Software & Systems Modeling
https://doi.org/10.1007/s10270-018-0658-5

SPECIAL SECTION PAPER

Type inference in flexible model-driven engineering using classification algorithms

Athanasios Zolotas[1] · Nicholas Matragkas[2] · Sam Devlin[1] · Dimitrios S. Kolovos[1] · Richard F. Paige[1]

Received: 17 June 2016 / Revised: 27 July 2017 / Accepted: 11 January 2018

Abstract
Flexible or bottom-up model-driven engineering (MDE) is an emerging approach to domain and systems modelling. Domain experts, who have detailed domain knowledge, typically lack the technical expertise to transfer this knowledge using traditional MDE tools. Flexible MDE approaches tackle this challenge by promoting the use of simple drawing tools to increase the involvement of domain experts in the language definition process. In such approaches, no metamodel is created upfront, but instead the process starts with the definition of example models that will be used to infer the metamodel. Pre-defined metamodels created by MDE experts may miss important concepts of the domain and thus restrict their expressiveness. However, the lack of a metamodel, that encodes the semantics of conforming models has some drawbacks, among others that of having models with elements that are unintentionally left untyped. In this paper, we propose the use of classification algorithms to help with the inference of such untyped elements. We evaluate the proposed approach in a number of random generated example models from various domains. The correct type prediction varies from 23 to 100% depending on the domain, the proportion of elements that were left untyped and the prediction algorithm used.

Keywords Model-driven engineering · Flexible model-driven engineering · Bottom-up metamodelling · Type inference · Classification and regression trees · Random forests

1 Introduction

In contrast to traditional rigorous MDE lifecycles where engineers start the DSL development process by creating a metamodel, in *flexible MDE approaches* engineers and domain experts start the process by defining example models in free-form drawing tools [11,18,23,38]. Flexible modelling is arguably more accessible to domain experts as the latter can use tools that they are already familiar with to express the concepts of the domain; the involvement of domain experts is widely argued to be important in the definition of high-quality DSLs [12,13,23,35]. In this fashion, modellers work without being restricted by a metamodel, which may be defined by MDE experts who are not necessarily domain experts. The sketched elements can have type annotations assigned to them and can be consumed by model management suites, which in turn can be used to determine whether the sketches are fit for purpose, e.g. by programmatically interrogating the sketches and building code generators for them. This process may lead to changes being made to the sketches incrementally. Using this approach, a potentially richer understanding of the domain is being incrementally developed, while concrete insights (e.g. type information) pertaining to the envisioned metamodel are discovered.

However, there are some drawbacks to what flexible MDE offers: the drawn elements do not conform to a specific pre-

Communicated by Prof. Alfonso Pierantonio, Jasmin Blanchette, Francis Bordeleau, Nikolai Kosmatov, Prof. Gabriele Taentzer, Prof. Manuel Wimmer.

✉ Athanasios Zolotas
thanos.zolotas@york.ac.uk

Nicholas Matragkas
n.matragkas@hull.ac.uk

Sam Devlin
sam.devlin@york.ac.uk

Dimitrios S. Kolovos
dimitris.kolovos@york.ac.uk

Richard F. Paige
richard.paige@york.ac.uk

[1] Computer Science Department, University of York, Deramore Lane, Heslington, York YO10 5GH, UK

[2] Computer Science Department, University of Hull, Hull HU6 7RX, UK

defined metamodel and as a result there is no guarantee that they will consistently obey the syntactic and semantic rules that a metamodel would impose. This can be revealed in the aforementioned type annotation process, where elements of a sketch have types proposed for them (by a domain expert), and thereafter attached to them using a sketching tool. This process can be problematic: in particular, different types of errors may arise in the annotation process.

1. *User input errors*: the types assigned to two elements of the same type are different due to spelling errors (e.g. Person vs. Prson)
2. *Changes*: due to better understanding of the domain following the incremental fashion of flexible MDE, a specific type may change to a new one. The new type should be assigned to all the elements manually. (Animal vs. Mammals)
3. *Inconsistencies*: different types might by assigned to elements that express the same concept due to the fact that multiple domain experts work on the same example models. (Doctor vs. Veterinarian)
4. *Omissions*: it becomes easier to overlook some elements and not assign them types, especially when example models become larger.

Such challenges need to be addressed to provide support for the transition from flexible to more rigorous (metamodel-based) modelling approaches.

