm has printed all the necessary rules and facts related

to the draft metamodel into the CSP file (lines 1-20 in the example file shown in Listing 4.1).

The process continues with transforming the example model into facts in the CSP by using Algorithm 6. The algorithm iterates through all the elements (lines 2-15); If an element is a node, then its type and id are stored (lines 4-8). If it is *untyped*, the "_" PROLOG wildcard is used to represent its type as described in Section 4.3.1. The *is_type()* fact is printed (line 9). If the element is an edge then the ids of the source and target nodes are stored (lines 11-12) and a *has_a()* fact is created (line 13).

At this point, the facts for the example model are printed and amended in the already created metamodel rules and facts (lines 22-31 in the example file shown in Listing 4.1). The CSP is now created and ready to be solved. Functions to solve the CSP using the ECLiPSe Constraint Logic Programming System [14, 110] are amended to the end of the file and are executed. The solution is saved in a text file, as described in the following Section 4.4.

Algorithm 6 Transforming a muddle to a CSP.

1: $E \leftarrow$ set of all elements in a muddle
2: **for all** $e \in E$ **do**
3: **if** e is $Node$ **then**
4: $NodeType \leftarrow$ Type of the node
5: **if** $NodeType$ is $Untyped$ **then**
6: $NodeId \leftarrow$ The id of the node
7: $NodeType \leftarrow$ "_"
8: **end if**
9: Print $is\_type(NodeId, NodeType)$
10: **else if** e is $Edge$ **then**
11: $SourceNodeId \leftarrow$ Source node id
12: $TargetNodeId \leftarrow$ Target node id
13: Print $has\_a(SourceNodeId, TargetNodeId)$
14: **end if**
15: **end for**

4.4. Experimental Evaluation

In this section, the experimental evaluation to assess the performance of the proposed approach is presented. The results and a discussion on both the quantitative and qualitative findings are also given. For the purpose of these experiments, the Muddles [4] approach will be used to express the example models. In principle, any other flexible modelling approach which provides the following minimal set of requirements could be used:

- provides a mechanism to extract the *types* of the (typed) *nodes* in the example models

- provides a mechanism to extract the *source and target nodes* and *multiplicities* of the *references/containments* in the example models

The following are the research questions looking for answers through the experiment that is presented in the following section:

- **RQ1:** How much is the search space for the correct type decreased by using the approach?
- **RQ2:** How much is the search space for the correct type decreased by using the approach if isolated nodes are not taken into account?

By the term *search space*, we refer to the number of candidate types the engineer has to pick from for an untyped node. Related to the hypothesis presented in Section 1.2, both these research questions will reveal if the approach offers, in a reasonable amount of time (no more than a few minutes), an acceptable reduction to the set of candidate types so it can feasibly be applied to a flexible MDE approach. Regarding **RQ2**, more details on the *isolated nodes* and why it is important to evaluate the approach on these scenarios as well, are presented in the following section that describes the experimentation process.

4.4.1. Experiment

In this section, the experiment run to evaluate the proposed approach is presented. An overview of the experimentation process is given in Figure 4.4.

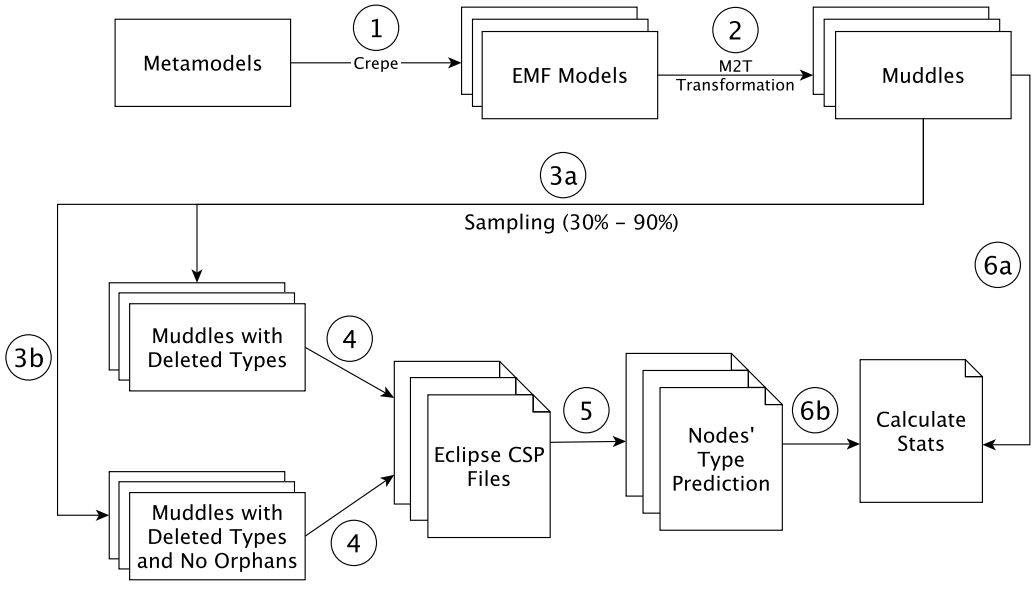


Figure 4.4.: An overview of the experimentation process for type inference using constraint programming principles.

The proposed approach was applied to a number of randomly generated models, each of which is an instance of one of the ten metamodels that were selected

in this experiment. The metamodels are the same used in evaluating the approach based on classification algorithms (see Chapter 3). These metamodels represent the draft/intermediate metamodel that the language engineers came up with from sets of muddle drawings, or inferred (semi-)automatically using one of the approaches presented in Section 2.4. For each of the metamodels, 10 models were randomly generated using the "Crepe" approach proposed in [155] (step (1) in Figure 4.4). These 100 randomly generated models are of varying size. The values of the attributes of the different classes of each model were randomly picked from a pool of characters/integers, as they do not affect the performance of the proposed approach. The fact that there is no muddles corpus available led us to the decision of using synthetic muddles based on randomly generated models. Any threats to validity of that decision are discussed in Section 4.5.

In step (2) the randomly generated models are transformed into muddles using the custom built M2T transformation of Algorithm 3 presented in Chapter 3. At this point, the constructed muddles contain nodes that have *all* their types assigned.

In order to simulate the scenario of having muddles with untyped nodes, a script runs on the example models and randomly deletes types from nodes (step (3a) and (3b)). It is of interest to identify whether the proportion of the untyped nodes affects the performance (in terms of the ability to infer the correct type) of the approach. Thus, 7 different sampling rates from 30% to 90% were selected. A 30% sampling rate means that 30% of the nodes had a type assigned to them whereas the rest 70% is the set of the nodes that are left untyped. The selected rates are the same as the ones used in the previous approach (see Chapter 3). In order to control for sampling bias which can affect the results positively or negatively (e.g., the type of the nodes picked for a simulated type deletion are those whose type can be easily predicted), the type deletion was performed 10 times for each sampling rate for each muddle. At the end of this step, a corpus of 7,000 different muddles was created (10 metamodels x 10 models = 100 x 7 sampling rates = 700 x 10 sampling repetitions = 7,000). In addition, we considered the scenario where some nodes are not only left untyped but are also left isolated (i.e., are not connected with any other node). Our approach cannot currently infer the type of such *orphan* nodes as there is no type assigned to them and they have no relationship with any other node on the diagram. For the second experiment (step (3b)), these nodes were removed from the muddle before commencing the prediction mechanism.

A file that contains the constraints associated with a model-metamodel pair is used by the ECLiPSe solver to identify the possible types for each untyped node. In step (4), the PROLOG files for each of the 7,000 muddles (and the 7,000 muddles with no orphan nodes) are automatically generated. An extract from such a file is shown in Figure 4.1. These files are consumed by the ECLiPSe solver (step (5)) and the results are stored in a text file. A single text file is generated for each

muddle and contains the mapping between the id of each node and the set of all the suggested types. An example is shown in Listing 4.2.

```
1:[zoo]
...
4:[fan]
5:[zoo, doctor]
...
12:[zoo, tamer, fan, doctor]
13:[zoo, doctor]
14:[doctor]
...
15:[fan]
16:[tiger, lion]
...
20:[tiger, lion]
```

Listing 4.2: An example file containing the returned results solving a CSP for type inference.

The performance of the approach is calculated by comparing the real type of each node and the types that are contained in the set of suggested nodes (step (6b)). The real type of each node for each muddle is kept in a text file before commencing the type deletion (step (6a)) to facilitate the comparison. The measures that are used in this work to assess the performance of the approach are discussed in the next section (Section 4.4.2) along with the results.

4.4.2. Results & Discussion

Before presenting the quantitative and qualitative results of running the experiment we provide an insight into the random models used as input. Table 4.1 summarises the corpus of metamodels and the generated models used for the experiment. More specifically, the number of concrete types (metaclasses) for each metamodel is given (column “#Types"). The minimum and maximum number of instantiated classes for the 10 generated models of each metamodel are provided in columns “Min" and “Max", respectively, followed by the average number of elements in these 10 models (column “Average"), for both the experiments with and without including the orphan nodes. As shown in the data, the smallest metamodel is the one used to describe a university professor, consisting of 4 concrete types. The largest metamodel is that of describing Wordpress CMS websites with 19 different metaclasses.

Performance Measurement

The purpose of the proposed approach is to identify the set of types that an untyped node can have and reduce the number of these suggested possible types by

Table 4.1.: Input data summary table for the experiment using constraint programming principles.

		With-Orphans			**No-Orphans**		
Model Name	#Types	Min Elements	Max Elements	Average #Elements in instances	Min Elements	Max Elements	Average #Elements in instances
Professor	4	46	71	56.3	37	54	45.2
Chess	5	41	74	57.8	18	40	28.9
Zoo	5	24	76	35.8	22	63	31.4
Ant Scripts	6	40	79	62.2	38	69	53.9
Use Case	7	41	80	52.7	35	68	45.6
Conference	7	43	80	64.5	36	61	50.7
Bugzilla	7	41	80	59.6	22	55	36.4
Cobol	12	23	30	25.9	22	30	25.2
BibTeX	14	40	79	64.4	31	58	47.2
Wordpress	19	22	45	35.7	19	42	31.9

applying the constraints that the draft metamodel includes. When the metamodel is not used for the type inference, the total number of possible types that a node can have is the number of the concrete types that appear in the metamodel. In the Zoo example of Figure 4.1, for instance, the number of possible types for each untyped node, is 5. An identified measure of the performance of the proposed approach is the following:

$$AverageSavingsPercentagePerMuddle = \frac{\sum_{i=1}^{n}(1 - \frac{TST_i}{TCT})}{n} \times 100 \qquad (4.1)$$

where n is the total number of nodes that are left untyped in the example model, TST_i stands for the Total Suggested Types returned by the CSP algorithm for the i-th node, while TCT stands for the Total Concrete Types in the metamodel. In the Zoo example of Figure 4.1, the Average Savings Percentage value is 73.3% as the individual savings values for nodes 4 and 7 are 0.8 (1 - (1/5)) and for node 6 is 0.6 (1- (2/5)). Thus, the average in this example model with 3 missing nodes is $(0.8 + 0.8 + 0.6)/3 = 0.733$. This value, which represents the performance of the approach, is interpreted as the reduction in effort required by the language engineer to pick the correct type for of each of the nodes that was left untyped. Thus, the greater the value the better the performance of the approach. A value of 0 means that the algorithm offered no savings at all because the suggested types are equal to all the possible types included in the metamodel.

Quantitative Analysis for With-Orphans Experiment

Table 4.2 presents the results of the average total savings for each of the 10 metamodels used in the experiment, for the 7 different sampling rates. The results are averaged as each metamodel had 10 random models generated. Moreover, for each sampling rate the type deletion was performed 10 times to avoid the case of a lucky or an unlucky sampling. For example, the highlighted value of 67.39 means that the average savings percentage for 100 example models (10 models x 10 type deletion sessions for each) which have 60% of their types known for the Usecase metamodel, is 67.38%. Three metamodels are marked with asterisks. These are the metamodels for which not all the 700 runs were finished in reasonable time due to large statespace. The reasons behind that and more details are discussed in the qualitative analysis subsection.

Table 4.2.: Average savings results table (With-Orphans)

		Average Total Savings Percentage for Different Sampling Rates (With-Orphans)								
Model Name	**#Types**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Profesor	4	63.18	63.09	62.93	63.12	63.36	63.33	63.13	63.161	0.39
Chess	5	39.96	39.78	40.34	40.04	40.10	41.61	38.49	40.045	0.11
Ant*	6	60.58	62.29	61.90	63.09	62.54	62.61	61.98	62.141	0.46
Zoo	6	69.05	68.38	68.91	68.54	69.25	68.75	69.60	68.925	0.43
Bugzilla	7	19.67	18.98	19.90	20.19	19.68	19.21	18.93	19.510	-0.29
Conference	7	67.11	67.79	67.27	67.76	67.80	66.88	67.23	67.406	-0.14
Usecase	7	67.34	67.36	67.89	**67.39**	67.35	67.33	67.65	67.473	0.07
Cobol*	12	75.28	75.70	75.72	76.19	76.04	75.81	78.35	76.156	0.86
Bibtex	14	49.28	49.02	49.48	48.85	49.74	48.76	48.70	49.120	-0.54
Wordpress*	19	79.12	80.18	80.77	82.08	81.91	80.24	81.83	80.876	0.57
	Avg.	59.06	59.26	59.51	59.72	59.78	59.45	59.59		
	Cor. 2	0.39	0.39	0.39	0.39	0.39	0.39	0.39		

By assessing the raw results, the average savings for the experiment vary from 19.51% (for the Bugzilla metamodel) to 80.88% (for the Wordpress metamodel). For most of the metamodels in the experiment, the average savings percentage varies between 60% to 70% which is arguably a significant amount of effort saved. This means that about two thirds of the available types were pruned, returning a significantly smaller set of possible types to the language engineer to pick from for each of the untyped nodes.

It is of interest to identify if the proportion of untyped nodes in the example model and the size of the metamodel (i.e., number of the available types for suggestion) affect the savings score. Correlations "**Cor. 1**" and "**Cor. 2**" below express the above two questions.

Cor. 1: How strong is the dependency between the sampling rate and the average savings score?

Cor. 2: How strong is the dependency between the number of types in a metamodel (size of metamodel) and the average savings score?

The correlation coefficient values for each of the 10 metamodels for the first and second question are given in column "Cor. 1" and row "Cor. 2" of Table 4.2, respectively. From these values, two conclusions can be safely made:

1. The savings percentage is not correlated to the size of the metamodel. There are large metamodels (i.e., Wordpress, Cobol) where the scores are high while in others (i.e., BibTeX) the score is significantly lower. In the same manner, there are small metamodels (i.e., Professor, Zoo, Ant Scripts) where the savings are high while in other small metamodels (i.e., Chess) the results are lower. The same applies to the mid-sized metamodels (i.e., Use Case and Conference vs. Bugzilla). The correlations coefficient values verify this visual observation.

2. The savings are not affected by the sampling rate. That means that no matter the number of nodes left untyped, the performance remains the same. The correlation coefficient values are fluctuating a lot based on the metamodel. There is negative correlation starting from -0.54 for the Bibtex metamodel, going up to 0.86 positive correlation for the Cobol metamodel. Even in cases where the correlation is strong, the actual changes in savings are very small to be considered a result of the change in the sampling rate rather than the random sampling itself. This is an expected result as the CSP algorithm used is not based on machine learning techniques. Hence, the amount of knowledge that is available (i.e., the number of known nodes) does not affect its performance. This behaviour was identified in the previous proposed approach presented in Chapter 3 where there was a significant improvement in the prediction scores in higher sampling rates.

As described in Section 4.2, one of the differences of this work with the one based on classification algorithms (see Chapter 3) is the fact that the algorithm returns a set of suggested types for each node rather than a prediction for the most probable one. The trade-off is that the correct type is guaranteed to be in the list of the suggested types. In the "With-Orphans" experiment, there were about 155,000 nodes left untyped. For all these nodes, the list of the suggested types included the correct type verifying the previous argument.

It is of interest to assess how many types are returned as suggestions for each of the nodes that are left untyped. Figure 4.5 presents a histogram to help explore this. For 38% of the nodes, there is exactly one type returned. This is very important because for more than the one third of the nodes this approach automatically predicted the correct type of the node; there was no need of verification or extra help by the language engineer. Although in the type inference approach based on classification algorithms about 80% of the node types were predicted correctly, there was

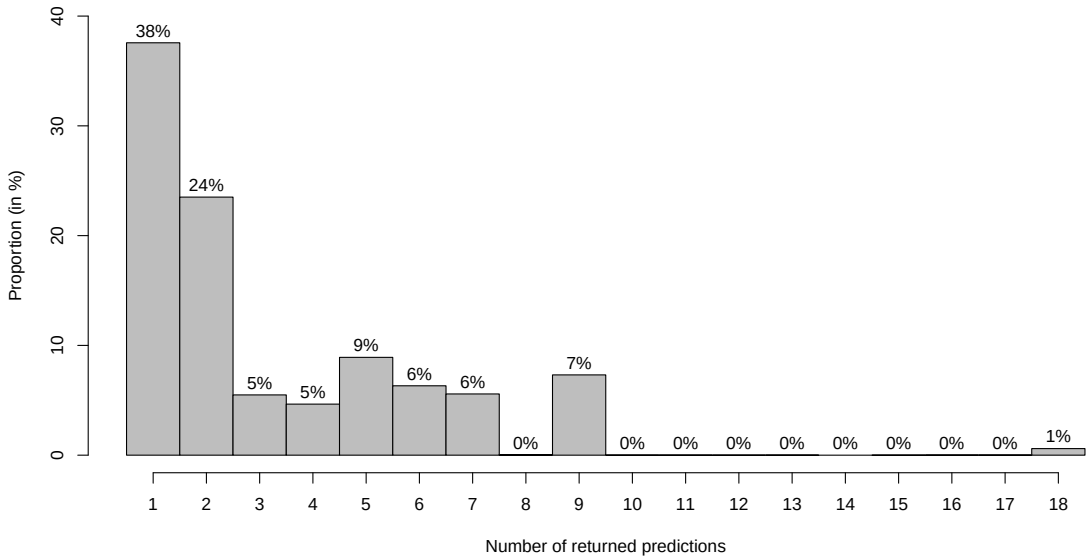
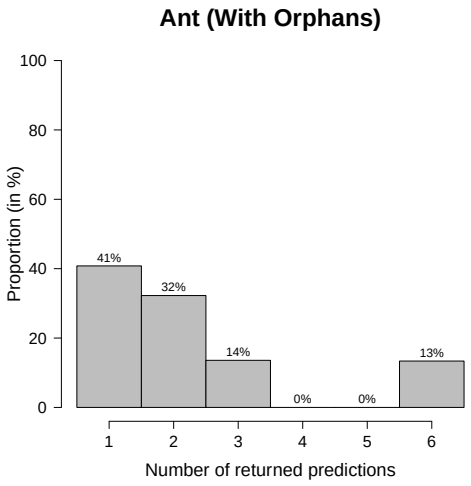


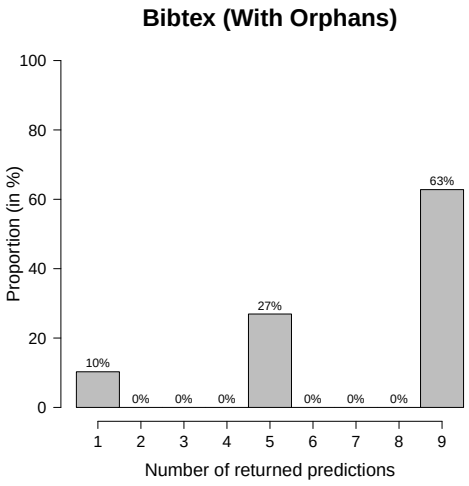
Figure 4.5.: Histogram for the number of suggested types for each node that is left untyped in the With-Orphans experiment.

no guarantee about the correctness of the prediction meaning that the language engineer had to verify the prediction manually for each single node. In addition, by analysing the same histogram, one can see that for about the two thirds (67%) of the nodes, the algorithm suggests 3 or fewer types from which the language engineer has to select the correct type reducing the amount of effort needed to a minimum.

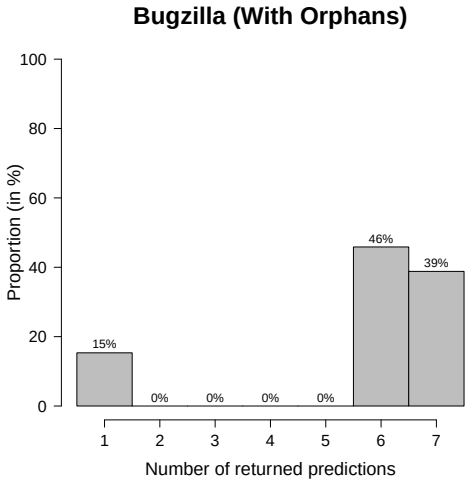
Because some metamodels contain fewer types than the maximum values appearing in the histogram of Figure 4.5, separate histograms for each metamodel are provided in Figure 4.6. For some metamodels like Ant, Profesor, Usecase, Zoo and Wordpress, the returned results contain either one or two types for more than the 70% of the untyped nodes. This is very important, especially for larger metamodels (e.g., Wordpress - see Figure 4.6(i)) because: a) the type can be assigned automatically for more than half of the nodes; and b) the savings against the total possible types of the metamodel are significant for about the one third of the rest where the engineer has to pick between two possible types. In two metamodels (i.e., Chess and Bugzilla), the savings cannot be considered as significant as for the rest. Especially in the Bugzilla metamodel (see Figure 4.6(c)), for 85% of the elements there is no or small type pruning provided by the algorithm. For 39% of the untyped nodes, the size of set containing the suggested nodes is equal to the number of types the metamodel has (7) and for 46% only one type is removed from the set of possible. Possible explanations for this behaviour are provided in the qualitative analysis section.



