Type Inference in Flexible Model-Driven Engineering

Athanasios Zolotas, Nicholas Matragkas, Sam Devlin, Dimitrios S. Kolovos and Richard F. Paige

Department of Computer Science, University of York, UK
Email: {amz502, nicholas.matragkas, sam.devlin, dimitris.kolovos, richard.paige}@york.ac.uk

Abstract. In Model-Driven Engineering (MDE), models conform to metamodels. In *flexible modelling*, engineers construct example models with free-form drawing tools; these examples may later need to conform to a metamodel. Flexible modelling can lead to errors: drawn elements that should represent the same domain concept could instantiate different types; other drawn elements could be left untyped. We propose a novel *type inference* approach to calculating types from example models, based on the Classification and Regression Trees (CART) algorithm. We describe the approach and evaluate it on a number of randomly generated models, considering the accuracy and precision of the resultant classifications. Experimental results suggest that on average 80% of element types are correctly identified. In addition, the results reveal a correlation between the accuracy and the ratio of known-to-unknown types in a model.

1 Introduction

In traditional MDE approaches, engineers build models that conform to (typically pre-defined) metamodels. *Flexible modelling tools* [1, 2] seek to combine free-form modelling (e.g., sketching on a whiteboard) and more formal modelling (e.g., modelling with a MDE tool). Flexible modelling tools sacrifice some formality to facilitate exploratory modelling of the domain, but as a result cannot provide the powerful domain-specific editors generated by MDE tools. Flexible modelling is arguably accessible to domain experts, who may sketch elements that represent concepts of a future metamodel; these experts may also assign types to these drawn elements, when they believe it is suitable to do so.

When examples are constructed using flexible modelling tools, there is no guarantee that they will consistently obey syntactic and semantic rules that a rigid metamodel would impose: elements that in traditional MDE would instantiate the same class could have different types assigned by domain experts. This could happen for a variety of reasons:

1. *User input errors:* incorrectly assigning different types to nodes/edges that should have the same type due to typing errors.

2. *Changes:* a domain expert may assign a type to an element, and then later choose to assign a different type to a different instance of the same element.
3. *Inconsistencies:* the participation of multiple experts in building examples could lead to the assignment of different types to elements (nodes/edges) that are conceptually instances of the same type.
4. *Omissions:* for example, if the examples are large, it may be easier to overlook some elements, and not assign them types.

This paper addresses the challenges associated with identifying and managing omissions during type assignment in flexible modelling; such challenges need to be overcome in order to provide support for eventual transition from flexible to more rigorous (metamodel-based) modelling approaches.

There are at least two approaches that can help to address these challenges. The first is to provide a mechanism to validate that all elements drawn on the canvas have exactly one type assigned to them. If not, the domain expert could be prompted to assign missing types and resolve any inconsistencies; a constraint and repair language [3] can be used to support this. However, this approach may force users to make decisions about types when they are not ready to do so. Also, such approaches tend to reveal *all* omissions and inconsistencies at once, and so it can be difficult to find and repair specific problems.

The second alternative is that of *type inference* [4]: missing types could be inferred by computing and analysing matches between untyped and typed elements that share the same characteristics. The benefit of this approach is that users can avoid re-applying the same type to elements that are already defined in the diagram.

This paper contributes a novel approach to type inference for flexible models, allowing types to be calculated from example models using *classification algorithms*, specifically CART [5]. In our approach, the metamodel of the example models is *not* needed to perform the type inference, as it runs on instances only. We present the approach in detail, using an illustrative flexible modelling approach based on GraphML and the flexible modelling technique called *Muddles* [1]. We demonstrate the approach's accuracy, precision and limitations via experiments on a number of randomly generated models. The 80% success rate indicates that fully automated CART-based type derivation shows promise when applied to arbitrary muddles.

2 Related Work

In this section, we present literature from the fields of type or metamodel inference and model matching. Flexible modelling is also briefly summarised. We describe classification algorithms in more detail in Section 4.2.

Type inference has been widely studied in programming languages, particularly for dynamically typed languages. Type inference often relies on the Hindley-Milner [6] [7] algorithm and its extension by Milner and Damas [8]. In these approaches, program statements are reduced to basic constructs for which a type

is already known. Such approaches are challenging to apply in flexible modelling where there is no predefined abstract syntax.