In this paper, we propose a technique to tackle errors that belong to the fourth category given above: that of type omissions. In this category, sketch elements are overlooked and are not annotated with type information. There are at least two different approaches for solving this problem. The first is the application of a mechanism that will check for untyped nodes on the example models when these are created and request from the domain experts to provide their types; a constraint and language with model repairing capabilities [28] can be used to support this. However, such an approach may force users to take decision on types when they are not ready to do so. Also, this kind of approaches tend to reveal *all* omissions and inconsistencies at once, and so it can be difficult to find and repair specific problems. We consider this solution to be closer to the spirit of rigorous MDE (and constraint checking), rather than the spirit of flexible modelling.

The second approach is that of *type inference*: missing types can be inferred by analysing and calculating specific characteristics between the elements that are typed and those that are left untyped. One benefit of this approach is that users can avoid reapplying the same type to elements that are already defined in the diagram. In addition, elements can be created without having a type assigned to them, which can be calculated when it suits the domain expert, not the modelling tool.

This paper builds on top of the work presented in [43]. In [43], we proposed the use of a *classification algorithm*, specifically classification and regression trees (CART) [4], to calculate matches between typed and untyped elements, based on five characteristics of each of them. We demonstrated the approach using a flexible modelling technique based on GraphML, called *Muddles* [18]. The accuracy and limitations of the approach were evaluated via experiments on a number of randomly generated models.

In this extended version:

1. We test the use of a second classification mechanism, that of random forests (RF) [3] in an attempt to improve the accuracy of the prediction.
2. In addition to the initial dataset of random generated models, both CART and RF are now evaluated on a new set of random generated models that this time is injected with noise affecting 4 of the 5 variables used as input in the prediction algorithm.
3. Finally, the importance of each of the 5 variables in creating the decision tree(s) is also calculated and presented.

The rest of this paper is structured as follows. Section 2 includes a brief review of a specific flexible modelling approach, Muddles [18], which is based on GraphML. In Sect. 3, the proposed approach is described. In Sect. 4, details on the experiments that were carried out are presented. The results of running the experiments are given in Sect. 5, along with threats to experimental validity. In Sect. 6, related work in the field of type and metamodel inference is presented. In Sect. 7, we conclude the paper and outline plans for future work.

2 Background: Muddles

In this section, we present the Muddles [18] flexible modelling approach that is used in this work to illustrate the proposed approach to type inference.

2.1 Overview

In Muddles [18], yEd,[1] a GraphML-compliant drawing editor, is used to create the example models. Language engineers start by drawing the examples which they can then annotate with types and attributes. The relations (references and containments) between elements can be expressed using edges and group containers, respectively. Using a multipass model-to-model transformation, the annotated diagram is automatically transformed to an intermediate Muddle (the Muddle metamodel is given in Fig. 1), so it can be consumed