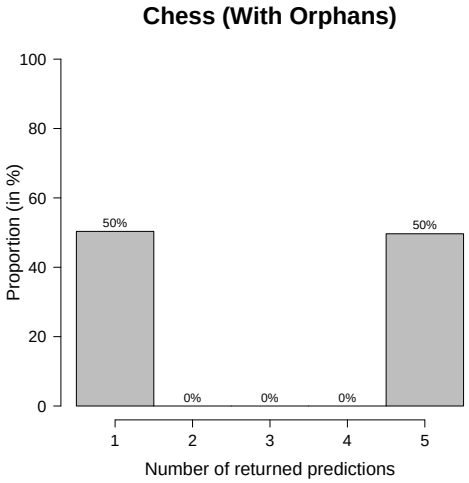
(a) Ant



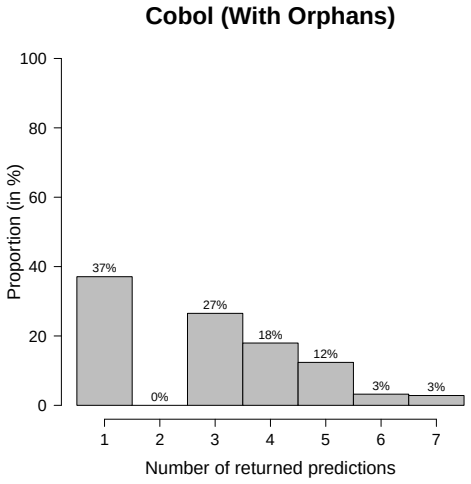
(b) Bibtex



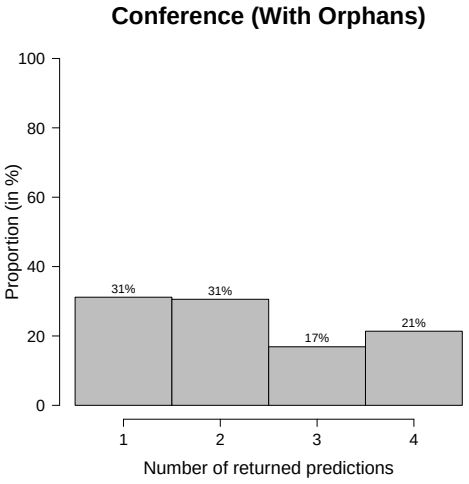
(c) Bugzilla



(d) Chess

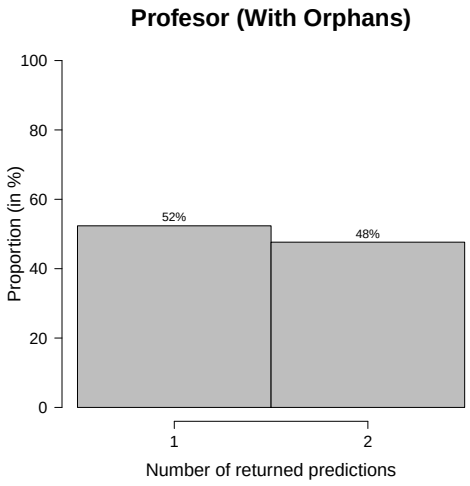


(e) Cobol

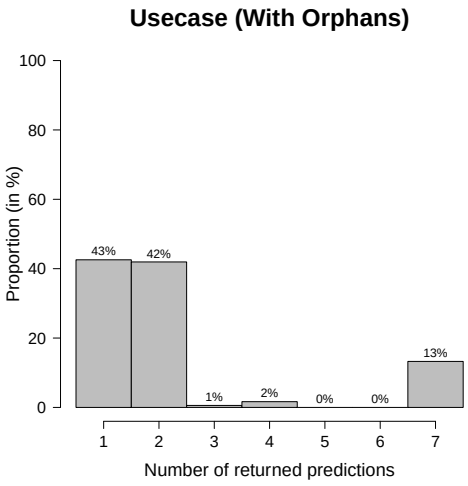


(f) Conference

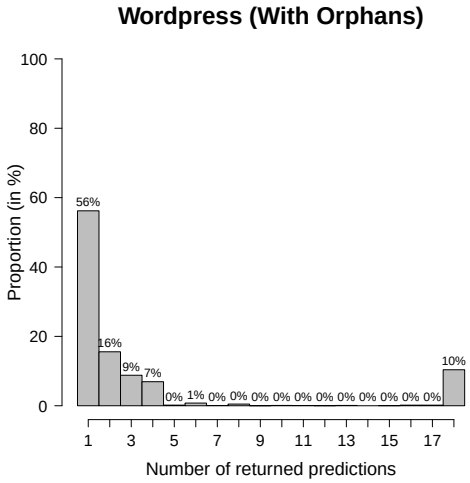
Figure 4.6.: Number of returned predictions for each metamodel.



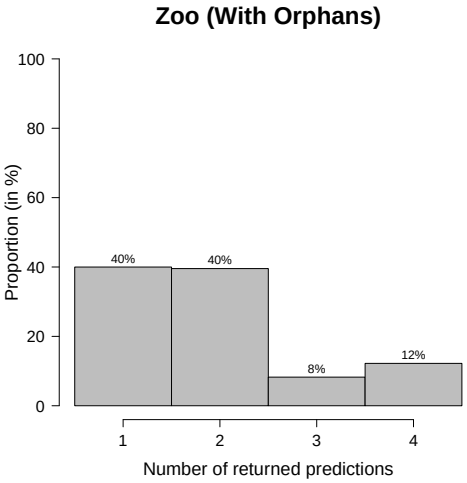
(g) Profesor



(h) Usecase



(i) Wordpress



(j) Zoo

Figure 4.6.: Number of returned predictions for each metamodel (Continued).

Quantitative Analysis for No-Orphans Experiment

In this section, the results of running the experiment having the orphan nodes removed are presented. A summary of the results is given in Table 4.3. As described for the "With-Orphans" experiment, the table presents the average savings percentage for the 100 runs for each metamodel and sampling rate. For example, the highlighted value of 72.99 in Table 4.3 means that for the Zoo metamodel, the saving percentage on average was 72.99% between the 10 random models instantiated from it, when 60% of the nodes were typed (i.e., 40% left untyped).

The results suggest that among the different metamodels, the minimum performance was in the Bugzilla (31.6%) and the maximum in the Wordpress metamodel (90.43%).

Table 4.3.: Average savings results table (No-Orphans)

		Average Total Savings Percentage for Different Sampling Rates (No-Orphans)								
Model Name	**#Types**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Profesor	4	66.29	66.40	66.50	66.32	66.32	65.89	67.46	66.453	0.18
Chess	5	80.00	80.00	80.00	80.00	80.00	80.00	80.00	80.000	NA
Ant	6	70.92	71.32	71.78	71.96	72.36	72.96	72.01	71.899	0.89
Zoo	6	73.07	73.18	73.14	**72.99**	73.68	73.52	73.89	73.353	0.71
Bugzilla	7	31.82	32.47	32.67	31.23	31.75	30.19	31.11	31.605	-0.79
Conference	7	74.04	73.98	73.55	73.69	73.93	74.12	74.26	73.939	0.43
Usecase	7	77.67	77.57	77.73	77.57	77.49	77.36	77.97	77.624	-0.14
Cobol	12	75.67	76.21	76.81	76.72	77.37	77.38	76.75	76.702	0.71
Bibtex	14	44.34	44.24	44.76	43.89	44.55	43.35	44.56	44.241	0
Wordpress	19	88.05	89.94	90.36	90.86	91.18	91.19	91.46	90.433	1
	Avg.	68.19	68.53	68.73	68.52	68.86	68.60	68.95		
	Cor. 2	0.17	0.17	0.17	0.17	0.17	0.20	0.17		

The same correlations defined in the "With-Orphans" experiment, are assessed here, too. Regarding **"Cor. 2"** (see row "Cor. 2" in Table 4.3), the same behaviour as in the "With-Orphans" experiment is identified: there is no correlation between the size of the metamodel and the saving percentage. Regarding **"Cor. 1"**, we find a large fluctuation among the different metamodels. In some, there is positive correlation, meaning that fewer missing types lead to higher effort saving percentage, whilst in other there is a negative correlation. That does not provide with evidence that sampling ratio affects the performance of the proposed approach.

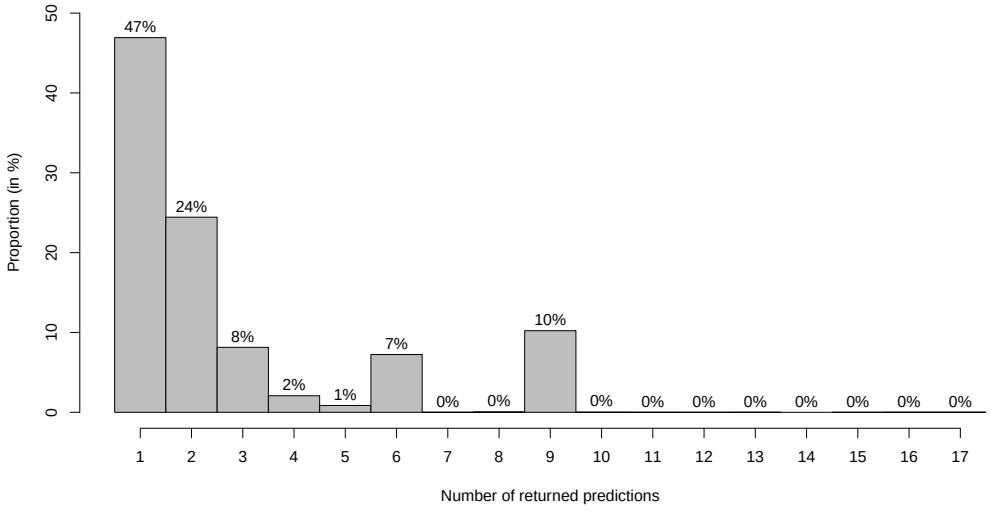
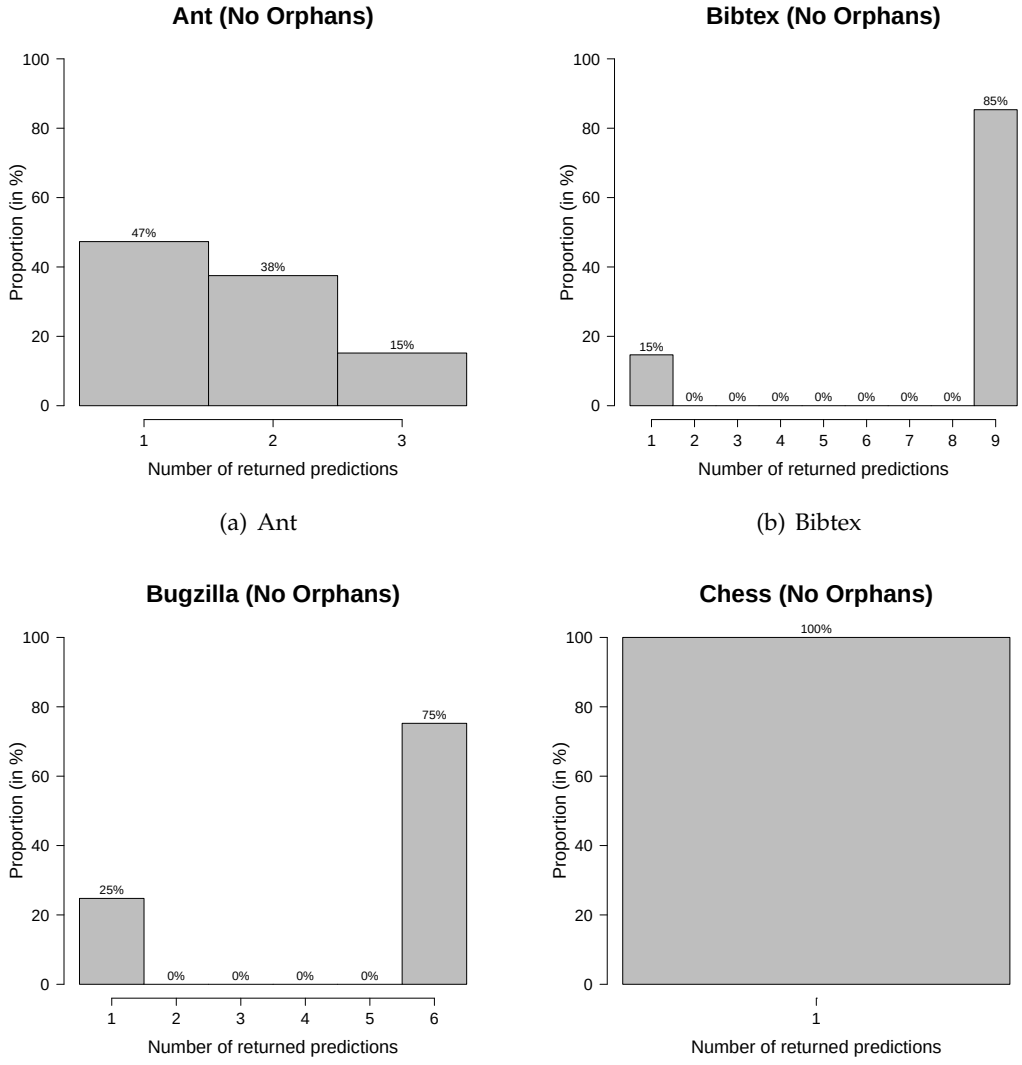


Figure 4.7.: Histogram for the number of suggested types for each node that is left untyped in the No-Orphans experiment.

The histogram in Figure 4.7 shows the size of the returned set containing the suggested types for each of the about 109,000 untyped nodes in the "No-Orphans" experiment. For 47% of the nodes, there was only one type returned. Thus, a

guaranteed correct type could be assigned automatically. Cumulatively for the 79% of the untyped nodes the returned set contained 3 or less types, in this experiment.

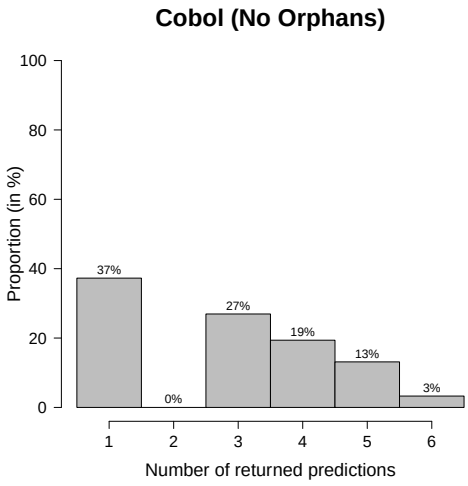
Histograms containing the number of returned types for each metamodel are given in Figure 4.8. The same behaviour with the "With-Orphans" scenario (see Figure 4.6) is also identified here. For some metamodels (i.e., Ant, Profesor, Usecase, Zoo, Wordpress), the savings are significant. Two more metamodels (i.e., Conference and Chess) can be added to this list in the "No-Orphans" scenario. Especially for the latter (see Figure 4.8(d)), all the untyped nodes can be automatically filled with their type as the size of the set containing the suggested type is 1 for all the untyped nodes. Possible reasons for that are discussed in the comparison section that follows. For the Bugzilla metamodel (see Figure 4.8(c)) there was some improvement but still for three quarters of the untyped nodes the assistance provided is minimal.



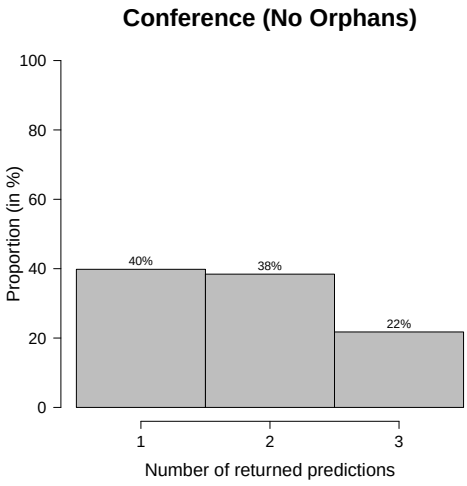
(a) Ant (b) Bibtex

(c) Bugzilla (d) Chess

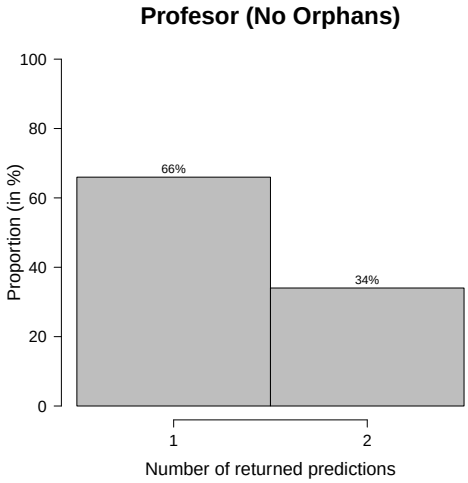
Figure 4.8.: Number of returned predictions for each metamodel.



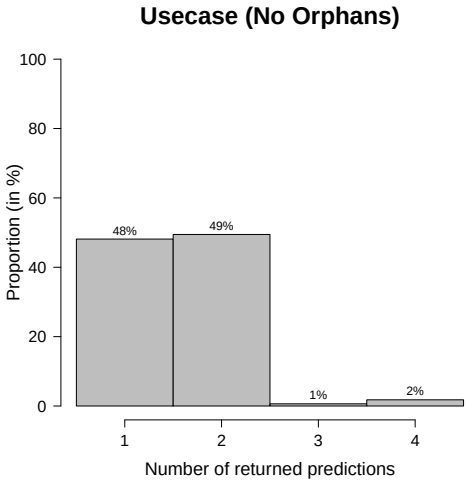
(e) Cobol



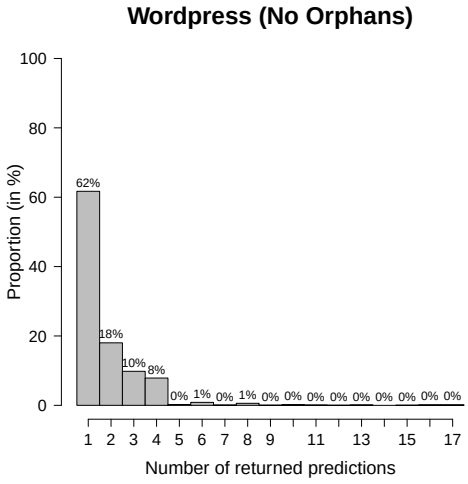
(f) Conference



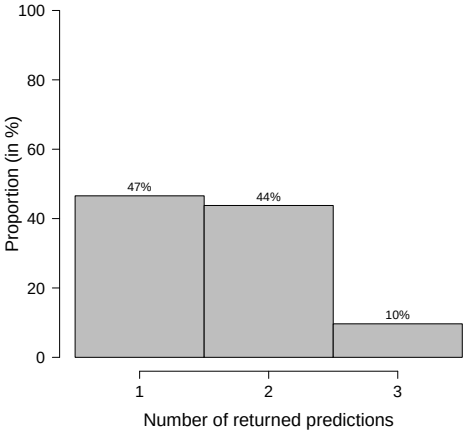
(g) Profesor



(h) Usecase



(i) Wordpress



(j) Zoo

Figure 4.8.: Number of returned predictions for each metamodel (Continued).

The results for both algorithms answer both research questions **(RQ1 & RQ2)** set at the beginning of this experiment. For both scenarios ("With/No Orphans") the reduction in the set of candidates is significant for almost all the cases, regardless the percentages of nodes that are left untyped. There are some exceptional metamodels mentioned above that the reduction is not significant, but still acceptable. The reasons for this behaviour are investigated in the qualitative analysis section of this chapter that follows. In almost all the cases (except for some models of the metamodels marked with asterisks in the results tables) the calculation was performed in a few (milli-)seconds. Thus, the approach can be feasibly applied into the workflow of a flexible MDE approach as mentioned in the hypothesis of this project.

Comparison & Qualitative Analysis

In this section a comparison between the "With-Orphans" and "No-Orphans" scenarios is presented followed by a qualitative analysis of the results.

In the CSP defined and presented in Section 4.3.1, all the constraints are related to the references that each type has in the draft metamodel. Thus, it is expected that for isolated nodes (i.e., those that are not connected with any other) there is no significant pruning that can be done. This expectation is verified by the experiments ran. By comparing Tables 4.2 and 4.3 one can see that for 8 of the metamodels there was an improvement in the average total saving percentage. For the Cobol metamodel, there was a small improvement while for the Bibtex metamodel the performance was worse. This is explained by the histograms produced for this metamodel for both scenarios (see Figures 4.6(b) and 4.8(b)).

The savings percentage in the "With-Orphans" scenario was better because a number of nodes (about 27% - see Figure 4.6(b)), for which the size of the set containing the possible suggested types was five, were contributing positively to the averages. In a metamodel that contains 14 types, pruning them down to five for some nodes it is an improvement in the average savings percentage value. However, in the "No-Orphans" scenario, these nodes completely disappear (see Figure 4.8(b)), and thus the average performance is worse. This behaviour is contradictory to our expectation regarding the removal of orphan nodes but it is explained by examining the Bibtex metamodel (see Figure A.2). There is a number of the available types that extend the "AuthoredEntry" class. This class, and accordingly, all of its children require at least one reference going to a node typed as "Author". Orphan nodes have no references at all, thus the algorithm was discarding from the set of possible types all those extending the "AuthoredEntry" class improving the results.

In contrast, in the Chess metamodel, the savings were maximised in the "No-Orphans" experiment. By looking at the separate histograms for "No-Orphans"

scenario (see Figure 4.8(d)), the algorithm returns 1 type for all the nodes, where in the alternative scenario (i.e., "With-Orphans") that value was down to 50% (see Figure 4.6(d)). By examining the random generated models and the Chess metamodel (see Figure A.4) we see that there are only optional references (i.e., lower bound is 0). Consequently, the set of possible types for the orphan nodes cannot be pruned like in the Bibtex scenario.

Regarding the Bugzilla metamodel the approach's under-performance is explained by the fact that there are six classes (see Figure A.3) that are having no outgoing reference and one (the same) incoming edge from the "Bug" class which is of "*" multiplicity. This gives no extra constraints for the approach to reduce the number of possible types as everything that it is connected with a class of type "Bug" can be any of the above six types.

The time needed for the algorithm to predict the possible types for all the missing nodes in an example model takes from a few milliseconds up to a few seconds. There were a few experiments for three of the metamodels which were not finished in reasonable time. Table 4.4 presents the number of experiments that were not finished, and thus not included in the results. In all six cases, the unfinished experiments are mostly part of the 30% or 40% sampling rate simulations. Any threats to validity are discussed in Section 4.5.

Table 4.4.: Number of unfinished experiments in the type inference approach using constraint programming principles evaluation.

Metamodel	**With-Orphans**	**No-Orphans**
Ant	86	13
Cobol	3	2
Wordpress	44	3

In the following, we discuss the causes of these timeouts. The execution time of Algorithm 4 depends on the *size* of the model and the associations and multiplicity constraints. Regarding model size, considering a model with n untyped nodes and a metamodel with m concrete types, the number of potential type assignments grows exponentially in the order of n^m. Hence, the number of untyped nodes and types has an impact on the efficiency of the solver. This happens because the majority of experiments with a timeout are those with the highest rate of untyped objects (e.g., scenarios like the 30% or 40%). Furthermore, the fact that the number of CSPs that needs to be solved in Algorithm 4 grows with the size of the (untyped elements in the) model and (concrete types in the) metamodel, makes the impact of the problem size even more significant.

Nevertheless, there is another factor that plays a larger role in the efficiency of the solver which is the number and restrictiveness of the constraints. This happens because the solver takes into account the constraints when searching for a solution,

using them to prune the search space. Hence, in CSPs where constraints are very tight (e.g., tight bounds for multiplicities), the solver will be able to discard large sections of the search space without needing to explore them. Conversely, in CSPs with few constraints or where constraints are loose, most solutions will satisfy the CSP and the solver will again complete the search quickly.

However, there is a certain threshold between those two extremes where solving the CSP becomes more complex and requires exponential runtime. Within this threshold, constraints discard many solutions, forcing the solver to evaluate many candidates but at the same time pruning is not effective enough to avoid exploring most of the state space. This phenomenon is well known and has been observed empirically in random CSPs [166–168], i.e., CSPs with random constraints. A priori, it is not possible to predict when a CSP will fall between this threshold.

Performance Analysis

Table 4.5 summarises the average execution time for each of the two model sets ("With-Orphans" and "No-Orphans") used in this approach. The specification of the machine used to run the experiments is the following:

- Architecture: x64 (64-bits)
- Processor: Intel(R) Core(TM) i5-4288U CPU @ 2.60GHz
- RAM: 2x8GB DDR3 @ 1600 MHz
- Hard Disk Drive: 256GB PCIe SSD
- Operating System: Mac OS X 10.11.6

Table 4.5.: Average execution time for each metamodel in the CSP approach

		Average Execution Time for Different Metamodels (in seconds)	
Model Name	**#Types**	**With-Orphans**	**No-Orphans**
Profesor	4	0.02	0.02
Chess	5	0.01	0.01
Ant	6	1.95	17.03
Zoo	6	0.11	0.13
Bugzilla	7	0.10	0.08
Conference	7	0.25	0.19
Usecase	7	0.18	0.12
Cobol	12	1.27	2.33
Bibtex	14	0.78	0.78
Wordpress	19	14.45	8.79

As one can see from Table 4.5, the average execution times vary from 10 milliseconds up to 17.03 seconds. As described in the previous section, the number of types in the metamodel affects the time needed from the solver to provide a solution to the CSP however, this is not the only and the most significant factor as the number and the restrictiveness of the constraints play important role. As a result, although there is a trend in having increased execution times while types are increasing we cannot generalise and claim that this is the only factor. Thus, from these average execution times, it is not clear if scalability is an issue for the applicability of this approach as other factors might affect the execution time as well.