Inferring types (or metamodels) from example models boils down to a *matching* problem: elements that are "sufficiently similar" may have similar or identical types. Model matching has been widely studied, particularly to support model differencing and versioning. What is important about much of this work is that different techniques for identifying identical or similar elements have been proposed; a classification was published in [9]. In [10], a model matching technique is used for the generation of traceability links in MDE. Matching is performed by checking the *name similarity* between the data nodes of two attributed graphs (that represent models). Alanen and Porres [11] use each element's unique MOF identifier to calculate the difference and union of models that belong to the same ancestor model; both approaches are of limited flexibility as they depend on names or persistent identifiers for inference. In [12] the authors present a signature-based matching approach that is used for model merging. The signatures are written manually and include the names of the attributes/operations of each element that should be checked when two elements are examined for their similarity in order to be merged.

In the domain of flexible modelling, Cho et al. [13] propose the use of example models to calculate the metamodel in a semi-automatic way. Example drawings, in [14], created by domain experts, are used as the basis for the definition of a metamodel. In [15], a tool for the recovery of the original metamodel that was evolved, using instances models is presented. Finally, in [1], users use a simple drawing tool to define example models which are then amenable to programmatic model management (validation, transformation). To the best of our knowledge, this paper presents the first application of type inference in the domain of flexible modelling; it also is the first application of CART as the predictive mechanism. In addition, in contrast with the methods presented in this section, the matching mechanism is not based on name similarity between the types, the attributes, etc. but on other structural and semantic characteristics, based on the assumption that different domain experts that will express example models to define the same domain may use different names to express the same behaviour.

3 Background: Muddles

In this section, the Muddles [1] flexible modelling approach is presented. We use Muddles to illustrate our approach to inference.

3.1 Overview

In Muddles, drawing editors are used for the construction of example models (in [1] yEd[1] which is based on GraphML is used to prove the concept). Language engineers draw examples and then annotate elements with types and attributes,

[1] `http://www.yworks.com/en/products_yed_about.html`

while also expressing relations (references and containments) between elements using edges and group containers. The annotated diagram is then automatically transformed to an intermediate Muddle (the Muddle metamodel is shown in Figure 1). Using the Epsilon platform [3], muddles can be consumed so that model management programs (e.g. transformations) can be executed against them.

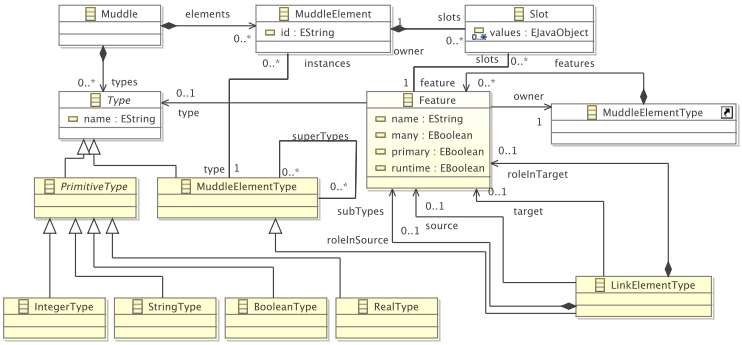


Fig. 1. The Muddle metamodel

3.2 Example

We illustrate "muddling" with an example, creating a language for zoos. The process starts with the creation of an example zoo diagram (see Fig. 2). Next, diagram elements are annotated with basic type information. For instance, one can define the type of the diamond shape as *Doctor* and the type of the directed edges from *Doctor* to *Animal* nodes (circles) as instances of the *cures* relationship. The types are not bound to the shape but to each element, meaning that in another example or even in the same drawing, a diamond can be of type *Doctor* while a different diamond can be of type *Animal*. Types and type-related information like properties (attributes of the type), roles and multiplicity of edges are specified using the appropriate fields in the yEd's custom properties dialog. More details about these properties are presented in [1].

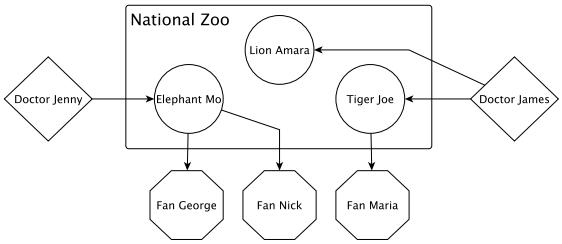


Fig. 2. An example Zoo diagram

Model management programs use this type-related information to access and manipulate elements of the diagram. For example, if the circle element (typed as *Animal*) has a String attribute named *name* assigned to it, then the following Epsilon Object Language (EOL) [16] script returns the names of all the elements of Type *Animal*. As such, muddles can be programmatically processed

like other models, without having to transform them to a more rigorous format (e.g., Ecore).

```
var animals = Animal.all();
for (a in animals) {
  ("Animal: " + a.name).println();
}
```

Listing 1.1. EOL commands executed on the drawing

4 Type Inference Approach

In this section we describe type inference for flexible models. An overview is shown in Figure 3. The source code for all the algorithms described in Sections 4 and 5 along with detailed instructions can be found at the paper's website[2].