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

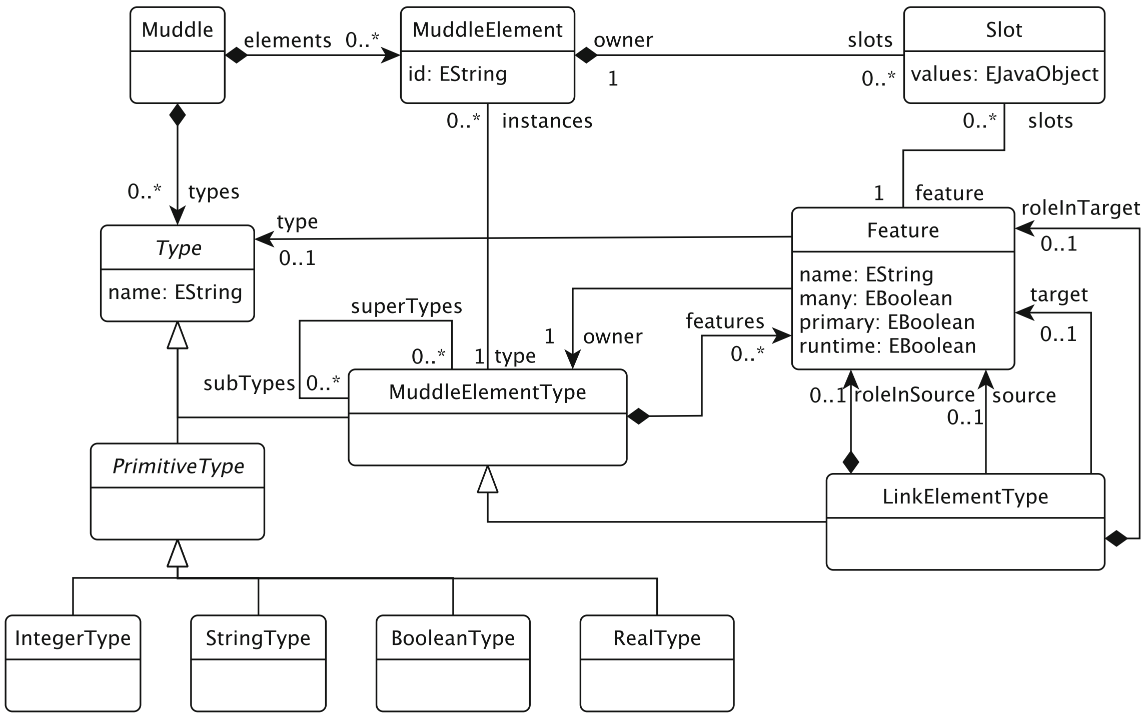


Fig. 1 The Muddle metamodel. (adapted from [18])

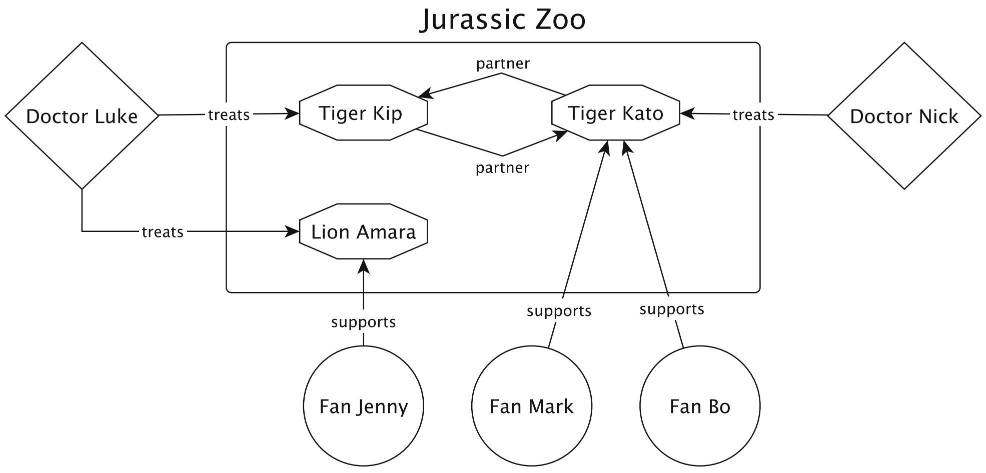


Fig. 2 An example Zoo diagram

by suites like the Epsilon platform [28] to perform model management operations (e.g. model-to-text transformation).

2.2 Example

We demonstrate “muddling” with an example, creating a language used to describe zoos. The language definition process starts with the creation of an example zoo diagram (see Fig. 2). Next, the elements are annotated with basic type information. For example, one can define the type of the diamond shape as “Doctor” and the type of the directed edges from “Doctor” to “Animal” nodes (hexagons) as instances of the “treats” relationship. In a muddle, the types are not bound to the shape; in the same drawing, a hexagon represents both elements of type “Tiger” and “Lion”. 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. The inheritance relationship can be denoted by using the “<” symbol in the type field. For example, for types “Lion” and “Tiger” this should be “Lion < Animal” and “Tiger < Animal”, respectively. More details about these properties are given in Table 1.

Model management programs use this type-related information to access and manipulate elements of the diagram.