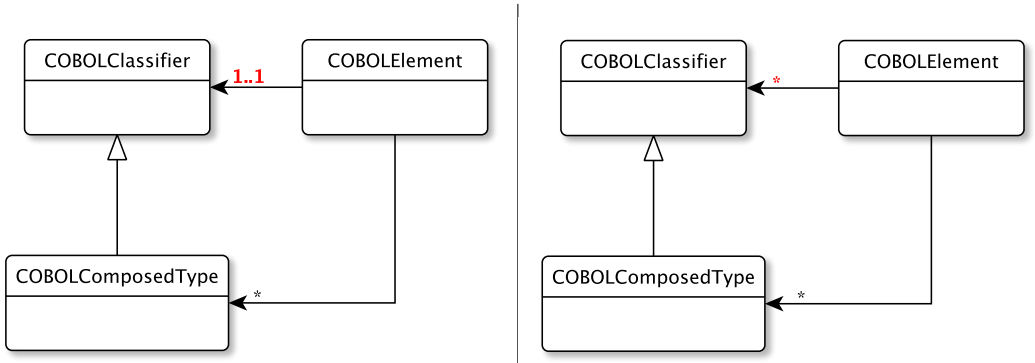
Comparing to the approaches presented in Chapter 3, the approach presented here performs quite slower in terms of time. However, it is still applicable if one thinks that in 7 out of 10 metamodels in the both (“With-Orphans" and “No-Orphans") scenarios the prediction finishes in less than 1 second. However, as described in the previous section, there were some examples where the prediction has not finished in a reasonable amount of time.

4.5. Limitations

As discussed in Section 4.4.2, the effort saving results are not affected by the number of missing types. Moreover, the number of the missing experiments is not significant for 5 out of the 6 cases where the experiments did not finish within reasonable time (with the exception of the orphans scenario for the Ant metamodel). As a result, we do not have reasons to believe that the experiments which were not complete affected the validity of our experimental evaluation.

In the cases where a class is extended by one or more other classes we need to accumulate the existence of this class’ children in the drawing to check if its upper and lower cardinalities are fulfilled. For instance, in the example of Figure 4.1, each element of type “Doctor" must treat no more than 5 elements of type “Animal". The constraint programming algorithm aggregates the instances of “Lion" and “Tiger" that each “Doctor" is connected with to validate if the constraint for the “Animal” class is fulfilled. However, the way the example models are represented in our algorithm does not make it feasible to include a corner case where a class/type is connected with the parent class while it is also connected with one of its children with a reference that has fixed cardinalities (Figure 4.9).

This corner case is found in two types in total in our experiments (one type in the Cobol metamodel and one type in the Wordpress metamodel). For this corner case, the Muddle-to-Text generator that produces the PROLOG constraints and commands raises the fixed multiplicity constraint for the class-to-parent reference to many (*). This impacts the effectiveness of the proposed approach as suggested types that normally would be rejected due to this multiplicity constraint are included in the list of suggested types decreasing the savings effort figures presented



(a) Original cardinality. (b) Changed cardinality.

Figure 4.9.: Example of a corner case scenario.

in the results of the experiment. An interesting logic paradigm called *Answer Set Programming (ASP)* [169, 170] could be used to tackle this problem. Modern ASP solvers use conflict-driven algorithms [171], promising improvements in performance.

The constraints are checked and applied to the example models by checking edges going from one node to the other. The names of the edges are *not* taken into account, thus, the definition of the names of all the edges that appear on the diagram is not a requirement for this approach to work. In contrast, if an edge is defined wrongly (i.e., connects to elements that should not be connected according to the draft metamodel) then the CSP will not be able to be solved as the mistaken edge will be a violation to the rules contained in the algorithm. The algorithm points to the node/edge that violates the rules. As a consequence, this behaviour has the potential to be exploited to turn the CSP algorithm into a validation tool for example models in the same manner it is used in [145].

Finally, in the experiment 10 metamodels from different domains were used to generate random models which are then transformed into muddles. We do not believe that the use of models that conform to metamodels rather than using muddles will directly have a significant impact in the performance of the approach. In this approach, the references of each type and their multiplicities are the only constraints applied in the CSP. The fact that all the metamodels have optional references (references which lower bound is 0) lead in the creation of random models that will not always instantiate these. Therefore, the models already contain noise that a muddle could contain. However, it would be of interest to inject more noise to the generated models in the manner done for the "Sparse" experiment, where classification algorithms are used for type inference (see Section 3.5.1), to identify how much such type of noise affects the performance of the proposed approach. Directions for future work are presented in Section 6.2.

4.6. Chapter Summary

In this chapter we proposed the use of Constraint Programming to facilitate language engineers infer the types of nodes that were left untyped in the example models. Specifically, a Model-to-Text transformation is used to automatically translate example models and constraints related with references and their multiplicities appearing in draft metamodels to a Constraint Satisfaction Problem (CSP). By using ECL^{i}PS^e [14, 110], a CSP solver, language engineers get back a set containing all the possible types a node can have based on the constraints proposed in the metamodel.

In order to test the proposed approach, we run two experiments on a large number of example models. The first took into account "orphan" nodes that were isolated (i.e., they were not connected with any other node) and a second that excluded these. The results suggest that a minimum of 19.51% of reduction in effort saving was occurred. In some cases the total effort saving was up to 80.87%. As expected, the average effort savings were improved in the scenario were the orphan nodes were removed from the experiment (values range from 31.60% to 90.43%). An analysis based on the number of the possible types for each untyped node, the approach returned, showed that for more than one third of the nodes, only one possible type was identified, and thus an automatic assignment could be done. For a great proportion of the remaining nodes, the returned types were less than three, reducing the effort needing for picking the correct type significantly.

Finally, no evidence was found to extract safe conclusions if the proportion of the types missing in the example models and the number of types in the metamodel affect, positively or negatively, the performance of the algorithm.

CHAPTER 5

Type Inference using Graph Similarity

5.1. Introduction

During the exploratory stages of DSL definition where a flexible MDE approach is followed, the language engineers may come up with a draft version of the envisioned DSL's metamodel based on the domain knowledge that is already exposed in the example models. This process is depicted in Figure 1.1. Following this iterative manner, a richer understanding of the domain is acquired incrementally. As described in Section 2.4 the *draft metamodel* inference can be done either automatically (e.g., using the metamodel inference mechanism of metaBUP [5, 29]) or manually. This neither implies that this draft metamodel is a final version of the envisioned DSL, nor that it follows the best practices and DSL patterns available. It may even be a flat metamodel with no abstractions and grouping of concepts (e.g., elements that have the same attributes most probably should extend the same class). This draft metamodel can prove useful for type inference. Moreover, flexible MDE tools do not enforce conformance to this metamodel. Therefore, despite of the existence of the metamodel, one can still have untyped nodes in the example model. In this chapter, an approach to type inference that exploits the similarity of untyped nodes in the example models with the elements defined in this draft metamodel based on the "string similarity" of their features, is proposed. More specifically, a state-of-the-art algorithm called *Similarity Flooding* [13] is used to predict the types of the nodes in the example models that are left untyped by comparing the names of their attributes and relationships.

Comparing with the inference approaches in metaBUP [5] and Flexisketch [6], one can understand that the approach proposed here differs from them as it re-

quires the existence of a draft metamodel; this is not a requirement for either metaBUP [5] or Flexisketch [6]. A draft metamodel was also required in the approach presented in Chapter 4 where types of nodes were inferred using CSP principles. However, the approach presented here does not rely on the concrete syntax of the example models as the aforementioned approaches do. As a result, language engineers and domain experts do not need to follow specific concrete syntax conventions, in line with the fact that they may not be ready to do so in the exploratory phases of the domain understanding.

The approach proposed here similar to the approach presented in [11] where the similarity flooding algorithm [152] is used to automatically produce the rules for model-to-model transformations. A graph representation that was used in [11] (see Section 2.8.2) is used in this chapter. However, the approach proposed in [11] identifies similarities between two metamodels, while in our approach, similarities between a metamodel and an example model are of interest.

The rest of the chapter is structured as follows. In Section 5.2 the proposed approach is presented. Section 5.3 demonstrates the graph representations that are used as input to the similarity flooding algorithm in this research. In Section 5.4, a detailed description of the similarity flooding algorithm is provided. Sections 5.5.1 and 5.5.2 summarise the experiment that was run to evaluate the proposed approach and discuss the results, respectively. Finally, Section 5.6 outlines the limitations of the approach.

5.2. Type Inference Using String Similarity

In this section the proposed approach to type inference is described in details. An overview of the approach is depicted in Figure 5.1.

Domain experts and language engineers use a simple drawing tool to express example models of the envisioned metamodel (step (1)). Then, from these example models a *draft metamodel* can be extracted either manually or (semi-)automatically using one of the approaches described in Section 2.4 (step (2)). At this point the example model may contain nodes that were left untyped by the engineers for the reasons explained in Section 1.1.1.

The proposed automated approach starts from step (3). Both the inferred metamodel and each of the example models are transformed into directed graphs which are used as input to the similarity flooding algorithm. The transformation is performed by executing a model-to-text transformation following rules whose definition is based on the graph representation chosen for this work. More details on the selected graph representation are given in Section 5.3.

The generated graphs from the draft metamodel and the example model are given as input to the similarity flooding algorithm (step (4)). The algorithm follows a fixpoint computation process to produce a list containing the similarity values for

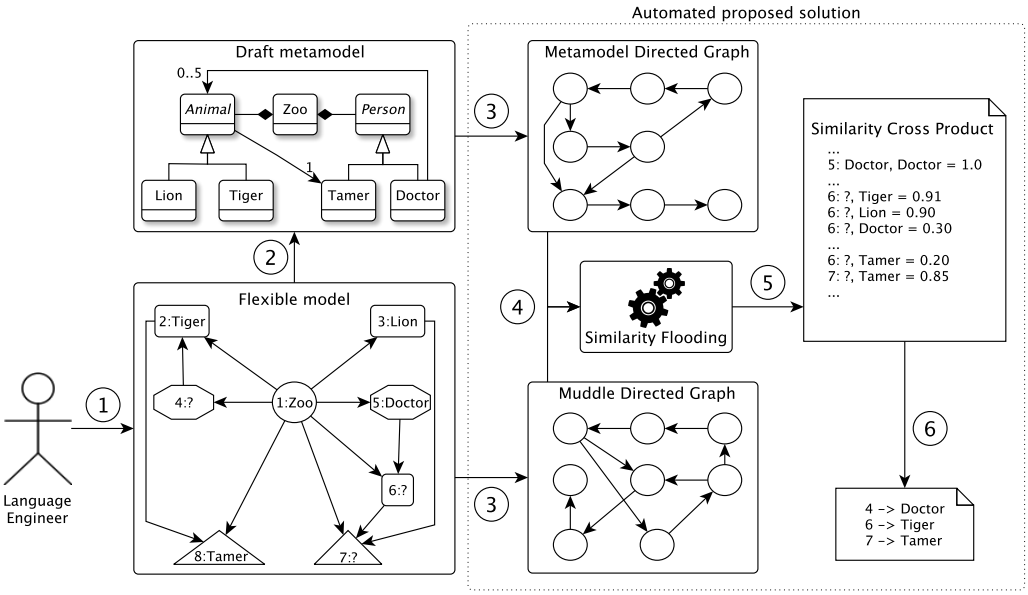


Figure 5.1.: An overview of the proposed approach to type inference based on similarity flooding algorithm.

the nodes' cross product (i.e., for each pair of nodes) appearing in the input graphs (step (5)). The similarity flooding algorithm is presented in Section 5.4. Finally, the metamodel's type with the best similarity value for each model's untyped nodes is selected as the proposed type for the node (step (6)).

This approach works under the *Closed World Assumption (CWA)* [163] meaning that an untyped node could only be of a type that is already defined in the metamodel. This explains why in our approach we are interested in similarities between the model's and metamodel's nodes; the type of the untyped node is one of those appearing in the metamodel. If the correct type of a node is one that does not appear in the metamodel, thus the CWA is not met, then the proposed approach will not be able to predict it correctly.

The approach proposed in this chapter produces an *ordered* list containing possible types from the untyped nodes in the example models. This approach has certain advantages and disadvantages when compared to the CSP approach proposed in Chapter 4. The latter can be applied in the same scenarios where a draft metamodel is already inferred. One advantage is that even if typing errors exist in the labels of some nodes and references, the approach from this chapter is still able to perform the prediction. In contrast, the CSP approach requires the labels of the nodes to be correctly typed (except those that are untyped). Another advantage is that this approach returns a *sorted* list of the candidate types; the top in the list is the most similar, the next one is the second most similar, etc. In the CSP approach, the returned suggested types are not sorted. A trade-off for that, which is the disadvantage of the approach presented in this chapter against the CSP approach, is

that in the the latter, the size of the list containing the candidate types is pruned and in any case is smaller or equal to the number of total types in the metamodel. In the similarity flooding approach the size of the list returned is the similarity of each untyped node with *all* the nodes in the target graph (metamodel); including those that represent attributes and references.

5.3. Graph Configuration

The similarity flooding algorithm is a schema matching algorithm. One of its advantages is that it can be applied to match schemas from different domains, thus not targeting specific matching problems (e.g., similarity between XML schemas, or database schemas, etc.). In order to calculate similarities and support its generic nature, the input of this algorithm is two directed labelled graphs, the source and the target, where elements of the source graph need to be matched with those from the target. In the scenario of an XML schema matching problem, the XML schemas are transformed into graphs. However, the domains where the similarity flooding algorithm can be applied are quite broad. This does not allow its authors to propose specific rules that could be used to transform the source and target artefacts (e.g., XML schemas) to the source and target graphs.

In the domain of MDE, Falleri et al. [11] employ the similarity flooding algorithm to automate the generation of model-to-model transformations. Their goal is to identify matches between two different metamodels, thus the source and target schemas of their matching problem are of the same domain, that of ECore metamodels. As described in Section 2.8.2 they proposed six different set of rules for the transformation of metamodels to graphs. These can be used as input to the similarity flooding algorithm in the domain of MDE. Their names are *minimal, basic, standard, full, flattened and saturated*. The minimal, which is the simplest one, is described in detail in Section 2.8.2. The rules referred to as *configurations* include different amount and type of information each time. Falleri et al. [11] evaluated them on the specific problem, that of automatic model-to-model transformations, and identified that the minimal configuration had the worst performance. The remaining five performed significantly better than the minimal but there was no significant difference among them.

From the description of the proposed approach for type inference given in Section 5.2, one can identify that the matching problem in need of a solution in our approach is between an example model and a metamodel. Thus, our source schema differs from that of Falleri et al.'s as we have an instance model and not a metamodel. A metamodel is a model itself, however, it differs from the example models in at least one point that is of importance for this work. The difference is that in the example models abstract classes are not represented as in instances only the concrete classes can be instantiated. Amongst the six proposed representations in [11],

the *flattened* representation is the only one not taking into account these abstract classes; their features are explicitly inherited to their children while the abstract classes themselves are deleted from the graphs. As the flattened representation omits the identified difference and its performance is amongst the top, we decided to select it as the configuration for the input graphs in our matching problem. The description of the flattened configuration follows.

5.3.1. Flattened Configuration

In our matching problem the source schema is an example model created as part of a flexible MDE approach and the target schema is a metamodel. We describe the flattened configuration through an example, using part of the example model introduced in Section 3.3.1. For easier reference we highlight that part of the example model in Figure 5.2. The flattened configuration for this extract is presented in Figure 5.3. A detailed description of the rules to transform the example models to this configuration follows. These rules are proposed by Falleri et al [11] and they were adapted to fit to the purposes of this chapter's proposed approach (in [11] only metamodels are transformed into directed graphs while in our approach example models also need to be transformed).

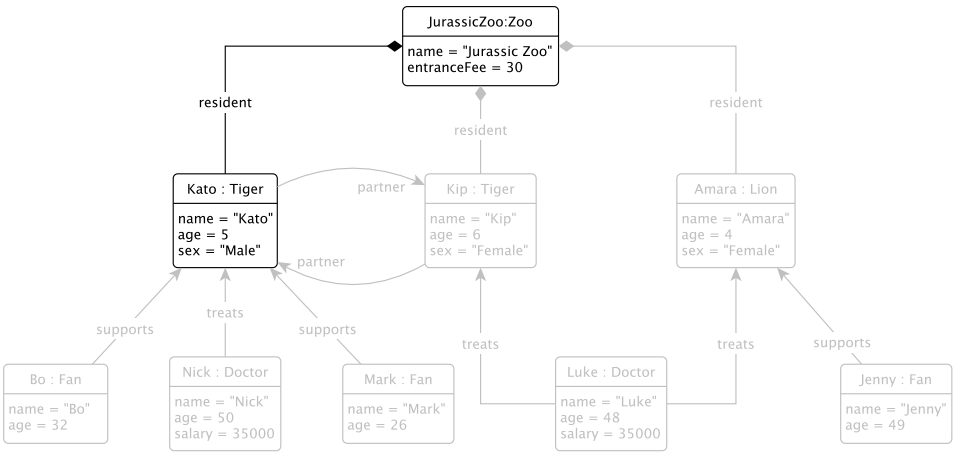


Figure 5.2.: Extract of the example model.

- For each class node in the model a unique identifier node in the graph is created. This id node consists of the "#" character and a number. For example, the class element "JurassicZoo:Zoo" is represented by the graph node typed as "#1". The "#2" is the representation of the "Kato:Tiger" class. The same process happens for every attribute and reference in the model. For example, the attribute named "age" is represented by the node "#5" in the graph, while the reference "resident" by the node "#9". Finally, a hash node is created for all the datatypes in the model (e.g., the "String" datatype is node "#7" in the

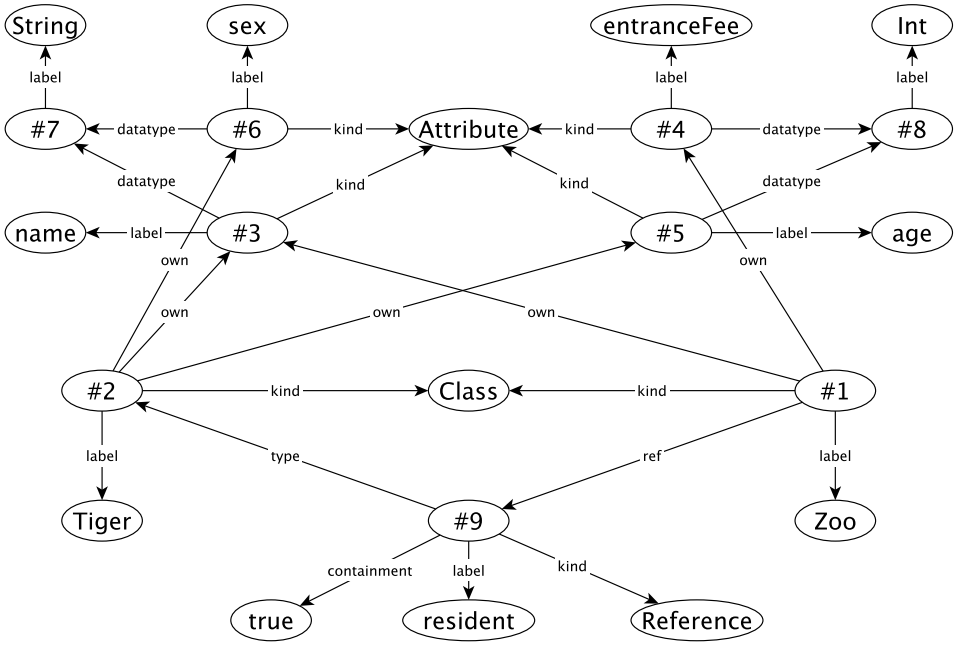


Figure 5.3.: The directed labelled graph of the example model of Figure 5.2 using the flattened configuration.

graph). For the attributes that have already been created (e.g., the "name" attribute is a feature for both the "Zoo" and the "Tiger" class), only one hash node that represents all the occurrences of these attributes is generated.

- For all these hash nodes another node is produced holding the label/name of that node. For example, as soon as the "#1" hash node represents the "Zoo" class, a node named "Zoo" is created in the graph. In the same manner, "age", "resident", "String", etc. nodes are created. An arc named "label" connects the hash nodes with the nodes holding their names.

- Nodes for the kind of each hash node are also created. The different kinds are "Class", "Attribute" and "Reference". Each hash node is connected with the appropriate kind node using an arc labelled "kind". For example, the "#1" node represents a class, thus a "kind" arc is going from it to the "Class" node.

- For nodes that represent attributes a "datatype" arc is created to connect their hash nodes with the hash node of the datatype. For example, the "#5" representing the "age" attribute is connected with the hash node "#8" (i.e., the node that represents the "Int" datatype) with a "datatype" arc. In order to declare that these attributes are features of specific classes, an "own" arc is created that connects the hash node of the class owning that attribute with the hash node of the attribute. For example, node "#1" (i.e., "Zoo") owns an attribute called "entranceFee" (i.e., hash node "#4"), thus an "own" arc connects "#1" node with "#4" node in the graph.

- For nodes that represent references, an arc named "containment" is created and connected to either a node named "true" or "false" to declare if that reference is a containment or not, respectively. For example, reference "residents" (i.e., "#9" in the graph) is a containment in the model, consequently, a "containment" arc connects it with the "true" node. A "ref" arc is also created to declare the ownership of this reference by a class. For example, a "ref" arc is created to connect the "#1" hash node (i.e., "Zoo") with the "#9" (i.e., residents) hash node.

- Finally, a "type" arc is created that connects each reference with its type class. For example, the "type" arc connects the "#9" (i.e., residents) hash node with the "#2" hash node which represents a "Tiger".

The same rules are followed for the transformation of the target schema (i.e., a metamodel). The transformation of the metamodel is carried out using Algorithm 7. A similar algorithm is run for the example model-to-graph transformation.

As described in Section 5.2, the creation of the directed graphs based on the example model and the metamodel is the first step of the automated process (step (3) in Figure 5.1). The next step is the consumption of these graphs by the similarity flooding algorithm to produce the matching results. A step-by-step description of the way the similarity algorithm operates to produce the similarity results is presented in the following section.

5.4. Similarity Flooding

An overview of the similarity flooding algorithm was given in Section 2.8.1. In this section, the three-step process is explained in detail based on a toy example given in [13]. The example with the three steps of the algorithm is shown in Figure 5.4.

In this example, the goal is that of matching two directed labelled graphs, A and B. In directed labelled graphs, each edge is represented as a triple (s, l, t) where s is the source node, l is the label of the edge and t is the target node. For example, the triple for the highlighted nodes and edges in the example Model A in Figure 5.4 would be $(a, l1, a1)$.

Step 1: The first step in the algorithm is the construction of the *pairwise connectivity graph (PCG)*. In a PCG the nodes and edges are calculated based on the following formula:

$$((x, y), l, (x', y')) \in PCG(A, B) \leftrightarrow (x, l, x') \in A \text{ and } (y, l, y') \in B$$

For example, the highlighted in the PCG of Figure 5.4 node-to-node mapping is created for the edge named $l2$ in model A (going from $a1$ to $a2$) and B (going from b to $b2$). The source node in the PCG is named after the names of the source nodes in

Algorithm 7 Transforming a metamodel to a directed graph based on the flattened configuration rules.