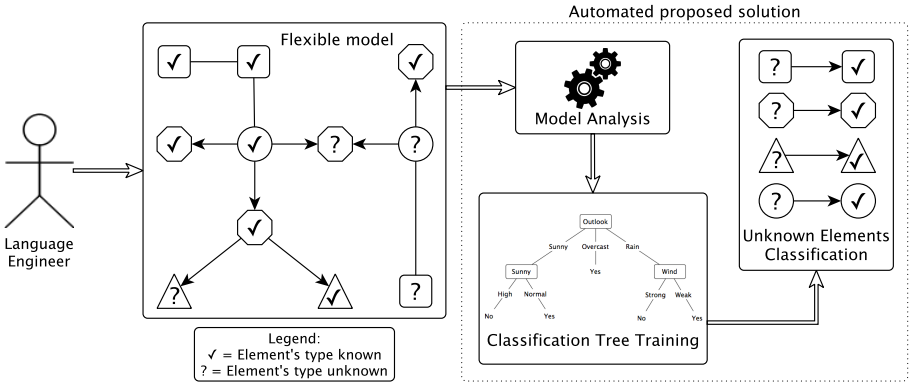


Fig. 3. An overview of the proposed approach

A language engineer constructs a flexible model using a GraphML-compliant drawing tool. The engineer annotates the model with as much type information as they see fit; after this, some elements will have known types, others will have no type. This annotated model is then analysed to extract characteristics of interest, and these characteristics are passed to the CART algorithm, which performs type inference. We now explain this process in more detail.

4.1 Model Analysis and Feature Selection

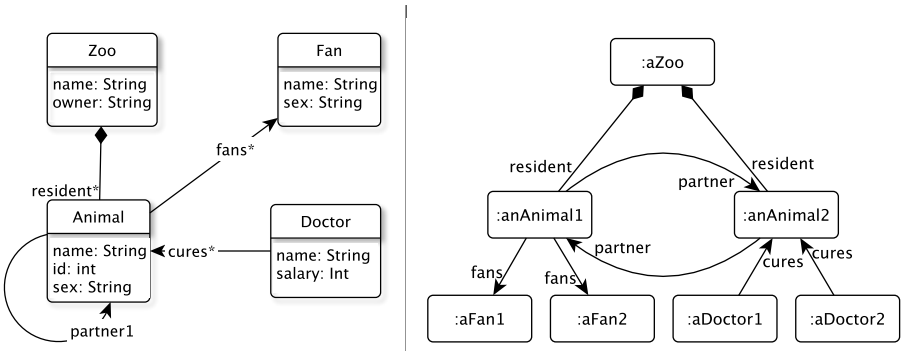
In order to be able to match untyped element with those that are typed, we first specify a set of characteristics that describe attributes of each element. Each characteristic is known as a *feature*; the set of characteristics for each node is a *feature signature*. At the end of the signature, the type of the element (if known) is also attached. In this approach we used a set of five features that are presented in Table 1. These were selected because our intention is to base the similarity measurement and prediction on structural and semantic characteristics of the models; arguably these features measure these characteristics. As mentioned in

[2] `http://www.zolotas.net/type-inference/`

Section 2, the names that domain experts use to express the same behaviour may vary, and our feature selection was based on this. However, we need to highlight that we do not claim that the names should be totally ignored as they can carry useful information. Methodologies proposed in Section 2 which base their similarity measurement on name matching could be combined with the approach we propose and possibly improve the prediction results. Plans for this combination are described in Section 7.

Table 1. Signature features for nodes

Name of Feature	**Description**
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. If a node is targeted by more than one references of the same type, only 1 instance of them is taken into account (unique references).
Number of different types of outgoing references	The number of all the types of references that come from that node. As above, multiple outgoing references of the same type are counted once.
Number of different types of children	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	The number of all the types that the node is contained in.