Table 1 Element properties (based on Table 1 from [18])

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 *

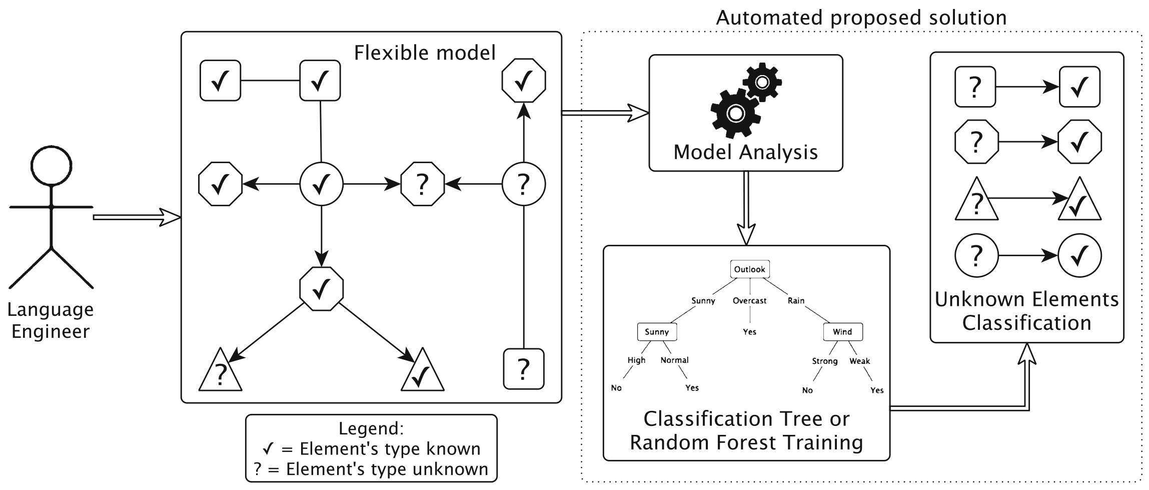


Fig. 3 An overview of the proposed approach (based on Fig. 3 from [43])

For example, if all the circled elements (typed as "Fan") have a string attribute called "name" assigned to them, then the Epsilon Object Language (EOL) [19] script in Listing 1 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).

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

Listing 1 EOL commands executed on the drawing

3 Type inference

In this section, we describe the proposed approach to type inference in flexible MDE approaches. An overview of the approach is given in Fig. 3. The source code for all the algorithms described in Sects. 3 and 4 along with detailed instructions can be found online.[2]

Language engineers initially construct a flexible model using a GraphML-compliant drawing tool, like yEd. Each element of this example model can then be annotated with types of the envisioned DSL. However, for the reasons mentioned in Sect. 1, some nodes may be left untyped. The annotated model is then automatically analysed to extract characteristics of interest, called *features*, which are presented in detail in the following section. 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.

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

Table 2 Signature features for nodes

Name of feature	Description
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 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 (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 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. As in Eclipse Modelling Framework and thus in Muddles one element can be contained in maximum 1 other node, this value is binary; 0: no parents, 1: has parent

3.1 Model analysis and feature selection

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 approach, we use a set of five features, presented in Table 2 which are the same as those used in our previous work [43]. These features were selected because they arguably measure structural and semantic characteristics of the models. As mentioned in Sect. 1, naming inconsistencies in types may appear especially if many domain experts work on the same models. Our feature selection was based on this assumption, and thus in this set of features we do not take into account string similarity between the various attributes of each element (i.e. names of attributes, names of references and names of containments). However, we need to highlight that we do not claim that the names should be totally ignored as they can carry useful information. Methodologies which base their similarity measurement on name matching could be combined with the approach we propose and possibly improve the prediction results.

The set of features for each node is called its *feature signature*. 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. Below we present some examples of feature signatures for the elements of the model illustrated in Fig. 4 which is an object-diagram-like representation of the muddle shown in Fig. 2.

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 sixth position of the signature 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, 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. Classification algorithms do not look for perfect matches but are trained to classify elements by using each time those and only 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 have different signatures.

A script that parses all the elements of the diagram was implemented. The parser constructs the signatures and stores them in a text file. The signatures that have types assigned to them are used to train the classification algorithm. The type of the rest is then predicted based on the outcome of this training.

3.2 Training and classification

Classification algorithms are a form of supervised machine learning for approximating functions mapping input features (e.g. our feature signature [3,3,1,0,1]) to a discrete output class (e.g. Tiger) from a finite set of possible values (e.g. [Tiger, Lion, Fan, Doctor, Zoo]. 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 the form [3,3,1,0,1,Tiger] a classification algorithm can learn to predict the class Tiger when given an unlabelled example such as [3,2,1,0,1].