1: {Step 1: Create graph and add default arcs and nodes}
2: $Graph \leftarrow labelArc, kindArc, classNode, attributeNode, ...$
3: {Step 2: Create arcs and nodes for classes, attributes and references}
4: $NodesMapping \leftarrow$ map of element's names to their unique # id
5: $N \leftarrow$ set of all classes in metamodel
6: **for all** $n \in N$ **do**
7: **if** n **not** $abstract$ **then**
8: Add label node of n, hash node of n, label and kind arc triples to $Graph$
9: $nodesMapping \leftarrow$ mapping of n's label to n's hash
10: $F \leftarrow$ set of all features (attributes & references) of n and its superclasses
11: **for all** $f \in F$ **do**
12: $owner \leftarrow$ hash node of parent of feature
13: **if** f **not** $\in nodesMapping$ **then**
14: Add label node of f, hash node of f, label arc triple to $Graph$
15: $nodesMapping \leftarrow$ mapping of f's label to f's hash
16: **if** f is of type $attribute$ **then**
17: Add kind, datatype and own (starting from $owner$) arc triples to $Graph$
18: **else if** f is of type $reference$ **then**
19: Add kind and ref (starting form $owner$) arc triples to $Graph$
20: **if** f is $containment$ **then**
21: Add containment arc triple (pointing to trueNode) to $Graph$
22: **else**
23: Add containment arc triple (pointing to falseNode) to $Graph$
24: **end if**
25: **end if**
26: **else if** $f \in nodesMapping$ **then**
27: **if** f is of type $attribute$ **then**
28: $attributeHashNode \leftarrow$node of the already created attribute
29: Add own arc triple (going from $owner$ to $attributeHashNode$) to $Graph$
30: **else if** f is of type $reference$ **then**
31: $referenceHashNode \leftarrow$node of the already created reference
32: Add ref arc triple (going from $owner$ to $referenceHashNode$) to $Graph$
33: **end if**
34: **end if**
35: **end for**
36: **end if**
37: **end for**
38: {Step 3: Create type arcs}
39: **for all** $n \in N$ **do**
40: **if** n **not** $abstract$ **then**
41: $R \leftarrow$ set of all references
42: **for all** $r \in R$ **do**
43: $pointingClassHashNode \leftarrow$ the hash node of the pointing class
44: Add type arc triple (from r to $pointingClassHashNode$) to $Graph$
45: **end for**
46: **end if**
47: **end for**

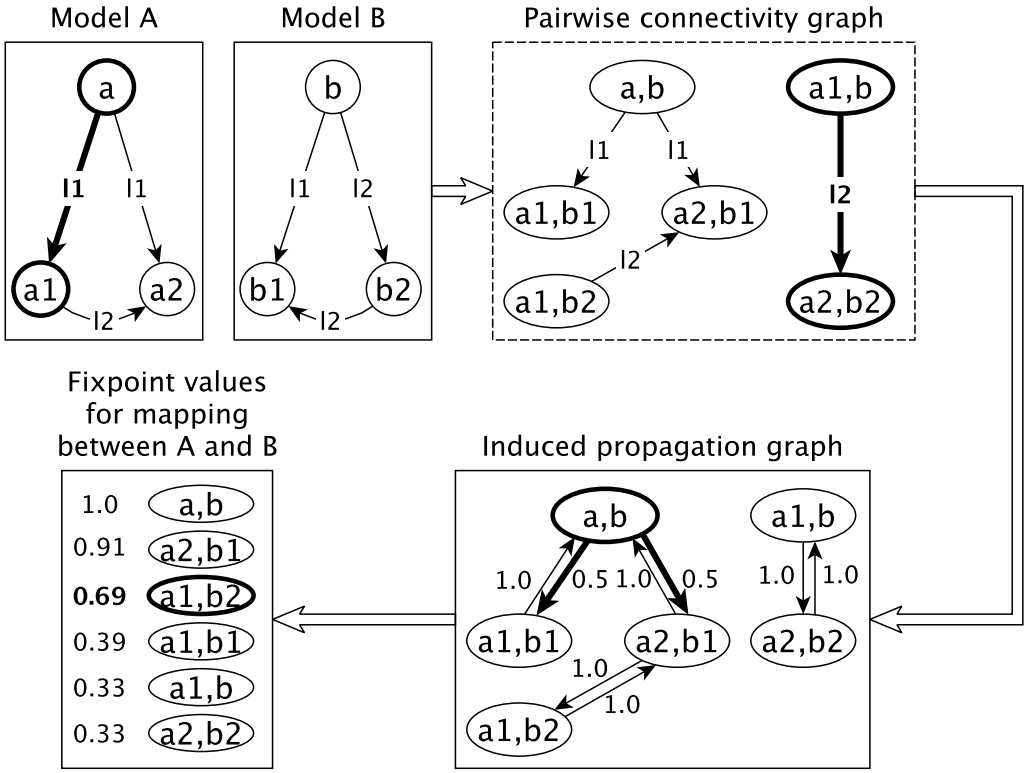


Figure 5.4.: An example of the similarity flooding algorithm's three-step process (based on Figure 3 of [13]).

the models and the target node after the names of the target nodes of the models. The nodes in the PCG are called *map pairs*. This pairwise connectivity graph is the heart of the similarity flooding algorithm as it is based on the intuition that if two nodes are similar, then the nodes they are also connected to with the same labelled arcs are somewhat similar. The nodes that are connected in the PCG are called *neighbours* [13].

Step 2: From the PCG another graph is constructed called a *propagation graph*. For each edge in the PCG a new edge heading to the opposite direction is created and added to the propagation graph. The values of each edge, named *propagation coefficients*, range from 0 to 1 and can be calculated in various ways. In this example, they are calculated by dividing the maximum value (1.0) with the number of edges that are leaving this node and have the same labels. For example, for the highlighted node in the propagation graph shown in Figure 5.4 there are two edges labelled $l1$ exiting the node. So, the propagation coefficient value of each is $1.0/2 = 0.5$. The propagation coefficients are weights that declare how well the similarity of a map pair will be propagated to its neighbouring nodes (map pairs). The authors proposed different ways of computing the propagation coefficients - more than the one presented here - and evaluated them through an experiment to identify the best. In this work we employ the one they propose to be the most efficient. More

details on this can be found in [13].

Step 3: The last step is the calculation of the similarity between all the elements of model A and all the elements in model B $(A \times B)$. This is done by using an iterative computation of the string similarity between the two elements in each map pair. In each iteration this similarity is propagated to the neighbouring map pairs weighted by the propagation coefficient value calculated in the previous step. The iterations stop either after a specific number of iterations or after the computation converges below a specific threshold. After completing all the iterations the final similarity is returned. For example, the best mapping for node $a1$ from Model A is node $b2$ from Model B and the final calculated similarity is 0.69 (highlighted in Figure 5.4). The authors propose four different formulas for accumulating the similarity of each map pair in each iteration. In this work we use the formula that was the most efficient in the evaluation the authors carried out. More details on these formulas are available in [13].

The similarity between two nodes of a map pair is calculated by a string similarity formula based on the Levenshtein metric [116]. The Levenshtein distance between two strings is the minimum number of single character insertions, deletions and substitutions needed in order to make these two strings similar. The formula for the calculation of similarity between two nodes x and y in a map pair n of the PCG is calculated based on the following formula also proposed in [13].

$$s_n = 1 - levenshtein(x, y)/max(len(x), len(y))$$

where $levenshtein$ is the Levenshtein distance between x and y, and $len(x)$, $len(y)$ are the lengths of the labels of nodes x and y, respectively. If at least one of the two nodes x and y is an identifier (i.e., its name includes the "#" symbol), then $s_n = 0$.

The similarity flooding algorithm is implemented by the authors of the approach in the Java [45] programming language. In their implementation the directed graphs are constructed using the Resource Description Framework (RDF) [36,172] Java implementation.

The similarity flooding algorithm returns a list of similarity values for each of the nodes in the source graph with all the nodes in the target graph. The mapping with the highest similarity value for each node is the one with the highest chances to be the type of the untyped element. The experiment run to evaluate the approach is presented in the following section.

5.5. Experimental Evaluation

In this section, the experimentation setup is explained (Section 5.5.1), followed by the results and the discussion on them (Section 5.5.2). To evaluate the proposed approach we will use the *Muddles* [4] flexible MDE approach, however, in principle,

the approach can be applied to any other type of example models. The minimal set of requirements the flexible MDE approach should have is the following:

- provides a mechanism to extract the names of *attributes* and *relationships* in the example models
- provides a mechanism to extract the type of the *attributes* (e.g., String, Integer, etc.) and the type of the *relationships* (i.e., reference or containment) in the example models

The following are the research questions where answers are sought through the experiment conducted in this section:

- **RQ1:** Which proportion of the types of the untyped nodes is predicted correctly by applying the proposed approach?
- **RQ2:** Which is the position of the correct type in the returned list containing the similarities of each node?

The similarity flooding algorithm returns a list containing the similarity value of each node in the example model with each node in the draft metamodel. Regarding **RQ1**, we identify the proportion of the times where the correct type was positioned *first* in the list, thus when the algorithm has successfully inferred the node's type. Regarding **RQ2**, we are interested in finding the position of the correct type in the results' list even if this was not the first suggestion (e.g., the second most similar, etc.). Related to the hypothesis presented in Section 1.2, **RQ1** will help investigate if the approach achieves acceptable accuracy in predicting the correct types of untyped nodes. **RQ2** will reveal if the approach offers, in a reasonable amount of time (no more than a few minutes), an acceptable reduction to the set of candidate types so can feasibly be applied to a flexible MDE approach.

5.5.1. Experiment

In this section we present the experiment used to evaluate the performance of the proposed approach to type inference. An overview of the experiment is shown in Figure 5.5. A detailed description of each step follows.

As with the previous experiments for the two other proposed approaches to type inference presented in Chapters 3 and 4, to evaluate our approach we applied it to a number of randomly generated models. These models are instances of 10 metamodels that are part of a corpus of metamodels presented in [154]. The metamodels were picked randomly with no specific criteria other than that of having a variation in size (number of concrete meta-classes). The reason for using randomly generated models that conform to metamodels and not muddles was largely pragmatic: there is no available corpus of muddles, so the approach could not be evaluated

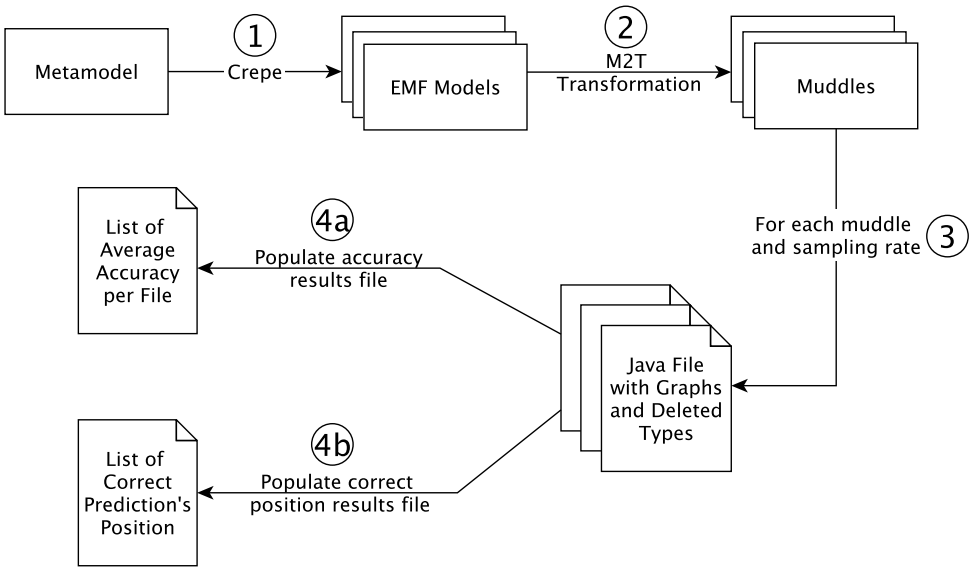


Figure 5.5.: The experimentation process for the type inference approach based on the similarity flooding algorithm.

against a large set of available data. Threats to validity due to this are examined in Section 5.6.

For each of the 10 metamodels, we produced 10 random instances employing the Crepe model generator [155] (step (1) in Figure 5.5). Crepe uses genetic algorithms to produce random models and assigns values for the attributes by randomly picking one from the pool of all the available values given as input. In our approach, the value of the attributes does not affect the performance of the algorithm as they are not taken into account when the example models are transformed to graphs. Thus, the attribute values were populated with random strings.

Step (2) of the experimentation process consists of the transformation of the models into muddle instances. This is accomplished by using the algorithm presented in Listing 3. These first two steps ((1) and (2)) are executed to create the sets of data for the experiment. This process could be avoided if there was a portfolio of example models available constructed as part of a flexible MDE approach. However, to the best of our knowledge, such a portfolio does not exist. At the end of these steps, we have 100 different muddles that conform to 10 metamodels.

The muddles' transformation and their respective metamodel into the directed graphs described in Section 5.3 is completed in step (3). Specifically, for each muddle we firstly transform its metamodel into a directed graph using Algorithm 7. A M2T transformation written in the Epsilon Generation Language (EGL) [65] is used for that purpose. As the available similarity flooding code supports input of direct labelled graphs expressed in the RDF syntax implemented in Java, the generated graphs are expressed in this RDF Java syntax. An extract of the generated Java code for the "Zoo" metamodel presented in Figure 2.3 is given in Listing 5.1

```
1  public static RDFFactory rf = new RDFFactoryImpl();
2  public static NodeFactory nf = rf.getNodeFactory();
3  public static Model MM = rf.createModel();
4
5  // Default arcs
6  Resource labelArc = nf.createResource("label");
7  Resource refArc = nf.createResource("ref");
8  ...
9  Resource kindArc = nf.createResource("kind");
10
11 // Default nodes
12 Resource classNode = nf.createResource("ClassElement");
13 Resource attributeNode = nf.createResource("AttributeElement");
14 ...
15 Resource trueNode = nf.createResource("true");
16 ...
17
18 // Class ''Zoo'' creation
19 Resource node5 = nf.createResource("Zoo");
20 Resource hash5 = nf.createResource("#5");
21 MM.add(nf.createStatement(hash5,labelArc,node5));
22 MM.add(nf.createStatement(hash5,kindArc,classNode));
23 // Attribute ''name'' creation
24 Resource node6 = nf.createResource("name");
25 Resource hash6 = nf.createResource("#6");
26 MM.add(nf.createStatement(hash6,labelArc,node6));
27 MM.add(nf.createStatement(hash6,kindArc,attributeNode));
28 MM.add(nf.createStatement(hash5,ownArc,hash6));
29 MM.add(nf.createStatement(hash6,datatypeArc,stringHashNode));
30 ...
```

Listing 5.1: Java generated code for the construction of the directed labelled graph of the "Zoo" metamodel of Figure 2.3.

In lines 1-2 a new RDF factory is instantiated with a node factory allowing us to create a new graph and nodes to attach to it. This new graph for the metamodel is generated in line 3. In lines 5-9, the arcs representing defaults edges of the flattened configuration like the "label" arc are created. In lines 11-16 the default nodes, like the "class" node are created. In line 18-30 the elements of the metamodel are transformed into directed graph nodes and edges and then added to the tree. More specifically, a node for the label of the "Zoo" class of the metamodel is created in line 19 followed by its appropriate hash node. The arc connecting the hash node with the label node has already been instantiated (in the default arc creation section) and is now added to create the triple in line 21. The same process is followed for the remaining classes, attributes and references.

Then, the RDF Java code for the directed graphs representing the muddle is appended to the already created Java file, below the code generated for the metamodel. An extract of the generated Java code for a muddle, which is an instance of the "Zoo" metamodel presented in Figure 2.3 is given in Listing 5.2

```
1  public static Model MDL = rf.createModel();
2
3  // Default arcs
4  Resource labelArcMdl = nf.createResource("label");
5  Resource kindArcMdl = nf.createResource("kind");
6  ...
7  Resource ownArcMdl = nf.createResource("own");
8
9  // Default nodes
10 Resource referenceNodeMdl = nf.createResource("ReferenceElement");
11 Resource classNodeMdl = nf.createResource("ClassElement");
12 ...
13 Resource intNodeMudl = nf.createResource("Int");
14 ...
15
16 // Reference ''supports'' creation
17 Resource mdlNode7 = nf.createResource("supports");
18 Resource mdlHash7 = nf.createResource("#7");
19 MDL.add(nf.createStatement(mdlHash7,labelArcMdl,mdlNode7));
20 MDL.add(nf.createStatement(mdlHash7,kindArcMdl,referenceNodeMdl));
21 ...
22
23 // Untyped element creation
24 Resource mdlNode15 = nf.createResource("&1");
25 Resource mdlHash15 = nf.createResource("#15");
26 ...
27 MDL.add(nf.createStatement(mdlHash15,ownArcMdl,mdlHash14));
28 MDL.add(nf.createStatement(mdlHash15,refArcMdl,mdlHash4));
29 ...
```

Listing 5.2: Java generated code for the construction of the directed labelled graph of an instance of the "Zoo" metamodel presented in Figure 2.3.

In lines 1-14, the graph, the default arcs and nodes for the muddle are created. In lines 16-29, classes', attributes' and references' nodes and arcs are instantiated and added to the graph (lines 16-21 demonstrate how a reference is instantiated and added to the graph). There are two differences between the creation of the metamodel and the muddle graph. The first difference is that in contrast with the metamodel graph where each type has strictly one node created for it, in the muddle graph there might be several occurrences of the same type. For example, the type "Zoo" is instantiated once in the metamodel graph as it appears once in the

metamodel. However, there might be multiple occurrences of a "Zoo" instance in the example model, thus there are multiple instances of a "Zoo" type node created in the graph. The second difference has to do with the objective of the proposed approach, that of type inference. At this point, due to the fact that the example models were randomly generated from metamodels, all their elements are typed. In order to simulate the scenario of having some nodes left untyped we need to delete types from the example model's nodes. This type deletion occurs during the transformation of the muddle into its directed labelled graph. Looking at line 24 of Listing 5.2 one sees that the label of the node "#15" is "&1". This does not imply, however, that there is a node in the example model for which the type is "&1" but that this is the name used to replace the original type (i.e., "Zoo"); we randomly pick nodes during the transformation and swap their original type with this placeholder to simulate the scenario of missing types.

It is of interest to assess if the sampling rate in the example model affects the performance of the proposed approach. Thus, the type deletion simulation is executed for 7 different sampling rates ranging from 30% to 90% (a step of 10% is applied). A 30% sampling rate means that 30% of the nodes in the muddle have their type declared while for the rest 70% the type has been deleted (swapped with the placeholder "&1"). In order to avoid a lucky (or an unlucky) sampling the random sampling is repeated 10 times for each sampling rate. Thus, for each metamodel the 10 random generated models are sampled 10 times for each of the 7 sampling rates. That sums up to 700 different instances for each metamodel.

In the previously created Java file which contains the graphs we amend the match method of the similarity flooding algorithm (already written in Java). The two created graphs are passed as arguments. The algorithm then returns a list with similarity values for the product of the nodes of both graphs (MDL $\times$ MM where MDL and MM stand for the graph of the muddle and the metamodel respectively). An extract of the returned list is shown in Listing 5.3.

```
[#1,#2: sim=0.23011235267, init=0.0, N=0.2271..., N1=0.2301...]
...
[#1,#4: sim=0.08796859768, init=0.0, N=0.0867..., N1=0.0879...]
...
[#10,#5: sim=0.21893315580, init=0.0, N=0.2167..., N1=0.2189...]
[#10,#17: sim=0.0742164313, init=0.0, N=0.0734..., N1=0.0742...]
[#10,#2: sim=0.07271544227, init=0.0, N=0.0717..., N1=0.0727...]
...
[#10,#15: sim=0.0284573766, init=0.0, N=0.0280..., N1=0.0284...]
...
```

Listing 5.3: An extract of the similarities between pairs of the muddle and the metamodel graphs.

In our matching problem the goal is that of finding similarities from the muddle example model (source) to the metamodel (target). Thus, the first identifier found in the results points to nodes of the muddle graph while the second points to nodes of the metamodel graph. For example, in line 1 of Listing 5.3, the returned result is interpreted as "The node with id "#1" in the muddle graph is similar to the node with id "#2" in the metamodel graph and their similarity value is 0.23". The algorithm then returns the results sorted for each node in the source graph; the first matching pair is that with the highest similarity for each node in the source graph.

Having the results stored in the list we can extract the matching accuracy of the prediction mechanism. This is achieved by comparing the original types (which were stored separately before performing the type deletion) with the best result for each of the untyped nodes. For example, if node's "#1" type was originally "Zoo", we check if the highest similarity pair returned (i.e., "#2" in Listing 5.3) is of the same type ("Zoo"). If this is the case then the correct prediction counter is increased by 1; if not, it remains the same. Next, we assess all the untyped nodes consecutively for each example model. The prediction accuracy score is the number of correct predictions to the total number of this model's untyped nodes. This score is stored in a text file (step (4a) in Figure 5.5). The same process is followed for all the 7,000 generated Java files representing the 700 different example models for each of the 10 metamodels. The accuracy result for each example model is afterwards appended to the results file.

The above metric counts the correct predictions in each example model. As described above a correct prediction occurs when the correct type of the untyped node is returned *first* in the sorted list of results. However, when this is not the case it is of interest to identify the position of the correct type in the returned results. The position of the correct type for each untyped node is subsequently added in a second text file (step (4b) in Figure 5.5). For example, if for an untyped "Tiger" node of the example model, the highest similarity value is with a "Lion" node of the metamodel but the second highest is with the "Tiger" node we store the value 2 in the results file. This way we can identify the proportion of the correct predictions in, for example, the top three guesses for each untyped node.

At the end, two results files are created, one having the success score for each of the example models and the second having the position of the correct prediction for each untyped node. These results are presented in the following section.

5.5.2. Results and Discussion

In this section the results of running the experiments are presented followed by a discussion on the quantitative and the qualitative findings of the experiment. Before that, a brief presentation of the data used as input in the experiment is given in Table 5.1.

Table 5.1.: Input data summary table for the similarity flooding experiment.

Model Name	#Types	Min	Max	Average
Professor	4	42	72	57.3
Chess	5	42	75	58.8
Zoo	6	21	77	36.8
Ant Scripts	6	41	80	63.2
Use Case	7	41	81	53.7
Conference	7	44	81	65.5
Bugzilla	7	42	81	60.6
Cobol	12	24	31	26.9
BibTeX	14	41	80	65.4
Wordpress	19	23	46	36.7

In this experiment we used the same metamodels used in Chapters 3 and 4. The smallest was that defining university professors and the largest was that expressing the structure of Wordpress Content Management System [173] websites. The number of types of each metamodel is shown in the column labelled "#Types" in Table 5.1. These numbers include *concrete* classes only as, firstly, abstract classes cannot be instantiated in the example models, and secondly, the flattened configuration ignores them. In columns labelled "Min" and "Max", the size of the smallest and the largest example model, randomly generated by Crepe [155], is given respectively. Finally, column "Average" shows the average number of nodes for the 10 model instances of each metamodel.