(a) Example Metamodel (b) Example Model

Fig. 4. Signature Example

The example muddle shown in Figure 4(b)) is conceptual instance of the example metamodel shown in Figure 4(a)). The feature signature of the node "aZoo" is [2,0,0,1,0,Zoo], as it has 2 attributes, no incoming or outgoing references, 2 children which are of the same type (so 1 unique child) and 0 parents. The 6th position of the signature declares the type of the element, which is useful for training the classification algorithm that will be used. Similarly, the feature signature of the node "anAnimal1" is [3,2,1,0,1,Animal] as it has 3 attributes, 2 unique outgoing references (partner and fans), 1 unique incoming reference (partner), 0 children and 1 parent. Its class is Animal. Note here that although the Animal class in the metamodel has a reference named "cures" of * multiplicity, it is not added to the signature of the "anAnimal1" element as it is not instan-

tiated in the model. This means that two entities of the same type may have different signatures. For instance, the signature of the element "anAnimal2" is [3,1,2,0,1,Animal]. This justifies the choice of using a classification algorithm to perform the matching. Classification algorithms do not look for perfect matches but are trained to classify elements by using each time those features that are most important in the specific set they are trained on, increasing the possibilities of identifying true positives even if two elements are not identical.

A simple querying algorithm was implemented as part of this approach. The algorithm parses each of the nodes in the drawing and constructs its feature signature. All the feature signatures for the elements in which the type is known are stored in a text file that will be used to train the classification algorithm.

4.2 Training and Classification

Classification algorithms are a form of supervised machine learning for finding hypotheses that approximate functions mapping input features to a discrete output class from a finite set of possible values. 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.

Many classification algorithms exist, some of the most established being decision trees, random forests, support vector machines and neural networks [17]. For this work we chose to use decision trees due to the interpretable output representing the hypothesis learnt. In practice other classification algorithms can often have higher accuracy, but will produce a hypothesis in a form that is not human readable. Given the high accuracy of classification achieved using decision trees in this application, these other algorithms were not deployed in favour of the aid to debugging provided by being able to interpret how the learnt hypothesis would classify future instances. Specifically, we used the rpart package (version 4.1-9)[3] that implements the functionality of Classification and Regression Trees (CART) [5] in R[4].

An example decision tree is illustrated in Figure 5. Internal nodes represent features (e.g. Number of Attributes, Parents, etc.), branches are labelled with values of the parent node 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 Fig. 5, a new instance with less than 3 attributes and 1 unique children would be classified as Zoo (path is highlighted in Fig. 5).

The success of a classification algorithm 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 accuracy of a model is the sum of true positives and negatives

[3] http://cran.r-project.org/web/packages/rpart/index.html
[4] http://www.r-project.org/

(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 overfitting bias in the dataset used for training. To overcome this k-fold classification can be implemented [18]. 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.

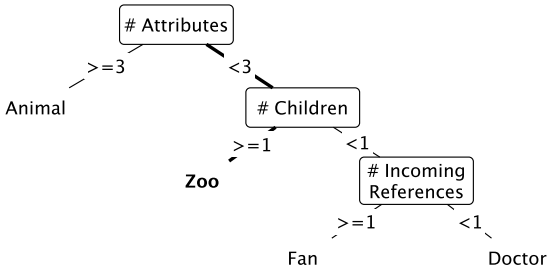


Fig. 5. Example Decision Tree

In our approach, the feature signatures list that contains the signatures of the known elements of the model are the input features to the CART algorithm. A trained decision tree is produced. This tree can be used to classify (identify the type of) the untyped nodes using their feature signatures.

5 Experiment

In this section, we present the experimentation process that was used to evaluate the proposed approach. An overview of the experiment is shown in Figure 6. Details about each step follow.

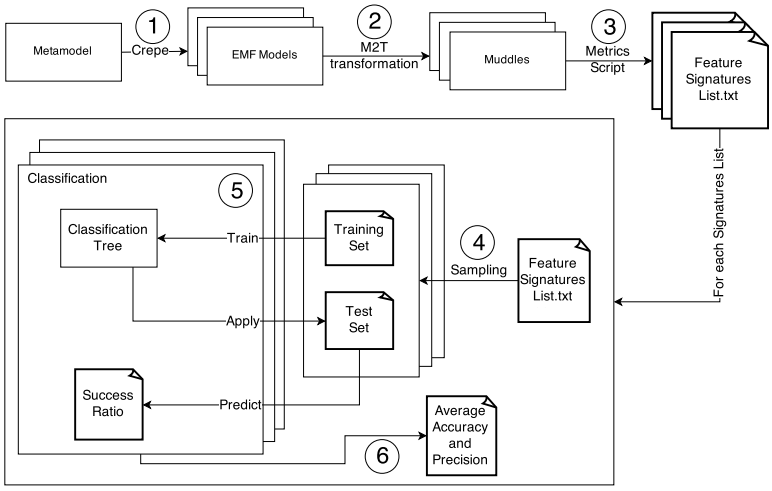


Fig. 6. The experimentation process

In order to carry out an evaluation we applied our approach to a number of publicly available metamodels that were collected as part of the work presented in [19]. For each of these metamodels we produced 10 random instances using the Crepe model generator tool [20] (step (1) in Figure 6). Crepe generates random models that conform to the provided metamodel, using a genetic algorithm. For our approach, the values that were assigned to the attributes of the instance models were randomly selected, as the content of the attributes does not affect the final feature signature of the element. We address any threats to validity of using randomly generated models in Section 6.3.