Many classification algorithms exist, some of the most established being decision trees, random forests, support vector machines and neural networks [15]. For our previous work [43], 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. As part of the extensions presented in this paper, we experiment also with random forests (RF) [3], a

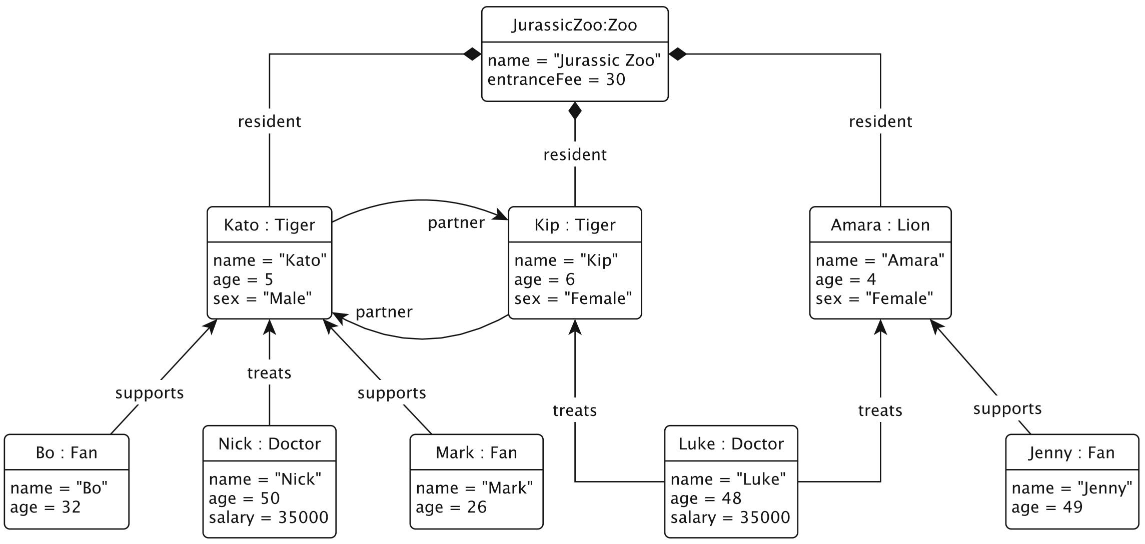


Fig. 4 Example model

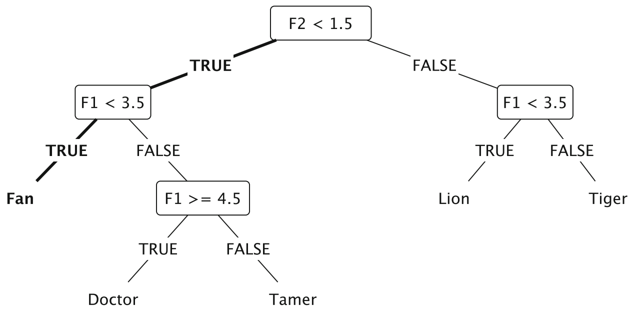


Fig. 5 Example decision tree

method that typically gives higher accuracy but less interpretable results [10].

Specifically, for decision trees, we used the *rpart* package (version 4.1-9)[3] that implements the functionality of CART [4] in R.[4] An example decision tree is illustrated in Fig. 5. Internal nodes represent conditions based on features (e.g. number of attributes, unique children), branches are labelled with "TRUE" or "FALSE" values for the condition of the parent node and leaf nodes represent the final classification given. 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 Fig. 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 Fig. 5).

CART generates a decision tree 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. The dataset is then split based on the values of the feature at the root node, and the process 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.

Additionally in this work, we used the *randomForest* R package (version 4.6-12)[5] to compare the performance of CART against a method that typically provides higher accuracy. An RF is an ensemble of multiple decision trees, each trained on a different set of training instances from the 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. More formally, the combined multiple weak hypotheses in an RF will typically outperform the single hypothesis generated by 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. While 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 consid-

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

[4] http://www.r-project.org/.

[5] https://cran.r-project.org/web/packages/randomForest/.