Quantitative Analysis

Table 5.2 summarises the results for the 7,000 runs of the experiment conducted to evaluate the proposed approach. The results are grouped by metamodel and sampling rate. Each value represents the average accuracy between all the 10 runs for the 10 instances of each metamodel (100 runs in total) for the specific sampling rate. For example, the highlighted value **"0.83"** in Table 5.2 denotes that for the Zoo metamodel, when the 70% of the nodes of each model are typed, the average accuracy between the 100 example models is 83%.

Regarding the raw values, the average performance varies from 41% (Conference metamodel) up to 100% (Profesor metamodel). There are metamodels with a bigger number of types which perform better than others with less types and vice versa. Between all the metamodels and the sampling rates there is an average correct prediction of about 67.5%.

It is also of interest to identify if the prediction of this mechanism is affected by the amount of untyped nodes in the diagram (see correlation "Cor. 1" below). In addition, we also assess if the size of the metamodel affects the prediction (see correlation "Cor. 2" below).

Table 5.2.: Results summary table for similarity flooding experiment

		Average Accuracy for Different Sampling Rates								
Model Name	**#Types**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Cor. 1**
Profesor	4	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.000	NA
Chess	5	0.77	0.76	0.76	0.77	0.76	0.77	0.76	0.762	-0.43
Ant	6	0.63	0.62	0.62	0.63	0.63	0.63	0.66	0.630	0.89
Zoo	6	0.80	0.80	0.80	0.81	**0.83**	0.82	0.79	0.808	0.07
Bugzilla	7	0.51	0.52	0.52	0.52	0.53	0.52	0.55	0.524	0.75
Conference	7	0.40	0.41	0.41	0.41	0.41	0.42	0.42	0.412	0.79
Usecase	7	0.70	0.70	0.71	0.70	0.71	0.70	0.71	0.704	0.75
Cobol	12	0.67	0.66	0.68	0.66	0.69	0.69	0.71	0.678	0.71
Bibtex	14	0.62	0.63	0.63	0.63	0.63	0.61	0.65	0.629	0.25
Wordpress	19	0.63	0.63	0.62	0.62	0.65	0.63	0.62	0.628	-0.43
	Avg.	0.67	0.67	0.67	0.67	0.68	0.68	0.69		
	Cor. 2	-0.57	-0.50	-0.59	-0.59	-0.57	-0.65	-0.63		

Cor. 1: How strong is the dependency between the sampling rate and the success score?

Cor. 2: How strong is the dependency between the number of types in a metamodel (size of metamodel) and the success score?

The first question can be answered by observing the raw results presented in Table 5.2. The average prediction is not affected and there are only small fluctuations as a result of the random sampling. The correlation coefficients are given in column labelled "Cor. 1". There are some correlation scores that declare *statistical* significance, however, the changes to the values are small so we cannot claim that they are important. For example, the highest correlation is that of the "Conference" metamodel, however the difference in the accuracy is only 0.02 between the 30% and 90%, the lower and upper sample rates of the experiment, respectively. Concluding, the amount of nodes left untyped in the diagram does not affect the performance of the approach.

That is an expected outcome if one takes into account the way example models are represented as graphs. For example, look at Figure 5.3 which is the graph for the flattened configuration of the extract of the model given in Figure 5.2. The example model consists of 2 nodes which are transformed to 21 nodes in the graph. If one node in the example model is left untyped (50% sampling rate), the single change in the graph would be the change in one label's node (e.g., if the node "Zoo" is left untyped, the node in the graph labelled "Zoo" will change to "&1"). The rest 21 nodes will remain the same, hence, that small change propagated to the rest of the diagram will not have any significant impact. In the experiment conducted here, the example models average from 26.9 to 65.5 nodes (see Table 5.1), consequently, changing the labels of even a large proportion of them is not expected to have any

significant impact.

Regarding the second correlation ("Cor. 2"), the correlation coefficient values for the different sampling rates are given in the row labelled "Cor. 2" (see Table 5.2). The values are nearly identical for all the sampling rates due to the outcome of the previous question: the accuracy is the same between the different sampling rates thus the correlation between similar values and fixed numbers (i.e., the number of types in each metamodel) is similar. When examining the values one cannot reach a definitive answer. The correlations suggest a weak negative similarity. These values may have been affected by specific metamodels (e.g., "Conference") that score relatively low given the number of types they have. Nevertheless, we have no evidence suggesting that the number of types affect the accuracy prediction or any evidence to support that they do not.

The second part of this experiment is related to the position of the correct prediction in the returned sorted list of candidate matches (RQ2). A histogram that summarises these values for all the untyped nodes in all 7,000 experiments is given in Figure 5.6.

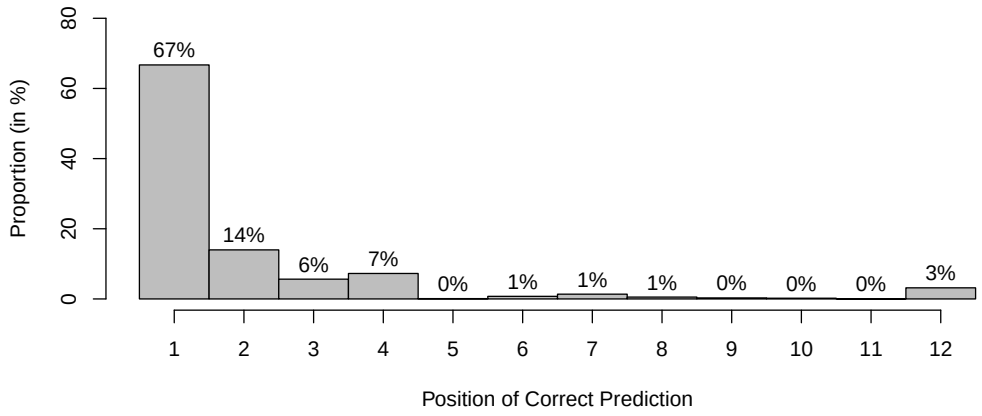
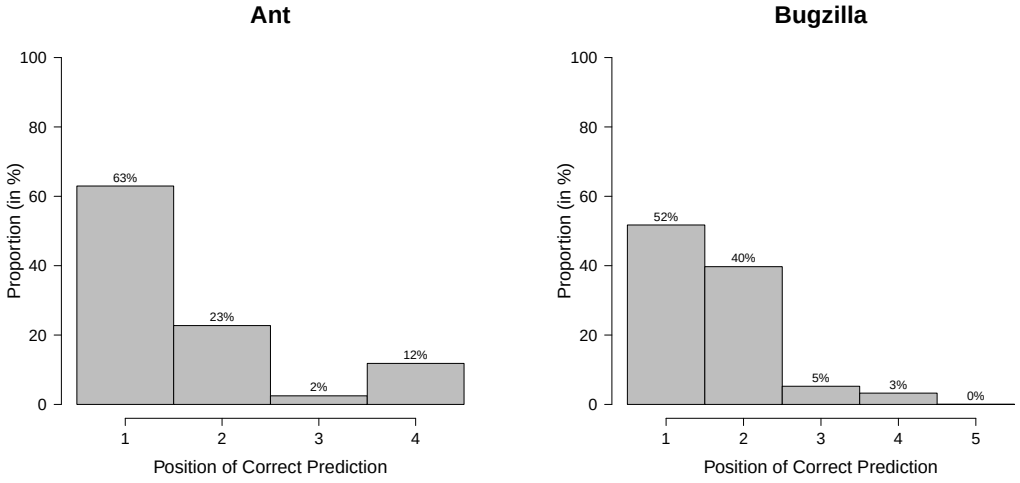


Figure 5.6.: Histogram of the correct prediction's position for each untyped node in the similarity flooding experiment.

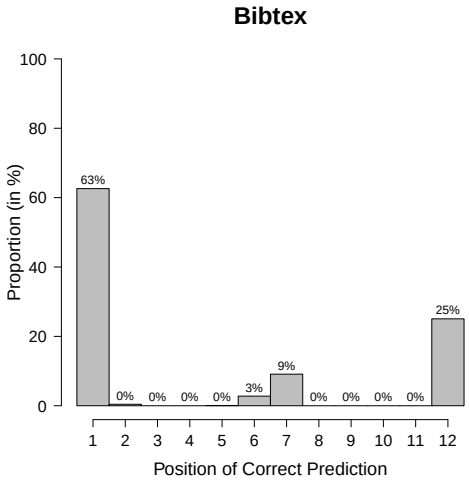
The histogram suggests that from approximately 140.000 nodes which were left untyped in the experiment, the correct prediction for a significant proportion (87%) of them was in the top 3 predictions returned by the similarity flooding algorithm.

In order to extract safe conclusions, the histograms for all the metamodels in the experiment are given in Figure 5.7. These histograms present the position of the correct prediction for each metamodel in the results list for each of the untyped nodes. For the vast majority of the metamodels the accumulative proportion of nodes for which the correct type was in the first 3 predictions is above 75%; for some it is more than 90%, in two of them it is 100%, with an exception, that of Bibtex with a 63% (see Figure 5.7(c)). A discussion on the qualitative findings and their possible explanation is provided in the following qualitative results' section.

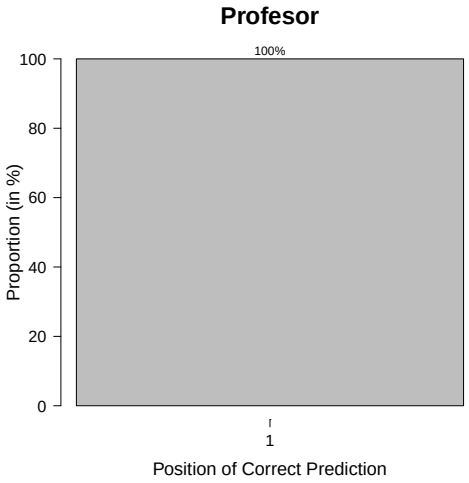


(a) Ant

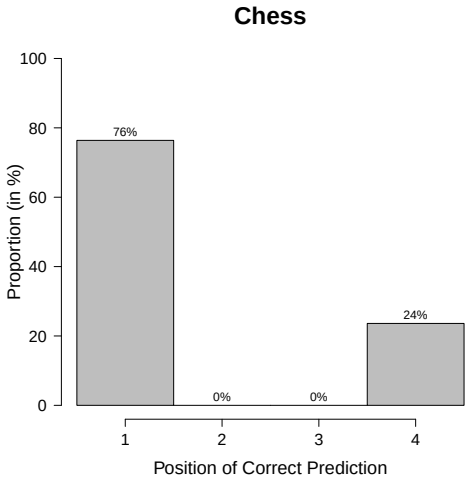
(b) Bugzilla



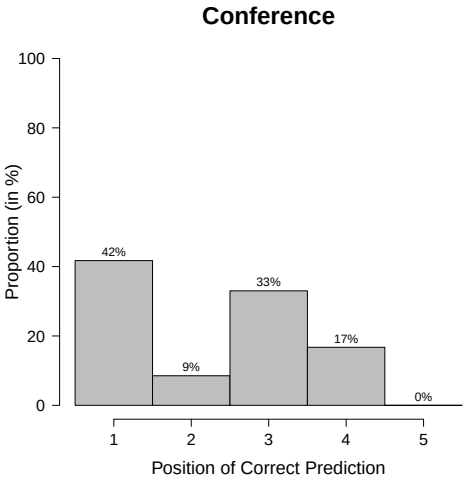
(c) Bibtex



(d) Profesor

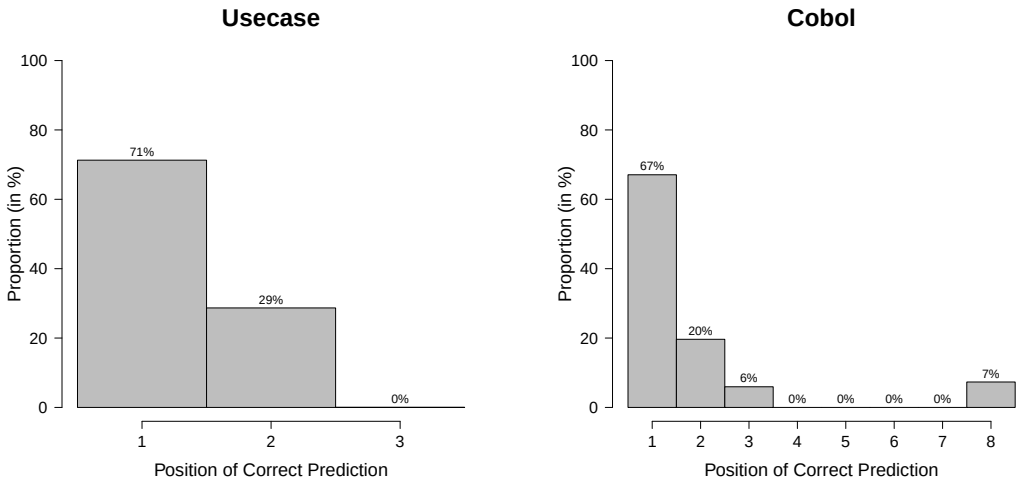


(e) Chess

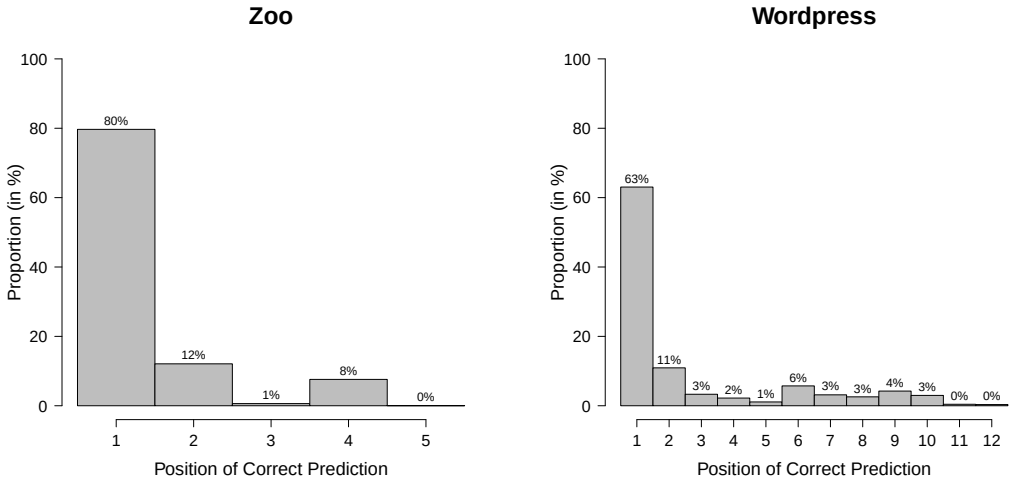


(f) Conference

Figure 5.7.: Position of the correct prediction for each metamodel in the similarity flooding experiment.



(g) Usecase (h) Cobol



(i) Zoo (j) Wordpress

Figure 5.7.: Position of the correct prediction for each metamodel in the similarity flooding experiment (continued).

The results answer both the research questions **(RQ1 & RQ2)** set at the beginning of this experiment. Regarding RQ1, the approach managed to successfully predict the type of a large proportion of untyped nodes. Regarding RQ2, we identified that for the majority of the metamodels, the correct type is included in the top three predictions of the algorithm. This suggests a significant reduction in the set of possible types the language engineer needs to check, especially when the total number of types in the metamodel is increased. The similarity flooding algorithm needs a few milliseconds to produce the similarity matrix and as a result it can be feasibly included into the workflow of a flexible MDE approach as mentioned in the hypothesis of this project.

Qualitative Analysis

From the analysis of the raw values presented in Table 5.2, one can observe that for the majority of the metamodels the proposed approach's performance is good. However, an investigation on where the similarity matching algorithm fails and possible reasons for that failure can be useful.

By manually assessing different models for each metamodel of the experiment we identified that the algorithm had a tendency to make wrong predictions between elements of types that extend the same class. Specifically, in the majority of the metamodels, when two classes were extending the same superclass and one of them had one (or more) extra differentiating features (e.g., an extra attribute), the algorithm showed the tendency to pick as the most similar type the one having the extra feature. For example, in the Bibtex metamodel, the algorithm was falsely predicting nodes of the type "Manual" as ones of the type "Unpublished". The difference between these two is that the latter had an extra attribute. This was also identified in the Bugzilla metamodel and is the reason why the approach underperforms. This behaviour is explained by the nature of the similarity flooding algorithm. The applied algorithm does not include any form of penalty (i.e., negative similarity) for having extra neighbouring nodes in the target graph's node which are not similar with the source graph's assessed node. In contrast, if the target node has some extra features, even if those are not available in the source node, the similarity is propagated to the node having these extra features. Hence, the algorithm chooses it as the best match. A possible solution for such cases could be the inclusion of the "top 3" predictions in the results and not only the first one. This way, the described phenomenon would be minimised.

A second repeated behaviour across the majority of the metamodels was that of having wrong predictions between some nodes and the parent node that is containing all the other nodes. This parent node is also referred to as "root node" and is a restriction of the underlying modelling technology which is used as part of this experiment. Specifically, EMF by default requires the creation of a "Root" class in the metamodels in order to be able to create instances of all the other types. In this root class, containment references are created to point to all the classes, except those already included as containments in other classes. Thus in some cases, the propagated similarity with the root node was higher than the similarity with the correct type resulting to a wrong prediction. This phenomenon occurred predominantly in the Conference metamodel. A possible solution to this problem could be the application of some constraints on the list of the returning types; an approach suggested by the authors of similarity flooding in [13]. As soon as it is known that in the specific domain the type "Root" is not a potential candidate for any of the untyped nodes, this prediction could be discarded from the final results' list. More details and directions for future work on applying constraints are described in Section 6.2.

Finally, there were cases where specific types were wrongly predicted as other types and the cause for that could not be traced back. Due to their nature both the similarity flooding algorithm and the seven proposed configurations for the graph representations in the domain of MDE proposed in [11] result to large size source and target graphs. Even the smallest viable example of a model with only two elements presented in Figure 5.2 led to the creation of a 22-nodes graph (see Figure 5.3). These complex, large graphs along with the fact that the similarity flooding algorithm performs a number of iterations to propagate the similarities to neighbouring nodes, makes errors' tracing and their potential causes a difficult procedure.

Performance Analysis

Table 5.3 summarises the average execution time for the similarity flooding algorithm used in this approach. The specification of the machine used to run the experiments is the following:

- Architecture: x64 (64-bits)
- Processor: Intel(R) Core(TM) i5-4288U CPU @ 2.60GHz
- RAM: 2x8GB DDR3 @ 1600 MHz
- Hard Disk Drive: 256GB PCIe SSD
- Operating System: Mac OS X 10.11.6

Table 5.3.: Average execution time for each metamodel in the similarity flooding approach.

Model Name	#Types	Average Execution Time for Each Metamodel (in seconds)
Profesor	4	0.38
Chess	5	0.33
Ant	6	0.47
Zoo	6	0.38
Bugzilla	7	0.53
Conference	7	0.39
Usecase	7	0.35
Cobol	12	0.47
Bibtex	14	0.41
Wordpress	19	0.67

As one can see from Table 4.5, the average execution time varies from 350 milliseconds up to 670 milliseconds. The average execution time is not affected by the size of the metamodel, as metamodels (i.e., Cobol) have the same average execution time with others (i.e., Ant) although they are double in size. This is probably due

to the fact that the example models for the "Cobol" metamodel were smaller than the "Ant" (see Table 5.1). In any case, the execution times are relatively small and consistent and thus scalability cannot be consider as an issue for this approach.

Comparing to the approaches presented in Chapters 3 and 4, this approach is slower (not significantly, though) than the approach based on classification algorithms and faster that the approach based on constraint programming.

5.6. Limitations

As described in Section 5.5.1, the data used to evaluate the proposed approach are created using a random model generator. A first issue is that the generator does not instantiate muddles directly, but models which are transformed into muddles. This happened as currently there is neither a random muddle generator available nor a large portfolio of muddles that could be used to evaluate the approach.

The flattened configuration we used in the approach takes into account two categories of features: attributes and references. Regarding the latter category, we do not believe that using muddles which have been transformed from models has any significant impact it the performance of the proposed approach. The majority of the references in the metamodels are optional meaning that they are not always instantiated for each element in the model, thus the necessary randomness is achieved.

Regarding the second type of features, that of attributes, we believe that it has an impact on the performance of the algorithm. Although Crepe generates random models, the number and names of attributes that each node has is always the same for nodes of the same type. As described in Section 5.3, each attribute is translated to a number of nodes in the flattened configuration graphs. The equivalent nodes between the source and the target graphs are compared and their similarity is propagated to neighbouring nodes. Having some of the attributes missing would lead to less "amount of similarity" propagated, resulting in the final similarity value being different. A small scale experiment ran and verified our expectation. In Section 6.2, plans are discussed for a full scale experiment where different levels of noise will be added to the attributes to identify the impact of this treatment to the performance of the proposed approach.

A significant advantage of this approach when compared to the one proposed in Chapter 4 is that it is less sensitive to typing errors (typos). In the type inference approach based on constraint satisfaction (see Chapter 4) any typing error in the example model's references would not allow the constraint satisfaction problem to have a solution until this error was fixed. In contrast, in this approach the algorithm will produce results for all the nodes; the typo will only affect the node it is related to. The significance of the impact for the type inference of this node is related to the size of the typing error. Nonetheless, due to the architecture of the flattened configuration (the label of the reference is one node in the graph out of the six

nodes and arcs related to a reference) this impact is reduced.

The results suggest that although the algorithm performs well in predicting the types of the nodes, the performance is affected by the domain (metamodel) it is applied to. This outcome was identified in the evaluation experiments in [13] and [11]. Thus, a fully automatic type inference is not suggested unless the engineers have previously tested the approach in the specific domain the example models describe and verified that it performs well. We believe that a semi-automatic approach like the one used in Flexisketch (see Section 2.4) would be more appropriate. That is to suggest the top three types for each untyped node that the algorithm returns and let the engineers pick the correct one. As shown in the histograms with the position of the correct prediction, the algorithms' performance is increased significantly when following such an approach.

Finally, in the experiment 10 metamodels were used in total from which a number of muddles was generated. The metamodels were picked randomly from a zoo of 500 metamodels. The number of types in these varied from 4 up to 19 as shown in Table 3.2. It would be of interest to experiment with even larger metamodels, although our experience from working on muddles suggests that having a flexible model with more than 20 different types is a marginally realistic scenario.

5.7. Chapter Summary

In this chapter, the use of a state-of-the-art graph matching algorithm called *Similarity Flooding* [13] to help with type inference in flexible MDE approaches is proposed. We used the flattened configuration deemed as appropriate for representing the source and target schemas of the matching problem (muddles and metamodels respectively). This was proposed in [11] and its performance was ranked among the best.

This approach can be used in scenarios where a draft metamodel has already been inferred. When applying the approach to the example models that contain untyped nodes, the language engineers and the domain experts receive a sorted list of similarities between each untyped node and the types available in the metamodel. The most similar types are listed first facilitating the selection of the correct type.

In order to test the proposed approach we ran experiments on a large number of example models. The results suggest that in the majority of the metamodels the prediction accuracy was high. This accuracy was increased when the top 3 results for each node were taken into account. Based on that outcome, we suggested the use of the proposed approach in a semi-automatic manner to facilitate the transition from partially typed example models to more complete ones.