Having the experimentation models generated, we had to transform them into muddles and then randomly erase the types of some of them in order to simulate a real scenario where some node types are known while others are not. For that purpose, a Model-to-Text (M2T) transformation was implemented that transforms instances of Ecore metamodels to GraphML files that conform to the Muddles metamodel (step (2)).

Steps (1) and (2) could be avoided either by directly using available muddles or by drawing muddles on our own. The first solution was rejected because flexible modelling approaches are quite new and there is not a repository/zoo available that hosts such flexible models. The second was also rejected because it would be a time consuming process but more importantly 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.

After generating the muddles we are now able to extract the feature signature of each node. As mentioned in Section 3, each muddle conforms to the metamodel shown in Figure 1. Building atop the Epsilon EMC driver (see Section 3) we developed a script that iterates through all the nodes of the graph stored in the GraphML file and collects the information needed for each node: the number of attributes, unique outgoing and incoming references, children and parents. This information is encoded in the comma-separated format presented in Section 4.1 producing one list for each model that contains the feature signatures of the nodes of this model (step (3)).

At this point the types of all the nodes are known to us and saved in the features signatures list. In order to test the performance of the approach, we need to simulate the scenario where some nodes in a model are left without a type. To simulate this we perform the k-fold cross-validation described in Section 4.2. Each feature signature list is split into two parts, the training and the test set (step (4)). The first set contains the feature signatures of the nodes that will be used to train the CART algorithm. This set can be thought as the set that contains all the nodes of which the type is known in the hypothetical scenario where not all the nodes have types assigned to them. The second set contains the elements that will perform the role of the nodes of which the type is not known and will be used to evaluate the accuracy of the trained CART tree. In reality the types are known, but are ignored during the prediction process and are only used at the end, to identify if the predictions were correct. The standard 10-Fold

cross-validation will be used in our study: each feature signatures list will be split 10 times to produce 10 different couples of training and test sets. Each time, one training set will be used to train the algorithm and its coupling test set will evaluate its success (step (5)). The success ratio (also referred as accuracy) is defined as the total number of correct predictions to the total number of nodes with missing type. The precision of the algorithm incorrectly predicting each type is also calculated. The precision is defined as the number of true positives divided by the sum of the true and false positive predictions for this class. [21] The algorithm is then reset and is trained and evaluated with the next couple of training and test sets, respectively. At the end the average accuracy and precision are calculated (step (6)).

The size of the training and the test set may be important in the algorithm success. We perform the same process for 7 different sampling rates: 30%, 40%, 50%, 60%, 70%, 80% and 90%. For example, in the 80% sampling experiment, 80% of the nodes are placed in the training set and the rest 20% in the test set. The sampling, the training and the prediction are carried out in R using the RPart library for the classification. Custom-made scripts were implemented to automate the calculations of the accuracy, its average values and the precision for each type.

6 Results & Discussion

Before discussing the results, we present data on the random generated models that were used in the experimentation process (see Table 2). The smallest metamodel, a toy example, comprises only 2 types. The largest metamodel is the one that is used to describe Wordpress Content Management System websites (taken from [19]) with 19 different types of classes. On average the test metamodels had 7.2 types with a median of 6. These numbers do not include abstract classes but only those that could be instantiated in the random models. For each metamodel, 10 models were instantiated each of which was of a different size. The size of the smallest (Min) and the largest (Max) instance model for each metamodel is shown in Table 2. The average number of elements for the instances of each metamodel are also given. 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 the that of applying it to real muddles. All raw data, results and charts can be found at the paper's website[5].

6.1 Quantitative Analysis

As discussed in the previous section, 10 random models were instantiated from each of the 10 metamodels. Seven different sampling rates were applied to each

[5] http://www.zolotas.net/type-inference/

of these models and the CART was 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). A summary of the results is presented in Table 3. 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 have only one instance available in contrast with the 10 random instances of the other metamodels and thus not the variability needed to extract safe conclusions from it.