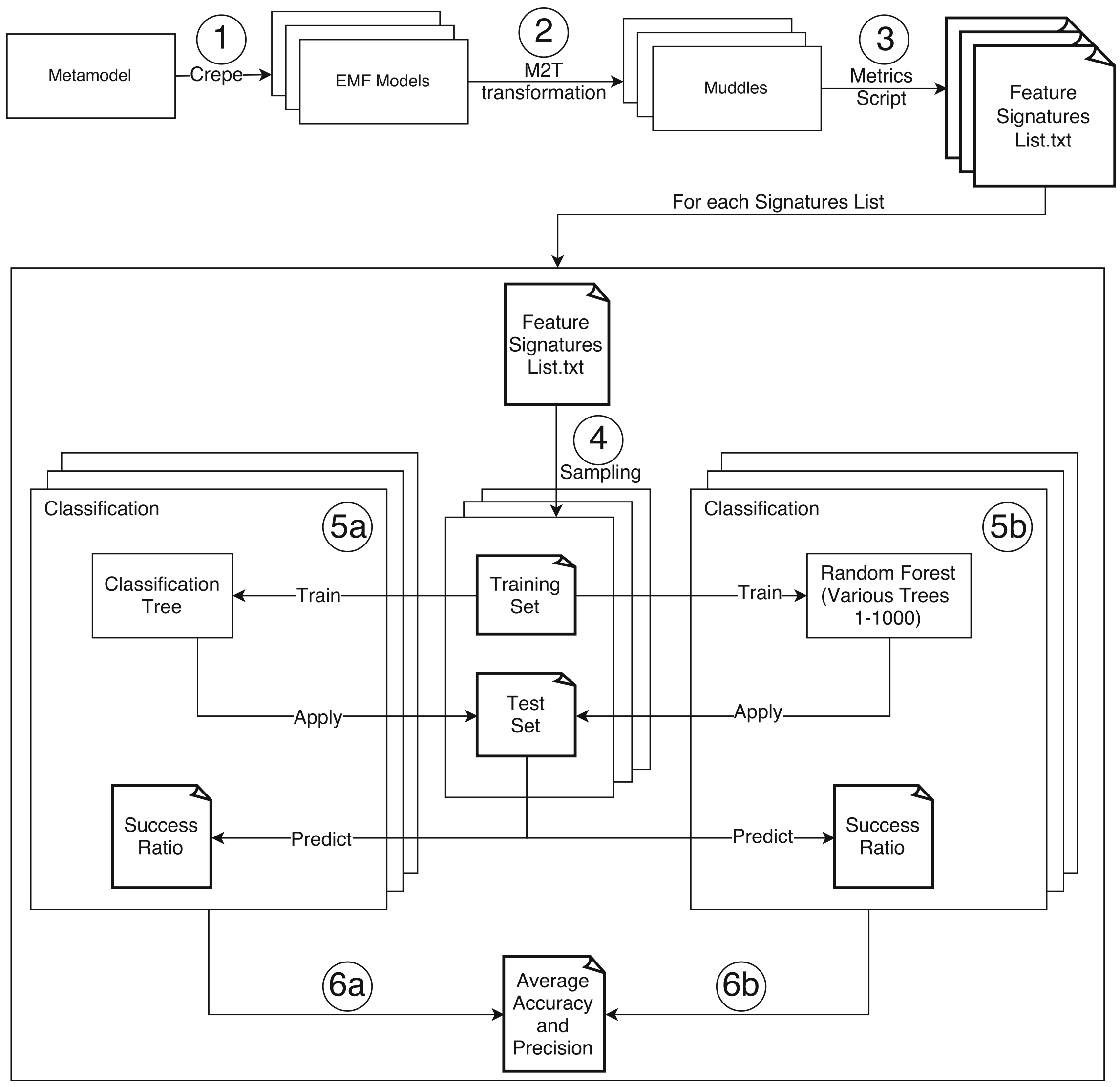


Fig. 6 The experimentation process

ered and the classifications of each combined to reach the final prediction of the model.

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 and RF algorithms. A trained decision tree or ensemble of trees is produced dependent on the algorithm used. These can then be used to classify (identify the type of) the untyped nodes using their feature signatures. To compare the relative performance of CART and RF, 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 (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 [27]. 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.

4 Experiment

In this section, the experiments ran to evaluate the proposed approach are presented. An overview of the experiment is shown in Fig. 6. Details about each step follow. All the scripts used in the approach are available at the paper's website.[6]

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

To evaluate our approach, we applied it to a number of randomly generated models, instances of publicly available metamodels that were collected as part of the work presented in [37]. The 10 metamodels selected are the same used in the evaluation of our previous work [43]. For each of these metamodels, we produced 10 random instances using the Crepe model generator tool [36] (step ① in Fig. 6) which 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.. *” and “0.. 1” references and containment relationships appeared in the metamodels. This might be a bias in the experiment as features 2–5 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. By injecting this type of noise, we create cases like those presented in the example feature signatures in Sect. 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 work 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 as “Sparse” set while the original is called as “Normal” in this work. 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 Sect. 5.6.