CHAPTER 6

Conclusions

This thesis proposes three solutions for type inference for nodes that are left untyped in example models. These example models are created as a result of following a flexible MDE approach. MDE has proved to offer benefits like better product quality, increased productivity, maintainability and portability [2, 80–83] in software engineering. Following MDE processes implies expertise in metamodelling, and in relevant technologies. While this may be an understandable process for MDE experts, this is not always the case with domain experts [5, 22]. However, the involvement of domain experts is important in the definition of high quality and well-defined DSLs that cover all the needed aspects of a domain [5,24,25,30]. To address the aforementioned issue, flexible modelling approaches have been proposed in the literature (e.g., [4–6, 31]). Such approaches are based on sketching tools for the definition of example models that describe the envisioned DSL. A definition of a metamodel during the initial phases of language engineering is not required; draft versions of it can be inferred either manually or automatically when a good understanding of the domain is achieved.

However, a trade-off for the above advantages of flexible MDE, is that bottom-up MDE approaches are prone to various types of errors. Sketching tools cannot offer semantic and syntactic correctness rules checking that rigorous MDE tools offer. Among others, a common error is that of type omissions: nodes that are sketched in the example models are left untyped for various reasons. Having nodes that are left untyped hardens the inference of metamodels based on the example models as critical information (i.e., types of concepts) is missing. The research in this thesis addressed this gap and has explored the following research hypothesis as this was stated in Section 1.2:

> *It is feasible to use classification algorithms, constraint programming and graph similarity techniques in the two phases of flexible MDE approaches to accurately suggest the most appropriate type for each untyped node of a model or reduce the set of possible types an untyped node can have. This can reduce the effort needed to produce complete example models from which metamodels can be inferred.*

Based on the above, the research objectives of this thesis, as identified in Section 1.2 are to:

- Facilitate the incremental acquisition of domain knowledge that flexible MDE approaches offer by inferring the type of the untyped nodes in example models.
- Identify existing research techniques that can be used for type inference in flexible MDE.
- Propose new algorithms that could be used in the same direction.
- Develop the artefacts to import the aforementioned techniques and algorithms in the domain of type inference in flexible MDE.
- Evaluate the performance of the proposed approaches and the proportion of assistance they offer to language engineers.

6.1. Thesis Contributions

In this section the contributions of this research work are presented.

Type Inference using Classification Algorithms

In Chapter 3, type inference approaches based on classification algorithms were proposed. More specifically, Classification and Regressions Trees (CART) and Random Forests (RF) were used to infer the types of untyped nodes. The classification is performed by analysing features of the elements in the example models without requiring the existence of any draft metamodel, which is the case early in the flexible MDE process. Two set of features were proposed, one based on semantics of the example models and a second based on the concrete syntax.

Regarding the former, the example models are parsed to extract the following five characteristics of interest as presented in Section 3.3.1:

- *Number of Attributes:* The number of attributes that the node has.
- *Number of different types of incoming references:* The number of all the types of references that target that node.

- *Number of different types of outgoing references:* The number of all the types of references that come from that node.
- *Number of different types of children:* The number of all the unique types that the node contains.
- *Number of different types of parents:* The number of all the types that the node is contained in.

The second set contains features related to the concrete syntax of the example models. The four features are listed bellow:

- *Shape:* The shape of the node.
- *Color:* The color of the filling of the node.
- *Width:* The width of node.
- *Height:* The height of the node.

Using one of the proposed set of features and one of the proposed classification algorithms the approach returns one suggested type for each untyped node based on knowledge gained from training on the nodes that are already typed in the example model. This suggested type is not guaranteed to be the correct one, and as a results language engineers need to approve the suggestions for all the nodes.

The proposed approach was evaluated on a number of random generated models from 10 different domains (metmodels). Results suggested that the approach had a good prediction performance for both feature sets proposed. In addition, the classification algorithm used did not affect the performance. In contrast, the size of the metamodel and the proportion of the nodes that are left untyped are affecting the prediction capabilities of the approach.

Type Inference using Constraint Programming Principles

In Chapter 4, a type inference approach based on Constraint Programming principles is proposed. This approach, in contrast to the one presented in Chapter 3 requires the existence of a manually or automatically inferred draft metamodel to perform type inference, thus it is suitable for later steps of flexible MDE processes. The approach parses this draft metamodel to extract rules and facts that it includes and construct the Constraint Satisfaction Problem (CSP). The example models are also parsed and included in the CSP. Using solvers, like ECL^{i}PS^e [14, 110], a set of possible types are returned for each of the untyped nodes.

In contrast to the approach presented in Chapter 3 that returns only one suggestion for each untyped node, this approach returns a set of suggested nodes. The trade-off is that the correct type is guaranteed to be in the set of the suggested

types returned. As a result, in case of nodes where only one type is returned, the approach can automatically apply the suggested type to the node.

The proposed approach was evaluated on a number of randomly generated models, instances of the 10 different metamodels used in the evaluation of the approach proposed in Chapter 3. A metric that calculates the savings in terms of effort that language engineers have to do in order to manually assign the type for each untyped node was proposed. The results suggested that the average savings percentage was up to 80.87%. As the rules in the CSP are imposed by the relationships between the different types in the metamodel and their multiplicities, the proposed approach was also applied in random models for which the isolated nodes (those that were not connected with any other) were ignored. As expected, a significant improvement in the average saving percentage was occurred in that scenario.

Type Inference using Graph Similarity

In Chapter 5, a type inference approach based on a widely used graph similarity algorithm, called Similarity Flooding [13, 152] is proposed. This approach also requires the existence of a manually or automatically inferred draft metamodel that depicts the elements appearing the example models. Both the draft metamodel and the example models are transformed to directed labelled graphs using a specific configuration proposed in [11]. Each node of the example model graph (source graph) is compared with every node of the draft metamodel graph (target graph) using the Levenshtein [116] string similarity metric. The similarities between nodes from the source and the target graphs are propagated to neighbouring nodes, based on the assumption that two nodes of two distinctive graphs are similar when their neighbouring nodes are similar. The proposed approach returns the similarity value of each node n in the source graph (example model) with each node m in the target graph (draft metamodel).

The approach is evaluated on randomly generated models, instances of the 10 different metamodels used in the evaluation of the two previous approaches proposed in Chapters 3 and 4. The approach managed to correctly predict a good proportion of the types of the untyped nodes. The success ratio varyied between 41% to 100% on average for the 10 different domains. There was no evidence found to support that the prediction capabilities of the algorithm is affected by the size of the metamodel or the proportion of untyped nodes in the example model.

Comparison of the Approaches and Usage Scenarios

In order to be able to identify usage scenarios for each of the approaches presented in this thesis it is important to highlight the benefits, the weaknesses and the differences among them.

One of the differences is that the approaches presented in Chapters 4 and 5 which

are based on constraint programming and string similarity, respectively, require the existence of a draft metamodel while the approach presented in Chapter 3 does not. From this difference one can understand that in early stages of a DSL development where a draft metamodel does not exist, then the only approach that can be used is the one based on classification algorithms (Chapter 3).

A second important difference is that the approach based on constraint programming guarantees that the correct type is always included in the returned set of correct types. That means that whenever the set of the suggested types contains only one type, this could be assigned automatically to the node without the need of having the engineer's interference. This is not the case with the other two approaches. This benefit of the CSP approach makes it preferable in scenarios where a draft metamodel exists however, it comes with a trade-off: This approach has higher execution time than the other two as presented in Section 4.4.2 while sometimes it is not possible to have it finished in a reasonable amount of time. As a result, a suggested scenario would be to use the approach based on constraint programming if waiting time is not a restriction (in the worst case scenario on average 17.03 seconds to finish execution). If results are needed immediately then this approach should be avoided.

The approach based on classification algorithms proved to perform better (in terms of accuracy) than the other two on example model with fewer possible types. In contrast the accuracy performance of the rest two approaches is not affected by the number of types in the metamodel. As a result, the classification algorithms approach should be preferred in scenarios with example models which contain few types (a threshold of 6 to 7 types seems to be a turning point as the experiments suggested). If the example models contain more types, then the other 2 approaches should be used.

In addition, the approach presented in Chapter 3 relies on the names used for the references in the example models. If engineers have reasons to believe that the references' **names** are inconsistent or missing then this approach should be avoided. The other two approaches that are not relying (CSP approach) or are more resilient to references' inconsistencies (Similarity Flooding approach) should be preferred instead.

Finally, in the scenarios where there is doubt about the structure of the example models, i.e, they contain references that are incorrectly placed (e.g., a reference links two elements that should be linked) then the approach based on CSP is more applicable as it can initially be used as it can expose some of such inconsistencies. When these doubts are raised any other approach could be used based on the aforementioned criteria.

6.2. Future Work

In this section, suggestions for future work and potential extensions of the proposed approaches are discussed.

Classification Algorithms

There are interesting directions for future work. The features proposed in this work can firstly be combined and probably improve the accuracy and secondly expanded by more features belonging to each category. For example, the font size, orientation and border thickness of the elements are reasonable additions to the concrete features.

For the evaluation of the proposed approach we have introduced noise to the four out of five semantic-related features. Directions for future work include the injection of noise to the last feature, that of *number of attributes*. More specifically, a post-generation script could be run on the example models and remove attributes from nodes. This way the scenario where language engineers have made errors in the definition of attributes in nodes would be simulated. The expectation is that there would be a decrease in the performance of the algorithm; its magnitude cannot be predicted though.

In the approach we have applied two of the most used classification algorithms, that of Classification and Regressions Trees (CART) and Random Forests (RF). Algorithms, like Support Vector Machines (SVMs) and Artificial Neural Networks (ANNs), described in Section 2.6, might be more efficient in this type of problem and should be checked as well. However, representing the problem in a way that these algorithms expect as input is more challenging and requires a deep understanding of both the classification algorithm and the domain of the problem [135].

Finally, the approach is intended to be used to support flexible modelling, where examples can be created in ways that are not restricted by metamodels. However, it could also be applied directly to traditional MDE, for instance, to infer types for an already-typed model, which may potentially reveal poor or incorrect type assignments or misuses of the metamodel. This can be done by adjusting the value of a specific parameter available in classification algorithm, that of *bucket*. This parameter, denotes the number of different classes (types) could be enclosed in a lead node of the tree. In our experiment this is set to 1, however increasing that will help grouping types that have similarities and might be considered to be extending the same superclass.

Constraint Programming

In the approach proposed in Chapter 4 there are two types of constraints checked and applied to the example models both of which are related with the references.

However, as described in Section 2.1.1, in modelling architectures like the Meta Object Facility (MOF) [3], constraints can be defined using the *Object Constraint Language (OCL)* [60]. Thus, in addition to the constraints imposed by the metamodel and already included in our approach, the OCL constraints could also be transformed and included in the CSP to further prune the possible types. The transformation of first-order logic constraints (OCL) to positive conditional equations that build the CSP can be done by the same way it is done in [145]. OCL invariants are parsed as an *abstract syntax tree (AST)* and depending on nature of the node (if it is a leaf, internal or root) a specific CSP predicate, compatible with the ECL^{i}PS^e solver used in our approach, in generated. In [145] such an approach is followed for the verification of EMF models.

The approach based on Constraint Programming principles returns a set of possible types. However, the returned types are not sorted and are equally likely to be the correct one. In order to improve the assistance that the approach offers to language engineers the types could be sorted. This could be done using different criteria. A simple string matching algorithm (even a more complex one like the similarity flooding [13] proposed in Chapter 5) could be used to check for similarities between the labels of the features (attributes and references) of the untyped nodes and the features of all the suggested types. The ordering will be based on these similarity values.

Similarity Flooding

In Chapter 5, in order to apply the similarity flooding algorithm to our schema matching problem, the default values for the algorithm, which has been proven to perform the best [13,152] were used. However, the authors of the algorithm [13,152] have proposed three more variations of the fixpoint formula and seven more ways to calculate the propagation coefficient. It would be of interest to experiment with combinations of these variations and check if they perform better in our domain. In the study most relevant to our work, Falleri et al. [11] use the same two default values that we use.

In addition, in the proposed approach the *flattened* configuration is used for the representation of the labelled directed graphs of the example models and the draft metamodel. That configuration was one of the 6 proposed in [11] and performed among the best. Although the rest of the configurations (except the *minimal* that has been proven to underperform) include *abstract* classes in their representations it would be of interest to check their performance by discarding the abstract classes' nodes and inheriting their features in the concrete classes that extend them (if any).

In our approach the similarity results between *all* the nodes of the source graph with *all* the nodes in the target graph are returned and taken into account on finding the best match. Melnik et al. [13, 152] propose the use of constraints in the process-

ing of the results: domain related information could be used to prune undesired matches returned from the target graph. In the domain of type inference, we are only interested in finding matches with nodes that represent types. For example, if the algorithm, for a reason, returns that the best match for the type of the untyped node in the example model is the node that represents the name of an attribute in the graph of the metamodel, this suggestion should be discarded.

Another filter that could improve the performance of the approach could be that of *selection metrics* as described in [13,152]. Picking the results with the highest similarity value is not always the best matching tactic as this may lead a lower cumulative similarity in the problem. Matching strategies, like the *perfectionist egalitarian polygamy* [152], could be applied to check if they affect the performance of the algorithm. Six matching strategies were proposed in [13, 152] and could be applied to our problem in the future.

As discussed in Section 5.6, the randomly generated example models include the same attributes that their relevant class in the metamodel owns. It would be of interest to inject noise to the attributes of each node in the example model to check how this affect the performance. At least two types of noise are possible: firstly, randomly delete attributes from the nodes and secondly insert typing mistakes to a proportion of the attributes. An initial small scale investigation was carried out in this direction and as expected the noise injection affected the performance but not critically.

Finally, Grammel et al.'s [12] model matching approach discussed in Section 2.8.2 could be used to identify if their similarity metric and proposed model representation will perform better than the similarity flooding [13] algorithm used here.

Other Directions of Future Work

The proposed solutions are not necessarily individual type inference approaches but could be combined to increase their performance. A useful combination could be that of the approaches based on constraint satisfaction and graph similarity: the results of the first could be sorted by using the results of the second. In addition, as soon as the CSP approach returns a set of types in which the correct type is contained, it could be used as a constraint to prune the results of the approach based on the similarity flooding algorithm: if a high similarity value is returned for a type that is not contained in the results of the CSP approach, then it is discarded as this type is not a plausible assignment.

In this research we focused on inferring the types of nodes that were left untyped in processes that are based in flexible MDE approaches (identified as error #4 in Section 1.1.1). Suggestions for future work include tackling the remaining three types of errors (i.e., *user input errors, inconsistencies due to collaboration, changes due to evolution*). The approaches (modified versions of them) have the potential

to be applied in tackling these problems as well. For example, the graph similarity approach (Chapter 5) could be directly applied to tackle problems related with user input errors. In principle the performance should not be significantly affected as typos will have a minimal effect in the propagated similarity values. Regarding the error related with inconsistencies due to collaboration, a solution based on finding matches based on synonyms in dictionaries could be deployed to tackle the problem. WordNet [174] offers the technical infrastructure for that. Finally, regarding changes to types due to evolution, techniques proposed in the domain of metamodel evolution could possibly be applied to identify evolved concepts in example models.

6.3. Closing Remarks

Model-driven Engineering is deemed to offer increased productivity and product quality in software engineering. The participation of domain experts in the definition of metamodels of good quality and high completeness is important. Flexible MDE approaches promise to fill the gap by promoting the domain experts' involvement in MDE processes. However, at this immature state these approaches are more error-prone than the rigorous, traditional MDE ones. The work presented in this thesis provides engineers with a set of tools that can be used to reduce the effort of applying flexible MDE approaches, bridging the transition from flexible to more rigorous MDE approaches when engineers acquire the necessary knowledge in the domain.

Appendices

APPENDIX A

Metamodels

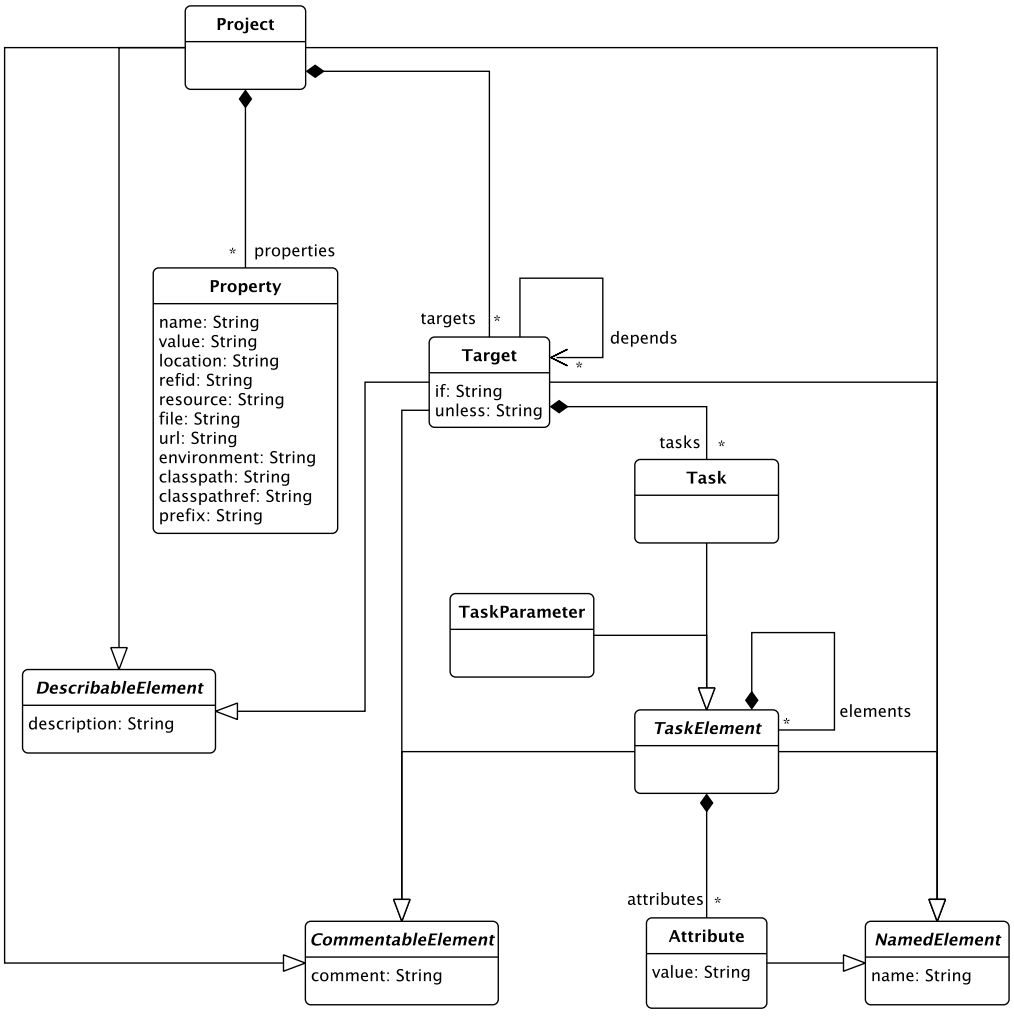


Figure A.1.: The Ant metamodel.

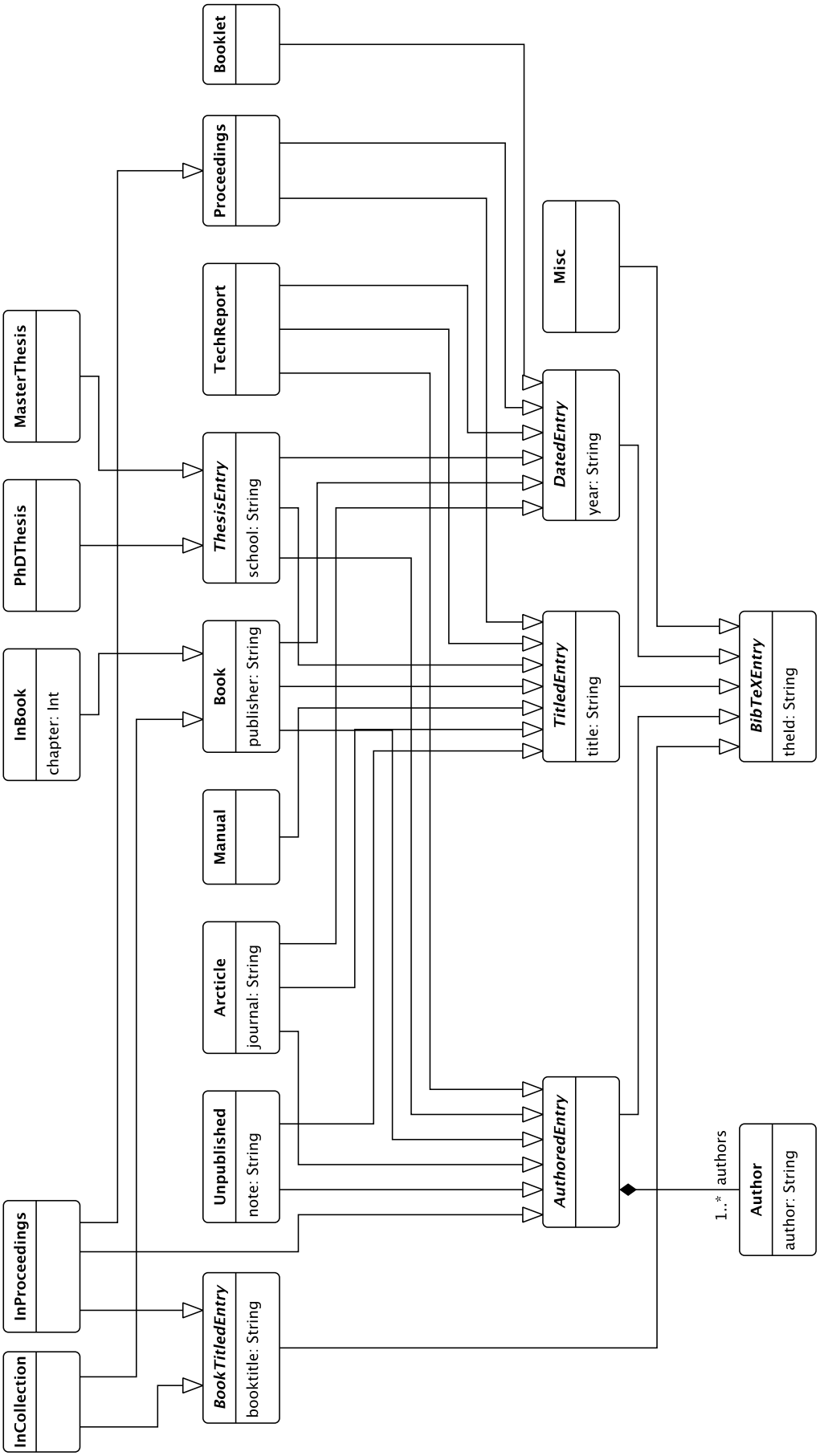


Figure A.2.: The BibTeX metamodel.