Table 2. Data summary table

Model Name	#Types	Min	Max	Average #Elements in instances
Chess	2	17	26	21.3
Professor	4	25	36	29.2
Conference	4	30	61	42.5
Zoo	5	47	73	57
Ant	6	53	78	65.3
Use Case	6	35	71	54.2
Bugzilla	7	21	56	39.9
BibTeX	8	56	106	78.8
Cobol	11	33	92	63.7
Wordpress	19	42	71	58.6
Muddle	20	105	105	105

In Table 3, one can see the average accuracy for all the models of each metamodel split into columns based on the sampling rate that was used each time. For instance, the highlighted value **0.81** indicates that on average, 81% of the missing types were successfully predicted for the 10 instance models of the Use Case motamodel, using 70% sampling rate.

Considering the raw values, there are some cases where even with small sampling rates (30%, 40%) all of the missing elements' types were successfully identified. More specifically, for smaller models (that were produced from metamodels with fewer than 5 types) the average success ratio was above 98.9% (between all the 7 different sampling rates). For metamodels with more types (> 6), that average dropped to between 53.9% and 79%. 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.

This observation leads us to two interesting questions, shown under the column labeled *Corel. 1* and the row labeled *Corel. 2*. Below are the definition of these correlations.

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

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

As expected, the correlation coefficient values for Corel. 1 indicate a strong or perfect dependency for all the metamodels, except one (BibTeX). The correlation

Table 3. Results summary table

		Average Success Ratio (Accuracy) for Different Sampling Rates								
Model Name	**#Types**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Corel. 1**
Chess	2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.000	-
Professor	4	0.96	0.97	0.98	0.98	0.98	1.00	0.99	0.980	0.90
Conference	4	0.87	0.92	0.95	0.97	0.98	0.99	0.98	0.951	0.90
Zoo	5	0.95	0.98	0.99	1.00	1.00	1.00	1.00	0.989	0.83
Ant Scripts	6	0.69	0.72	0.74	0.74	0.76	0.75	0.77	0.739	0.92
Use Case	6	0.75	0.78	0.79	0.8	**0.81**	0.79	0.81	0.790	0.82
Bugzilla	7	0.47	0.52	0.54	0.56	0.56	0.58	0.54	0.539	0.75
BibTeX	8	0.66	0.67	0.68	0.67	0.66	0.68	0.69	0.673	0.62
Cobol	11	0.53	0.59	0.62	0.65	0.66	0.67	0.67	0.627	0.92
Wordpress	19	0.44	0.51	0.61	0.66	0.72	0.74	0.77	0.636	0.98
Muddle	*20*	*0.55*	*0.56*	*0.59*	*0.65*	*0.59*	*0.67*	*0.64*	*0.607*	*0.82*
	Avg.	0.732	0.766	0.790	0.803	0.813	0.82	0.822		0.94
	Corel. 2	-0.80	-0.78	-0.70	-0.65	-0.57	-0.55	-0.49	-0.67	
	Corel. 3	0.77	0.72	0.64	0.58	0.53	0.49	0.45		

coefficient value on the averages of the accuracy for the different sampling rates is also strong (0.94). This means that prediction scores increase as training sets become larger. Regarding the second correlation (Corel. 2) we observe a strong (negative) correlation between the number of types in a metamodel and the success score when the sampling rate is lower than 50%. The correlation becomes moderate or weak when the sampling rate is larger than 60%. The outcome from this observation is that when a drawing is left heavily untyped (less than half of the elements are having a type assigned to them) the success score is affected by the number of the different types in the drawing: fewer types lead to better results. In contrast, if the drawing has more information (more than 60% of the nodes have a type assigned to them) then the number of the types of the envisioned metamodel doesn't affect the prediction performance.

The above metrics take only into account the sampling rate. However, it is of interest to check if the frequency that each type appears to a drawing, increases the prediction performance of the algorithm. We define frequency as the ratio of the total number of different types to the total number of elements in the drawn model. The results are shown in Table 3 (row labelled as *Corel. 3*) while the definition of this correlation is the following.

Corel. 3: How strong is the dependency between the frequency with which a type appears in a drawing and the success score?

Analysing the results, we are led to the same conclusions as with Corel. 2: For low sampling rates ($< 50\%$), the frequency with which a type appears in a drawing strongly affects its correct prediction chances. If the sampling rate increases, this relation no longer exists (or is moderate).