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).

These two steps (① and ②) could be skipped if there was a portfolio 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 two-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 feature signatures that works against the proposed approach.

After the generation of the muddles from the random models, we extract the feature signature of each node. We implemented a script that parses 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).

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 whose 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. As done in previous work [43], we picked 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. In order to conform to the standard 10-fold cross-validation in the domain of classifications algorithms, described in Sect. 3.2, we did this random sampling 10 times for each of the sampling rates for each example model, ending with 700 different couples of training and test sets for each metamodel in the experiment. The same process was done for both the “Normal” and “Sparse” set of models. It is important to highlight that each time the classification algorithm was trained using one training set and was tested using the coupled test set. After that, the algorithm was reset and trained/tested with the next couple of sets. In contrast with the previous work [43], we tried two different classification algorithms, CART (step $\textcircled{5a}$) and random forests (step $\textcircled{5b}$). In addition, 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 7 different values for the number of trees variable: 1, 5, 10, 50, 250, 500 and 1000.

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. In the next section, the results of the experiments are presented.

5 Results and discussion

In this section, the results of the experiment are presented. The raw results for all the experiments with plots and tables not included in the paper can be found at the paper’s website.

As described in Sect. 4, the experiment can be split into four sub-experiments based on 2 variables (see Table 3); the type of classification algorithm (CART vs. RF) and the density of the models (“Normal” vs. “Sparse”). In Sects. 5.1 and 5.2, the raw results of running the experiment on both the “Normal” and “Sparse” sets using the CART and RF algorithms will be given. In Sect. 5.3, a comparison of the results

Table 3 Experiments' IDs

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

for the CART versus RF and the "Normal" versus "Sparse" experiments will be presented. The results of the experiments on the importance of the variables used in the feature signatures will be discussed in Sect. 5.4 followed by a qualitative analysis and the threats to validity in the experiments.

Before going into the presentation and the discussion of the results, we give a summary of the models used as input in the experiments (see Table 4). 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 excludes 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 4. The average number of elements for the instances of each metamodel is also given for both sets.

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 booking system. We provide this muddle as an indication that the performance of the approach on the synthetic muddles from metamodels does not differ from the that of applying it to real muddles.

Finally, Table 5 presents the results of *randomly* assigning values to the untyped nodes. These values are provided for comparison with the results of our approach. To obtain these values for each model, we initially collected all the available types appearing in the model (i.e. the set of all the types of the typed nodes). We then visited each untyped node and assigned a type to it by picking one *randomly* from the aforementioned set. When all the untyped nodes in a model had a random type assigned, we compared the randomly assigned value with the correct one, which was already stored before the type deletion, to calculate the success ratio of the random assignment. Each field in Table 5 denotes the average success rate for each of the 100 models for each sampling rate. It is important to mention that the sampling rate is not important in the scenario of random type allocation. This is because, random allocation does not require any training and thus the information available in the model will not affect its performance as is the case with the CART and RF. As expected, the average accuracy across the same model is the same regardless the sampling rate. We include the results for all the sampling rates though to facilitate 1-to-1 comparison with the accuracy values of the classification algorithms.

5.1 Quantitative analysis for CART

As discussed in Sect. 4, 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 for 7 different values of the number of trees (49,000 runs in total). The results are summarised in Tables 6 and 7, respectively.