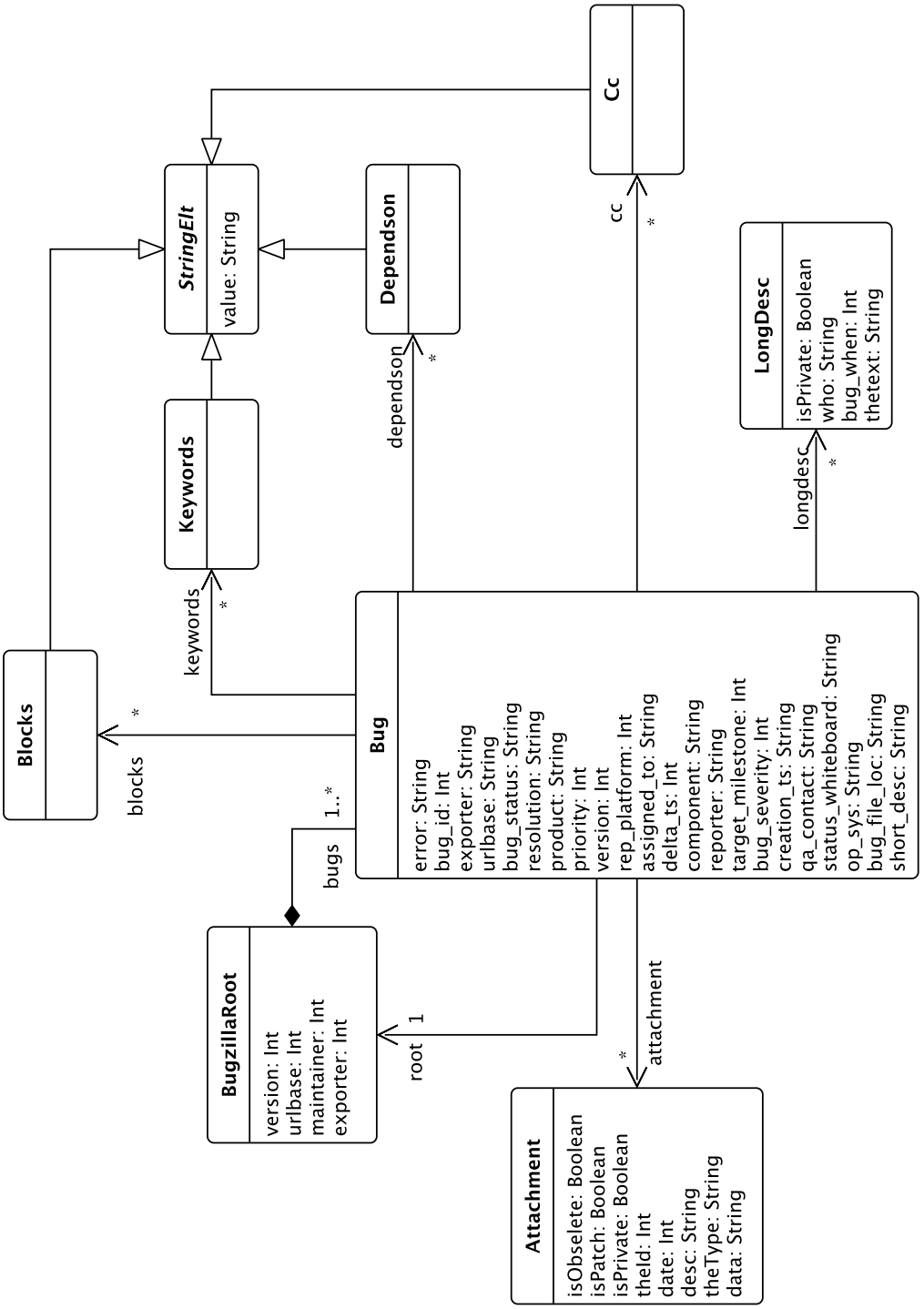
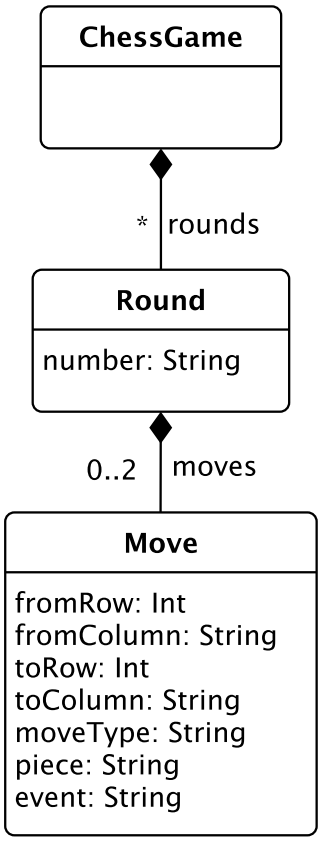
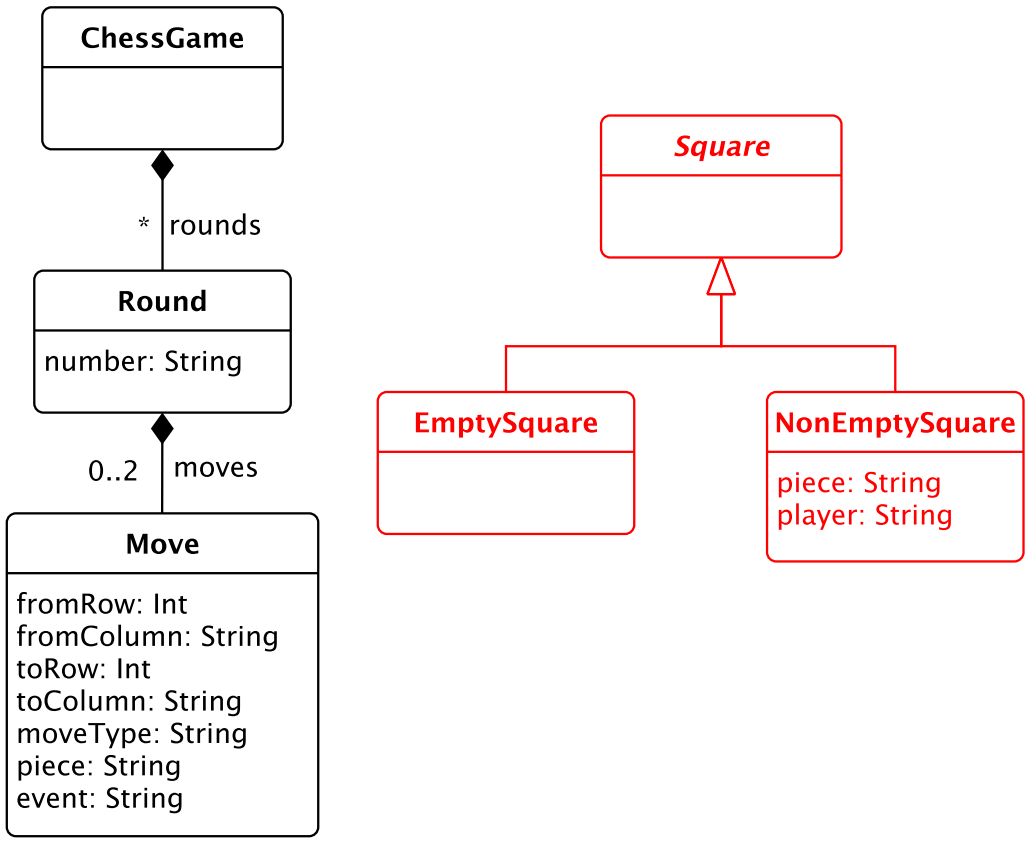


Figure A.3.: The Bugzilla metamodel.



(a) Chess metamodel used in classification algorithms experiments.



(b) Chess metamodel used in CSP and Similarity Flooding experiments.

Figure A.4.: The Chess metamodel.

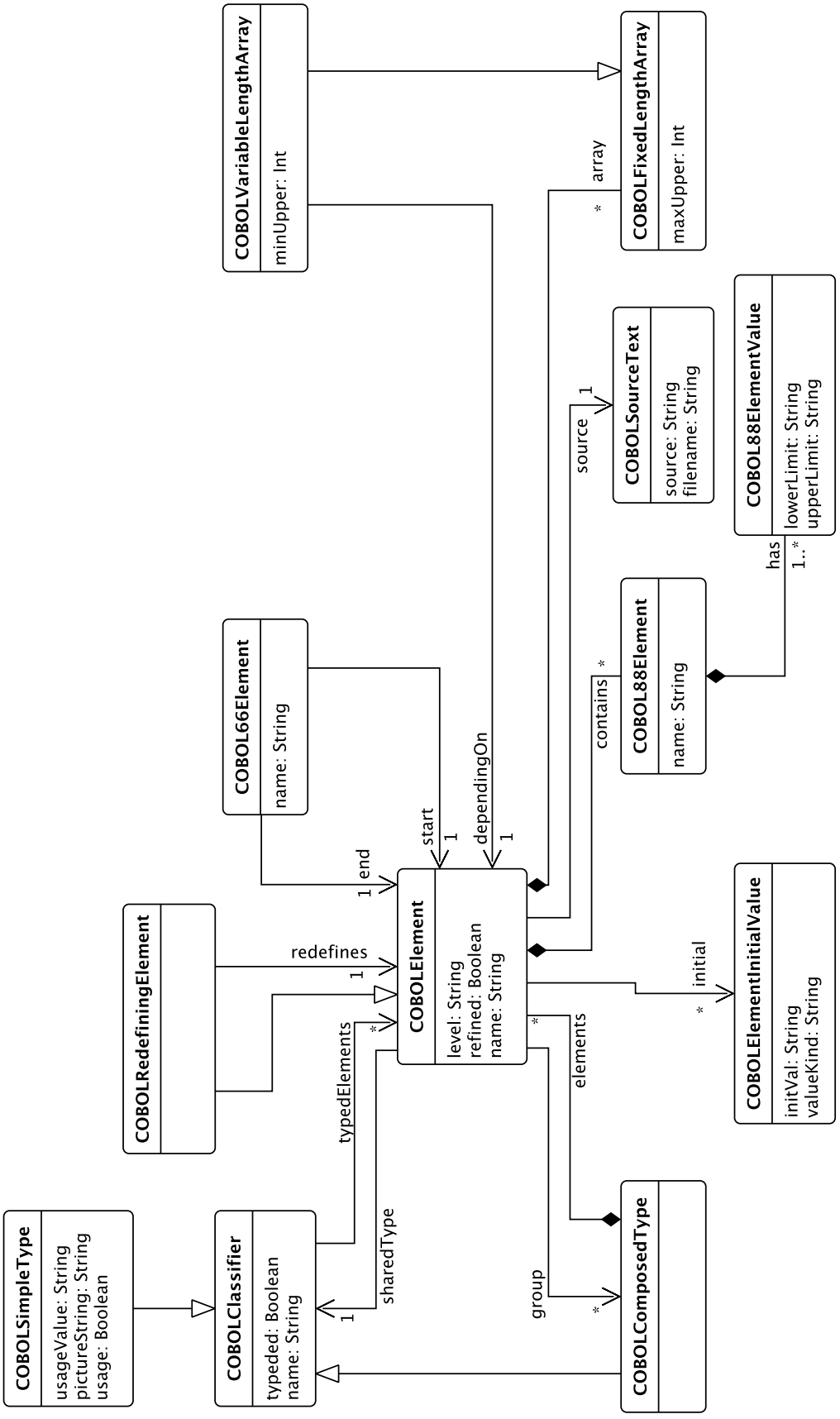


Figure A.5.: The COBOL metamodel.

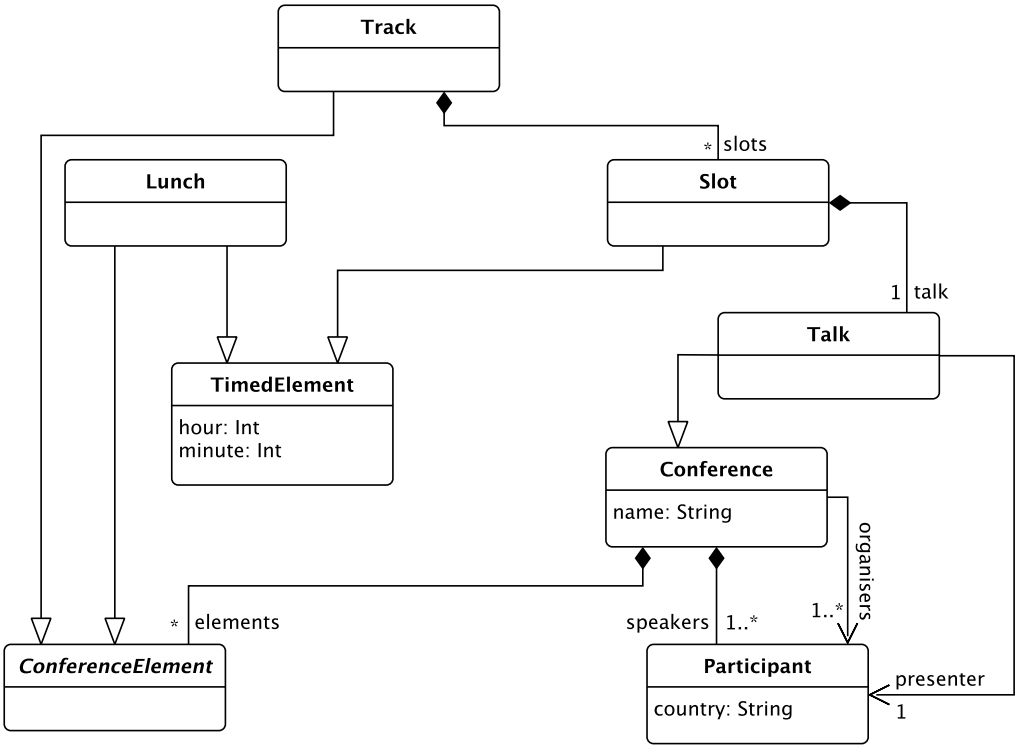


Figure A.6.: The Conference metamodel.

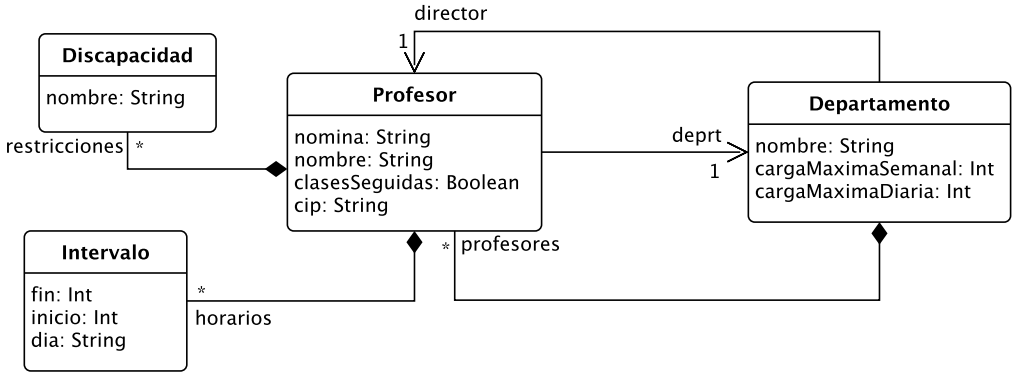


Figure A.7.: The Profesor metamodel.

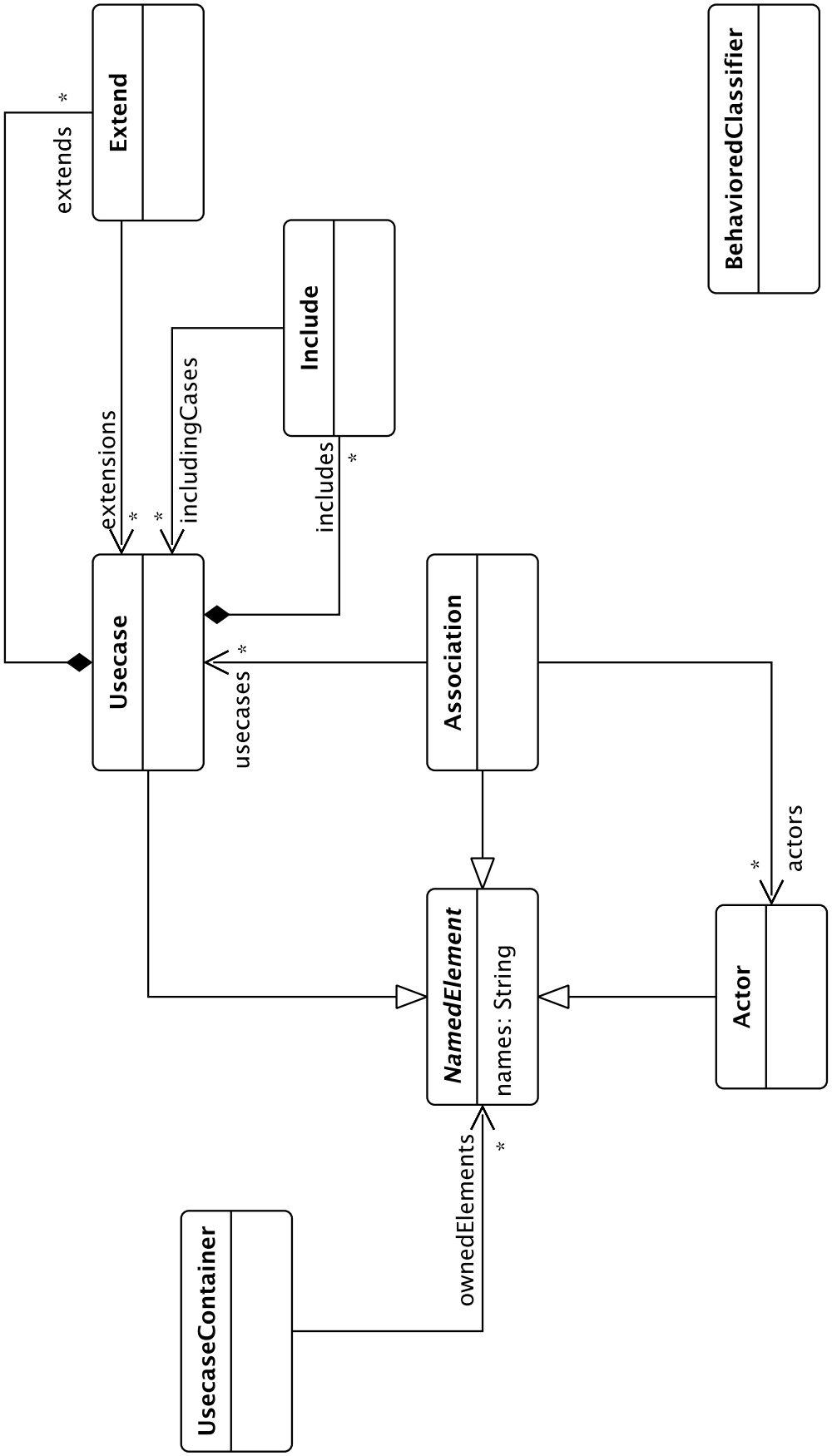


Figure A.8.: The Usecase metamodel.

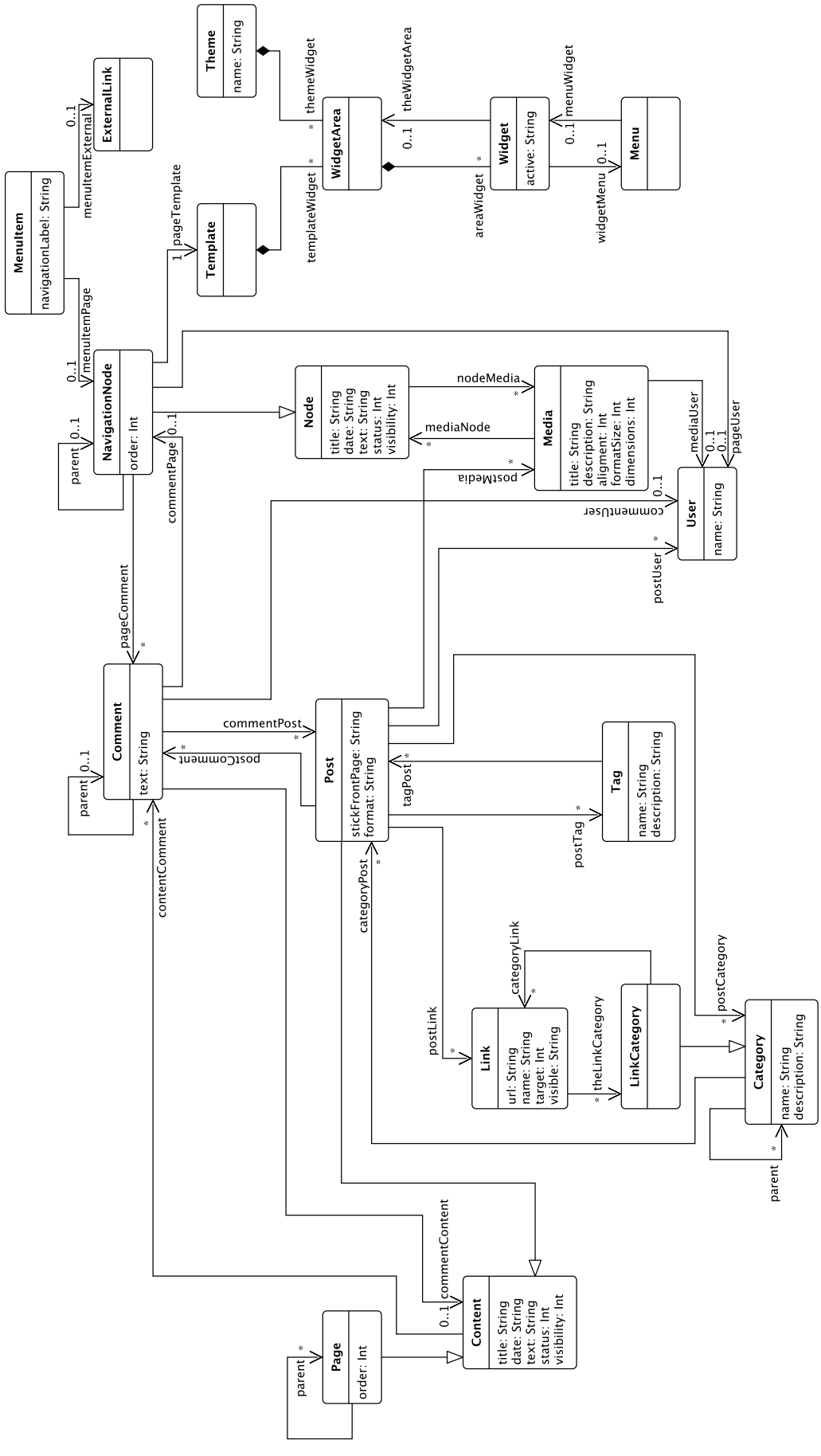


Figure A.9.: The Wordpress metamodel.

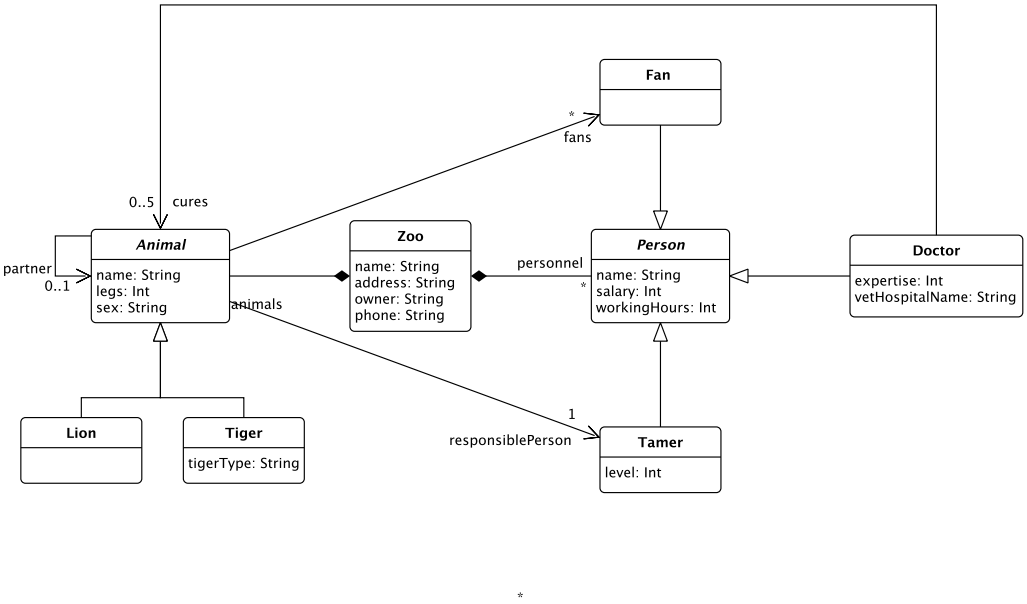


Figure A.10.: The Zoo metamodel.