Finally, we analysed the precision of the classifications. Precision is interpreted as the certainty in predicting each specific class. Its values vary from 0 (the algorithm never predicted correctly this specific type) to 1 (the algorithm

always predicted correctly this type). The calculation of the precision scores could be of value in our prediction mechanism because one could initially perform a simulation of the prediction in the set that contains all the known types by splitting it into a new training set and a new test set, running the CART algorithm and calculating the precision value for each type based on this simulated training. Then proceed by re-training the CART on the whole original training set and predict the types of the nodes originally left untyped. If a node is predicted to be of a type that in the previously described simulation had a high precision, then assign the type automatically. If the predicted type scores a very low precision value, then suggest the type for the node but mark this as an assignment that needs manual reviewing by the domain expert (semi-automatic assignment).

6.2 Qualitative Analysis

In order to better understand the quantitative results and expose useful information on how to improve the performance of the approach we examined the results from a qualitative perspective, too. Our goal was to identify common patterns and characteristics, if any, in metamodels, where the performance of the approach was lower. Taking a closer look at all the experimental results of the Bugzilla metamodel we identified that all incorrect predictions occurred between types that extend the same abstract class. More specifically, the 4 types (Keywords, DependsOn, Cc, Blocks) that extend the same class named "StringElt" were all identified as being of the same type: the one with the greatest presence in the training data. Looking at the metamodel, these 4 classes had no attributes, references or containment relations assigned to them which is a type of structure, modelling inheritance [22] with no concrete differentiating characteristics.

The same behaviour was also discovered in the BibTeX metamodel. More specifically, wrong predictions occurred between types that were extending the same class but their differentiating point was that they both had an extra attribute than the parent class. The same issue occurred between classes that had the same grandparent, where their parents had no differentiating characteristics.

Such a behaviour is expected as the elements that belong to the category described above, share the same characteristics that inherit from their ancestors. In addition, they do not have any of their own specific characteristics (Bugzilla scenario) or they have the same feature which differs (e.g. both have an extra attribute as in the Bibtex scenario). This means that these elements share exactly the same features signature. A way to tackle such a behaviour could be the introduction of other features which are not calculated based on semantic characteristics that are always the same in such situations. Ideas on the introduction of new features that could solve this problem are discussed in Section 7. However, having such a behaviour might be of interest if the goal is not restricted to type inference but is extended to metamodel inference. CART supports the functionality of collecting such entities that share enough common characteristics but are of different type in the same "bucket". This could also serve in tackling problems 1-3 identified in Section 1.

6.3 Threats to Validity

In the experimentation process the data for testing the performance of the proposed approach was generated using a random model generator. There are two issues with this. The first is that we are using models (that conform to a metamodel) and not *muddles*, to run the experiments. This was done for largely pragmatic reasons: we have an efficient model generator available; no muddle generator currently exists; and we wanted to evaluate the feasibility of using the CART-based approach for type inference first, before carrying out more detailed experiments on user-created flexible models. We do not believe that the use of models instead of muddles will have significant impact on the experimental results, because the accuracy of our classification algorithm depends only on the features identified in Table 1; randomly generated models and muddles will not be observably 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 fact. However, other user-defined models and muddles *may* differ - and as such our future work will involve conducting experiments with more user-created muddles.

A second issue in using 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 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 implementation of a post-generation script that would randomly delete attributes from the elements to make sure that different instances will not always have the same number of attributes.

Ten metamodels were used in total to generate testing models. These ten metamodels were picked randomly from a set of 500 metamodels and the number of different concrete types in each of them varied from 2 up to 19. It would be of interest to apply the approach to even larger metamodels, although our experience (extracted from experimenting with the real muddle provided in this paper) suggests that having a flexible model with more than 20 different types is marginal and is not a terribly realistic scenario – generally, language engineers switch to more formal metamodelling infrastructure for larger languages. The number of instances for each metamodel and the number of repeats (700 runs for each metamodel) that the experiment was run on each of them is sufficient as it complies with the standard 10-fold methodology used in the domain of classification algorithms.

7 Conclusions and Future Work

We have proposed and evaluated an approach to support type inference for flexible modelling, thus contributing towards making it easier to transition from (untyped) example models to MDE models. More specifically, we have presented a type inference approach to flexible modelling based on CART. The CART algorithm is trained with elements that have been typed (e.g., by domain experts),

in order to predict the types of the elements that have been left untyped. Experiments suggest that the average prediction success ratio was 80% of the elements between all the generated models. A positive correlation between the ratio of known to unknown elements with the success score was also identified, along with a threshold in sampling after which the success score is not affected by the number of total types in the tested model.

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 be applied directly to traditional MDE: the CART algorithm could be used, for example, to infer types from an already-typed model, which may potentially reveal poor or incorrect type assignments or misuses of the metamodel.

In the future, we plan to introduce and test additional features that could be used to characterise the nodes other than the five that we used in this paper. These new features are inspired by the fact that in flexible modelling there may be spatial and graphical characteristics that could be useful – for example, the shape of the nodes, their size or their color, or the semantics that these characteristics denote [23]. In essence, we will experiment with the extent to which concrete syntax information can be used to further enrich the CART classification. In addition, the names that the domain experts choose to assign to the semantic characteristics could also be assessed to improve the predictions (e.g. the name of the attributes of each class). As mentioned, we base this work on the assumption that domain experts may use different naming conventions to express the same behaviour however we could overcome this by assigning custom weight to the importance of name-matching feature: if the examples are generated by a lot of different people then decrease the impact of the name matching in the prediction; if they are generated by a few then increase it. Of interest is the calculation of the level of importance of each feature (those used in this approach and those proposed in this last paragraph) in the classification decision. This will help us identify a set of features that maximises the classification performance and thus discard those that do not offer any valuable information to the algorithm or even reduce its performance.

Acknowledgments

This work was carried out in cooperation with Digital Lightspeed Solutions Ltd, and was supported by the EPSRC through the LSCITS initiative and part supported by the EU, through the MONDO FP7 STREP project (#611125).

References

1. Kolovos, D.S., Matragkas, N., Rodríguez, H.H., Paige, R.F.: Programmatic muddle management. XM 2013–Extreme Modeling Workshop (2013) 2
2. Gabrysiak, G., Giese, H., Lüders, A., Seibel, A.: How can metamodels be used flexibly. In: Proceedings of ICSE 2011 workshop on flexible modeling tools, Waikiki/Honolulu. Volume 22. (2011)
3. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.: The design of a conceptual framework and technical infrastructure for model management

language engineering. In: Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on, IEEE (2009) 162–171
4. Mitchell, J.C.: Concepts in programming languages. Cambridge University Press (2003)
5. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and regression trees. CRC press (1984)
6. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the american mathematical society (1969) 29–60
7. Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3) (1978) 348–375
8. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1982) 207–212
9. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: An analysis of approaches to support model differencing. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. CVSM '09, Washington, DC, USA, IEEE Computer Society (2009) 1–6
10. Grammel, B., Kastenholz, S., Voigt, K.: Model matching for trace link generation in model-driven software development. Springer (2012)
11. Alanen, M., Porres, I.: Difference and union of models. Springer (2003)
12. Reddy, R., France, R., Ghosh, S., Fleurey, F., Baudry, B.: Model composition-a signature-based approach. In: Aspect Oriented Modeling (AOM) Workshop. (2005)
13. Cho, H., Gray, J., Syriani, E.: Creating visual domain-specific modeling languages from end-user demonstration. In: Modeling in Software Engineering (MISE), 2012 ICSE Workshop on, IEEE (2012) 22–28
14. Sánchez-Cuadrado, J., De Lara, J., Guerra, E.: Bottom-up meta-modelling: An interactive approach. Springer (2012)
15. Javed, F., Mernik, M., Gray, J., Bryant, B.R.: Mars: A metamodel recovery system using grammar inference. Information and Software Technology **50**(9) (2008) 948–968
16. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon object language (eol). In: Model Driven Architecture–Foundations and Applications, Springer (2006) 128–142
17. Jiawei, H., Kamber, M.: Data mining: concepts and techniques. San Francisco, CA, itd: Morgan Kaufmann **5** (2001)
18. Mitchell, T.M.: Machine learning. 1997. Burr Ridge, IL: McGraw Hill **45** (1997)
19. Williams, J.R., Zolotas, A., Matragkas, N.D., Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: What do metamodels really look like? EESSMOD@ MoDELS **1078** (2013) 55–60
20. Williams, J.R., Paige, R.F., Kolovos, D.S., Polack, F.A.: Search-based model driven engineering. Technical report, Technical Report YCS-2012-475, Department of Computer Science, University of York (2012)
21. Powers, D.: Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation (tech. rep.). Adelaide, Australia (2007)
22. Meyer, B.: Object-oriented software construction. Volume 2. Prentice hall New York (1988)
23. Zolotas, A., Kolovos, D.S., Matragkas, N., Paige, R.F.: Assigning semantics to graphical concrete syntaxes. In: XM 2014–Extreme Modeling Workshop. 12