Bibliography

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

[2] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Software Engineering*, vol. 1, no. 1, pp. 1–182, 2012.

[3] Object Management Group, "Meta object facility (MOF) core specification," ONLINE, 2014, http://www.omg.org/mof/.

[4] D. S. Kolovos, N. Matragkas, H. H. Rodríguez, and R. F. Paige, "Programmatic muddle management," *XM 2013–Extreme Modeling Workshop*, 2013.

[5] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara, "Example-driven meta-model development," *Software & Systems Modeling*, pp. 1–25, 2013.

[6] D. Wüest, N. Seyff, and M. Glinz, "Flexisketch: A mobile sketching tool for software modeling," in *Mobile Computing, Applications, and Services*. Springer, 2013, pp. 225–244.

[7] A. Coyette, S. Schimke, J. Vanderdonckt, and C. Vielhauer, "Trainable sketch recognizer for graphical user interface design," in *IFIP Conference on Human-Computer Interaction*. Springer, 2007, pp. 124–135.

[8] M. Kuhrmann, G. Kalus, and A. Knapp, "Rapid prototyping for domain-specific languages-from stakeholder analyses to modelling tools." *Enterprise Modelling and Information Systems Architectures*, vol. 8, no. 1, pp. 62–74, 2013.

[9] F. Javed, M. Mernik, J. Gray, and B. R. Bryant, "MARS: A metamodel recovery system using grammar inference," *Information and Software Technology*, vol. 50, no. 9, pp. 948–968, 2008.

[10] T. M. Mitchell, "Machine learning. 1997," *Burr Ridge, IL: McGraw Hill*, vol. 45, 1997.

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

[12] B. Grammel, S. Kastenholz, and K. Voigt, "Model matching for trace link generation in model-driven software development," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 609–625.

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

[14] M. Wallace, S. Novello, and J. Schimpf, "ECLiPSe: A platform for constraint logic programming," *ICL Systems Journal*, vol. 12, no. 1, pp. 159–200, 1997.

[15] A. Zolotas, D. S. Kolovos, N. Matragkas, and R. F. Paige, "Assigning semantics to graphical concrete syntaxes," in *XM 2014–Extreme Modeling Workshop*, 2014, p. 12.

[16] A. Zolotas, N. Matragkas, S. Devlin, D. Kolovos, and R. Paige, "Type inference in flexible model-driven engineering," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, G. Taentzer and F. Bordeleau, Eds. Springer International Publishing, 2015, vol. 9153, pp. 75–91.

[17] A. Zolotas, N. Matragkas, S. Devlin, D. S. Kolovos, and R. F. Paige, "Type inference using concrete syntax properties in flexible model-driven engineering," *1st Flexible Model-Driven Engineering Workshop*, 2015.

[18] A. Zolotas, N. Matragkas, D. S. Kolovos, and R. F. Paige, "Flexible modelling for requirements engineering," 2015, p. 32.

[19] A. Zolotas, R. Clariso, N. Matragkas, D. S. Kolovos, and R. F. Paige, "Constraint programming for type inference in flexible model-driven engineering," *Computer Languages, Systems & Structures*, pp. –, 2016.

[20] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, p. 19, 2003.

[21] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[22] H. Ossher, R. Bellamy, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, M. Desmond, J. de Vries, A. Fisher, and S. Krasikov, "Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 848–864, 2010.

[23] K. Bak, D. Zayan, K. Czarnecki, M. Antkiewicz, Z. Diskin, A. Wasowski, and D. Rayside, "Example-driven modeling: model = abstractions + examples," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 1273–1276.

[24] J. L. C. Izquierdo and J. Cabot, "Community-driven language development," in *2012 4th International Workshop on Modeling in Software Engineering (MISE)*. IEEE, 2012, pp. 29–35.

[25] M. Volter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. [Online]. Available: http://www.dslbook.org

[26] N. Cross, *Developments in design methodology*. John Wiley & Sons, 1984.

[27] G. Fischer, "Symmetry of ignorance, social creativity, and meta-design," *Knowledge-Based Systems*, vol. 13, no. 7, pp. 527–537, 2000.

[28] K. J. Fernandes, "Interactive situation modelling in knowledge-intensive domains," *International Journal of Business Information Systems*, vol. 4, no. 1, pp. 25–46, 2009.

[29] J. S. Cuadrado, J. de Lara, and E. Guerra, "Bottom-up meta-modelling: An interactive approach," in *MODELS'12: ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems*, ser. LNCS 7590. Springer, 2012, pp. 3–19.

[30] J. L. C. Izquierdo and J. Cabot, "Enabling the collaborative definition of DSMLs," in *Advanced Information Systems Engineering*. Springer, 2013, pp. 272–287.

[31] G. Gabrysiak, H. Giese, A. Lüders, and A. Seibel, "How can metamodels be used flexibly," in *Proceedings of ICSE 2011 workshop on flexible modeling tools, Waikiki/Honolulu*, vol. 22, 2011.

[32] M. Kuhrmann, "User assistance during domain-specific language design," in *FlexiTools workshop*, 2011.

[33] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. Polack, "The design of a conceptual framework and technical infrastructure for model management language engineering," in *14th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE, 2009, pp. 162–171.

[34] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[35] U. Brandes, M. Eiglsperger, J. Lerner, and C. Pich, "Graph markup language (GraphML)," 2004.

[36] O. Lassila and R. R. Swick, "Resource description framework (RDF) model and syntax specification," 1999.

[37] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1231146

[38] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon object language (EOL)," in *Model Driven Architecture–Foundations and Applications*. Springer, 2006, pp. 128–142.

[39] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

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

[41] S. Ceri, P. Fraternali, and A. Bongio, "Web Modeling Language (WebML): a modeling language for designing web sites," *Computer Networks*, vol. 33, no. 1, pp. 137–157, 2000.

[42] C. J. Date and H. Darwen, *A guide to the SQL Standard: a user's guide to the standard relational language SQL*. Addison-Wesley Longman, 1993, vol. 55822.

[43] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[44] M. Fowler and R. Parsons, *Domain-specific languages*. Addison-Wesley Professional, 2010.

[45] Oracle. (2016, July) Java oracle. ONLINE. [Online]. Available: https://www.oracle.com/java/index.html

[46] Object Management Group, "Business Process Model and Notation Standard (formal/2011-01-03)," http://www.omg.org/spec/BPMN/, December 2013.

[47] ——, "Unified Modeling Language," http://www.omg.org/spec/UML/, June 2015.

[48] G. Scherp, *A Framework for Model-Driven Scientific Workflow Engineering*. BoD–Books on Demand, 2013.

[49] T. Berners-Lee and D. Connolly, "Hypertext markup language (html): A representation of textual information and metainformation for retrieval and interchange," *Rapport technique, IETF IIIR Working Group*, 1993.

[50] S. Efftinge and M. Völter, "oAW xText: A framework for textual DSLs," in *Workshop on Modeling Symposium at Eclipse Summit*, vol. 32, 2006, p. 118.

[51] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, "Derivation and refinement of textual syntax for models," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2009, pp. 114–129.

[52] The graphical modeling project (gmp). ONLINE. The Eclipse Foundation. [Online]. Available: http://www.eclipse.org/modeling/gmp/

[53] V. Viyović, M. Maksimović, and B. Perisić, "Sirius: A rapid development of dsm graphical editor," in *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, 2014, pp. 233–238.

[54] J. De Lara and H. Vangheluwe, "Atom3: A tool for multi-formalism and meta-modelling," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2002, pp. 174–188.

[55] D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, F. A. Polack, and G. Botterweck, "Taming EMF and GMF using model transformation," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 211–225.

[56] F. P. Andrés, J. De Lara, and E. Guerra, "Domain specific languages with graphical and textual views," in *International Symposium on Applications of Graph Transformations with Industrial Relevance*. Springer, 2007, pp. 82–97.

[57] L. M. Garshol, "BNF and EBNF: What are they and how do they work," *acedida pela última vez em*, vol. 16, 2003.

[58] Object Management Group, "XML metadata interchange," http://www.omg.org/spec/XMI/2.5.1, June 2015.

[59] T. J. Grose, G. C. Doney, and S. A. Brodsky, *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley & Sons, 2002, vol. 21.

[60] Object Management Group, "Object Constraint Language," http://www.omg.org/spec/OCL/, February 2014.

[61] J. D. Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multi-level modelling," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 12, 2014.

[62] D. Kolovos, R. Paige, and F. Polack, "The Epsilon Transformation Language," *Theory and Practice of Model Transformations*, pp. 46–60, 2008.

[63] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 128–138.

[64] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró, "Viatra-visual automated transformations for formal verification and validation of uml models," in *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*. IEEE, 2002, pp. 267–270.

[65] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, "The Epsilon generation language," in *Model Driven Architecture–Foundations and Applications*. Springer, 2008, pp. 1–16.

[66] S. Efftinge and C. Kadura, "OpenArchitectureWare 4.1 Xpand language reference," 2006.

[67] B. Klatt, "Xpand: A closer look at the model2text transformation language," Institute for Program Structures and Data Organization (IPD), Tech. Rep., July 2007.

[68] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire, "Acceleo user guide," *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, vol. 2, 2006.

[69] S. Efftinge and S. Zarnekow, "Extending java–xtend: a new language for java developers," *PragPub, The Pragmatic Bookshelf*, no. 30, pp. 5–11, 2011.

[70] S. Efftinge, "Xtend language," 2016. [Online]. Available: http://www.eclipse.org/xtend

[71] M. Elaasar and L. Briand, "An overview of UML consistency management," *Carleton University, Canada, Technical Report SCE-04-18*, 2004.

[72] D. Kolovos, *An extensible platform for specification of integrated languages for model management*. University of York, 2008.

[73] D. S. Kolovos, R. F. Paige, and F. A. Polack, "On the evolution of OCL for capturing structural constraints in modelling languages," in *Rigorous Methods for Software Construction and Analysis*. Springer, 2009, pp. 204–218.

[74] F. Jouault and J. Bezıvin, "Using atl for checking models," in *Proc. International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia (September 2005)*. Citeseer, 2005.

[75] Eclipse, "The Eclipse Foundation open source community," http://www.eclipse.org, August 2016.

[76] C. Daly, "Emfatic language reference," *IBM alphaWorks*, 2004, http://www.eclipse.org/epsilon/doc/articles/emfatic/.

[77] D. Flanagan, *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[78] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Merging models with the Epsilon merging language (EML)," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 215–229.

[79] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, "GraphML progress report structural layer proposal," in *Graph Drawing*. Springer, 2002, pp. 501–512.

[80] P. Mohagheghi and V. Dehlen, "Where is the proof? - A review of experiences from applying mde in industry," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2008, pp. 432–443.

[81] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 471–480.

[82] M. Bone and R. Cloutier, "The current state of model based systems engineering: Results from the OMG SysML request for information 2009," in *Proceedings of the 8th Conference on Systems Engineering Research*, 2010.

[83] T. Weigert and F. Weil, "Practical experiences in using model-driven engineering to develop trustworthy computing systems," in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*, vol. 1. IEEE, 2006, pp. 8–pp.

[84] B. Selic, "What will it take? A view on adoption of model-based methods in practice," *Software & Systems Modeling*, vol. 11, no. 4, pp. 513–526, 2012.

[85] R. L. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE software*, vol. 18, no. 3, pp. 112–111, 2001.

[86] F. Steimann and T. Kühne, "Coding for the code," *Queue*, vol. 3, no. 10, pp. 44–51, 2005.

[87] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[88] yWorks, "yEd - Graph editor," https://www.yworks.com/products/yed, August 2016.

[89] The GNOME Project, "Dia diagram creation program," https://wiki.gnome.org/Apps/Dia, November 2013.

[90] Microsoft, "Microsoft PowerPoint," https://products.office.com/en-us/powerpoint, August 2016.

[91] B. Biafore, *Visio 2007 Bible*. John Wiley & Sons, 2007.

[92] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[93] OMG Architecture Board ORMSC, "Model driven architecture (MDA)," http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, July 2001, OMG document number ormsc/2001-07-01.

[94] J. De Lara and E. Guerra, "Deep meta-modelling with metadepth," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2010, pp. 1–20.

[95] H. Ossher, R. Bellamy, D. Amid, A. Anaby-Tavor, M. Callery, M. Desmond, J. de Vries, A. Fisher, T. Frauenhofer, S. Krasikov *et al.*, "Business Insight Toolkit: Flexible pre-requirements modeling," in *2009 31st International Conference on Software Engineering-Companion Volume*, 2009.

[96] M. Kuhrmann, G. Kalus, E. Wachtel, and M. Broy, "Visual process model design using domain-specific languages," in *Proceedings of SPLASH Workshop on Flexible Modeling Tools*, vol. 2010, 2010.

[97] M. Kuhrmann, G. Kalus, M. Then, and E. Wachtel, "From design to tools: process modeling and enactment with PDE and PET," in *Proceedings of Third International Workshop on Academic Software Development Tools and Techniques (WASDeTT-3), co-located with the 25th IEEE/ACM International Conference on Automated Software Engineer*, 2010.

[98] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-specific development with visual studio DSL tools*. Pearson Education, 2007.

[99] B. Roth, M. Jahn, and S. Jablonski, "On the way of bottom-up designing textual domain-specific modelling languages," in *Proceedings of the ACM workshop on Domain-specific modeling*, 2013, pp. 51–56.

[100] J. Gallardo, C. Bravo, and M. A. Redondo, "A model-driven development method for collaborative modeling tools," *Journal of Network and Computer Applications*, vol. 35, no. 3, pp. 1086–1105, 2012.

[101] N. Pinkwart, H. U. Hoppe, L. Bollen, and E. Fuhlrott, "Group-oriented modelling tools with heterogeneous semantics," in *International Conference on Intelligent Tutoring Systems*. Springer, 2002, pp. 21–30.

[102] N. Avouris, M. Margaritis, and V. Komis, "Modelling interaction during small-group synchronous problem-solving activities: The synergo approach," in *Proceedings of ITS 2004 workshop on designing computational models of collaborative learning interaction*, 2004, pp. 13–18.

[103] A. Pescador and J. de Lara, "DSL-maps: from requirements to design of domain-specific languages," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 438–443.

[104] T. Buzan and B. Buzan, "The mind map book: How to use radiant thinking to maximize your brain's untapped potential," 1996.

[105] J. J. López-Fernández, A. Garmendia, E. Guerra, and J. de Lara, *Example-Based Generation of Graphical Modelling Environments*. Cham: Springer International Publishing, 2016, pp. 101–117.

[106] M. Famelis, R. Salay, and M. Chechik, "Partial models: Towards modeling and reasoning with uncertainty," in *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 573–583.

[107] F. Rabbi, Y. Lamo, I. C. Yu, L. M. Kristensen, and L. Michael, "A diagrammatic approach to model completion," in *4th Workshop on the Analysis of Model Transformations (AMT)@ MODELS*, vol. 15, 2015.

[108] S. Sen, B. Baudry, and D. Precup, "Partial model completion in model driven engineering using constraint logic programming," in *International Conference on the Applications of Declarative Programming*. Citeseer, 2007.

[109] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, and A. Rutle, "DPF workbench: A diagrammatic multi-layer domain specific (meta-) modelling environment," in *Computer and Information Science 2012*. Springer, 2012, pp. 37–52.

[110] K. R. Apt and M. Wallace, *Constraint logic programming using ECLiPSe*. Cambridge University Press, 2006.

[111] W. Clocksin and C. S. Mellish, *Programming in PROLOG*. Springer Science & Business Media, 2003.

[112] M. Antkiewicz, K. Bak, K. Czarnecki, Z. Diskin, D. Zayan, and A. Wasowski, "Example-driven modeling using clafer." in *MDEBE@MoDELS*, vol. 1104. CEUR-WS.org, 2013, pp. 32–41.

[113] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: unifying class and feature modeling," *Software & Systems Modeling*, pp. 1–35, 2015.

[114] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[115] S. Schimke, C. Vielhauer, and J. Dittmann, "Using adapted Levenshtein distance for on-line signature authentication," in *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, vol. 2. IEEE, 2004, pp. 931–934.

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

[117] H. Cho, J. Gray, and E. Syriani, "Creating visual domain-specific modeling languages from end-user demonstration," in *2012 ICSE Workshop on Modeling in Software Engineering (MISE)*. IEEE, 2012, pp. 22–28.

[118] H. Cho and J. Gray, "Design patterns for metamodels," in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*. ACM, 2011, pp. 25–32.

[119] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[120] J. Sprinkle and G. Karsai, "A domain-specific visual language for domain model evolution," *Journal of Visual Languages & Computing*, vol. 15, no. 3, pp. 291–307, 2004.

[121] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[122] G. Karsai, M. Maroti, Á. Lédeczi, J. Gray, and J. Sztipanovits, "Composition and cloning in modeling and meta-modeling," *IEEE Transactions on Control Systems Technology*, vol. 12, no. 2, pp. 263–278, 2004.

[123] J. Clark *et al.*, "XSL transformations (XSLT)," *World Wide Web Consortium (W3C). URL http://www. w3. org/TR/xslt*, p. 103, 1999.

[124] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer, "LISA: An interactive environment for programming language development," in *International Conference on Compiler Construction*. Springer, 2002, pp. 1–4.

[125] H. Jiawei and M. Kamber, "Data mining: concepts and techniques," *San Francisco, CA, itd: Morgan Kaufmann*, vol. 5, 2001.

[126] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[127] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[128] M. A. Hearst, S. T. Dumais, E. Osman, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their Applications*, vol. 13, no. 4, pp. 18–28, 1998.

[129] B. Yegnanarayana, *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

[130] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics, Springer, Berlin, 2001, vol. 1, pp. 587–604.

[131] J. Surowiecki, *The wisdom of crowds*. Anchor, 2005.

[132] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip *et al.*, "Top 10 algorithms in data mining," *Knowledge and information systems*, vol. 14, no. 1, pp. 1–37, 2008.

[133] A. J. Smola and B. Schölkopf, *Learning with kernels*. Citeseer, 1998.

[134] B. Müller, J. Reinhardt, and M. T. Strickland, *Neural networks: an introduction*. Springer Science & Business Media, 2012.

[135] J. E. Dayhoff and J. M. DeLeo, "Artificial neural networks," *Cancer*, vol. 91, no. S8, pp. 1615–1635, 2001.

[136] V. Krkova, "Kolmogorov's theorem and multilayer neural networks," *Neural networks*, vol. 5, no. 3, pp. 501–506, 1992.

[137] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[138] J. W. Lloyd, *Foundations of logic programming*. Springer Science & Business Media, 2012.

[139] R. Kowalski and D. Kuehner, "Linear resolution with selection function," *Artificial Intelligence*, vol. 2, no. 3-4, pp. 227–260, 1971.

[140] C. Alain, F. Didier, R. Pasero, P. Roussel, and J. Trudel, "Répondre à," 1971.

[141] A. Colmerauer and P. Roussel, "The birth of PROLOG," in *History of programming languages—II*. ACM, 1996, pp. 331–367.

[142] R. L. Sites, "ALGOL W reference manual," 1972.

[143] J. Wielemaker, "An overview of the SWI-Prolog programming environment." *WLPE*, vol. 3, pp. 1–16, 2003.

[144] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[145] C. A. González, F. Buettner, R. Clarisó, and J. Cabot, "EMFtoCSP: A tool for the lightweight verification of EMF models," in *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*. IEEE Press, 2012, pp. 44–50.

[146] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[147] J.-L. J. Imbert, "Linear constraint solving in clp-languages," in *Constraint Programming: Basics and Trends*. Springer, 1995, pp. 108–127.

[148] J. Wielemaker and A. Anjewierden, "Programming in XPCE/Prolog," *Roetersstraat*, vol. 15, p. 1018, 1992.

[149] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming," in *Proceedings of the 22th IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 547–548.

[150] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *the VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[151] P. A. Bernstein, J. Madhavan, and E. Rahm, "Generic schema matching, ten years later," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 695–701, 2011.

[152] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm (extended technical report)," 2001.

[153] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, "Fundamentals of algebraic graph transformation (monographs in theoretical computer science. an eatcs series)." 2006.

[154] J. R. Williams, A. Zolotas, N. D. Matragkas, L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. Polack, "What do metamodels really look like?" *EESSMOD@ MoDELS*, vol. 1078, pp. 55–60, 2013.

[155] J. R. Williams, R. F. Paige, D. S. Kolovos, and F. A. Polack, "Search-based model driven engineering," Technical Report YCS-2012-475, Department of Computer Science, University of York, Tech. Rep., 2012.

[156] T. M. Therneau, E. J. Atkinson *et al.*, "An introduction to recursive partitioning using the rpart routines," June 2015.

[157] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2016. [Online]. Available: https://www.R-project.org/

[158] A. Liaw and M. Wiener, "randomforest: Breiman and Cutler's random forests for classification and regression," https://cran.r-project.org/web/packages/randomForest/index.html, October 2015, version: 4.6-12.

[159] C. Spearman, "The proof and measurement of association between two things," *The American journal of psychology*, vol. 15, no. 1, pp. 72–101, 1904.

[160] K. Pearson, "Notes on the history of correlation," *Biometrika*, vol. 13, no. 1, pp. 25–45, 1920.

[161] J. Lee Rodgers and W. A. Nicewander, "Thirteen ways to look at the correlation coefficient," *The American Statistician*, vol. 42, no. 1, pp. 59–66, 1988.

[162] B. Meyer, *Object-oriented software construction*. Prentice Hall New York, 1988, vol. 2.

[163] R. Reiter, "On closed world data bases," in *Logic and data bases*. Springer, 1978, pp. 55–76.

[164] R. Debruyne and C. Bessiere, "Some practicable filtering techniques for the constraint satisfaction problem," in *In Proceedings of IJCAI '97*. Citeseer, 1997.

[165] E. Tsang, *Foundations of constraint satisfaction: the classic text*. BoD–Books on Demand, 2014.

[166] P. Prosser, "An empirical study of phase transitions in binary constraint satisfaction problems," *Artificial Intelligence*, pp. 81–109, 1996.

[167] D. G. Mitchell, *Proc. of the 8th International Conference on Principles and Practice of Constraint Programming (CP'2002)*. Berlin, Heidelberg: Springer, 2002, ch. Resolution Complexity of Random Constraints, pp. 295–310.

[168] Y. Gao and J. Culberson, "Resolution complexity of random constraint satisfaction problems: Another half of the story," *Discrete Applied Mathematics*, vol. 153, pp. 124 – 140, 2005.

[169] T. Soininen and I. Niemelä, "Developing a declarative rule language for applications in product configuration," in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 1999, pp. 305–319.

[170] Y. Dimopoulos, B. Nebel, and J. Koehler, "Encoding planning problems in nonmonotonic logic programs," in *European Conference on Planning*. Springer, 1997, pp. 169–181.

[171] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set solving." in *IJCAI*, vol. 7, 2007, pp. 386–392.

[172] G. Klyne and J. J. Carroll, "Resource description framework (RDF): Concepts and abstract syntax," 2006.

[173] L. Sabin-Wilson and M. Mullenweg, *WordPress for dummies*. John Wiley & Sons, 2011.

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