Table 4 Data summary table

Model name	#Types	Normal			Sparse		
		Min	Max	Average #elements in instances	Min	Max	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	–	–	–

Table 5 Results summary table for random assignment

Model name	#Types	Average accuracy for random assignment							Avg.
		30%	40%	50%	60%	70%	80%	90%	
Chess	2	–	0.49	0.49	0.51	0.52	0.49	0.53	0.507
Profesor	4	0.41	0.42	0.40	0.41	0.41	0.45	0.38	0.411
Zoo	5	0.20	0.20	0.19	0.19	0.20	0.21	0.18	0.197
Ant	6	0.15	0.16	0.16	0.16	0.16	0.16	0.19	0.162
Conference	6	0.22	0.20	0.18	0.19	0.21	0.20	0.22	0.202
Usecase	6	0.18	0.16	0.15	0.16	0.17	0.16	0.19	0.167
Bugzilla	7	0.14	0.14	0.13	0.14	0.15	0.13	0.14	0.138
BibTeX	8	0.13	0.12	0.12	0.13	0.12	0.12	0.10	0.122
Cobol	11	0.09	0.09	0.09	0.09	0.10	0.09	0.08	0.092
Wordpress	19	0.05	0.05	0.05	0.06	0.05	0.06	0.05	0.053
Avg.		0.18	0.20	0.20	0.20	0.21	0.21	0.21	

Table 6 Results summary table for N-CART

Model name	#Types	Average accuracy for different sampling rates (N-CART)							Avg.	Cor. 1
		30%	40%	50%	60%	70%	80%	90%		
Chess	2	–	1.00	1.00	1.00	1.00	1.00	1.00	1.000	–
Profesor	4	0.97	0.98	0.98	0.99	0.99	1.00	1.00	0.985	1
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
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
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		

Table 7 Results summary table for S-CART

Model name	#Types	Average accuracy for different sampling rates (S-CART)							Avg.	Cor. 1
		30%	40%	50%	60%	70%	80%	90%		
Chess	2	–	1.00	1.00	1.00	1.00	1.00	1.00	1.000	–
Conference	4	0.87	0.91	0.93	0.94	0.96	0.97	0.97	0.936	0.95
Profesor	4	0.85	0.90	0.93	0.95	0.95	0.96	0.94	0.926	0.82
Zoo	5	0.72	0.82	0.88	0.93	0.94	0.99	0.99	0.896	0.95
Ant	6	0.71	0.74	0.77	0.78	0.78	0.80	0.79	0.767	0.91
Usecase	6	0.74	0.79	0.81	0.82	0.81	0.83	0.83	0.804	0.86
Bugzilla	7	0.53	0.57	0.61	0.62	0.63	0.67	0.56	0.599	0.50
BibTeX	8	0.67	0.68	0.67	0.67	0.68	0.67	0.69	0.676	0.49
Cobol	11	0.44	0.51	0.54	0.59	**0.62**	0.64	0.68	0.574	0.99
Wordpress	19	0.41	0.51	0.57	0.60	0.64	0.66	0.69	0.583	0.96
Avg.		0.59	0.74	0.77	0.79	0.80	0.82	0.81		
Cor. 2		− 0.17	− 0.83	− 0.81	− 0.79	− 0.75	− 0.74	− 0.63		

In Tables 6 and 7, the average accuracy is given for all the models of each metamodel. The results are separated based on the sampling rate that was used each time. For instance, the highlighted value 0.71 in Table 6 indicates that for the Cobol metamodel, on average (between the 10 models and 10 sampling simulations), 71% of the missing types were successfully predicted, using 70% sampling rate. The respective value for the "Sparse" case was 62% (highlighted in Table 7)

Considering the raw values of both tables, the average accuracy varied from 53.7 to 100% for the "Normal" dataset and from 57.4 to 100% for the "Sparse" dataset. Comparing with the random allocation baseline provided in Table 5, we can see that in any model and regardless the sampling rate our approach performed significantly better. 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 85–100%). There are cases where the scores are higher than 97% even for samples as low as 30 or 40%. The same outcome is noticed at the smaller metamodels (fewer than 4 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:

Q1: How strong is the dependency between the sampling rate and the success score?

Q2: 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 *"Cor. 1"* and row *"Cor. 2"*, respectively.

As expected, the correlation coefficient values for Cor. 1 indicate a strong or perfect dependency for all the metamodels, except two (i.e. BibTeX and Usecase) for the "Normal" and 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 some samples in the "Normal" experiments and for all (except 30%) for the "Sparse" experiments. What it is of interest is that in both experiments the correlation is dropping steadily as the sampling rates are increasing. In some cases (above 70% sampling rate), in the "Normal" experiments the correlation stops being significant. One can extract the following 3 observations by checking these trends:

1. Fewer types lead to better results.
2. Fewer untyped nodes result to higher accuracy of the approach.
3. As the sampling rate increases, the effect of the second observation becomes less strong.

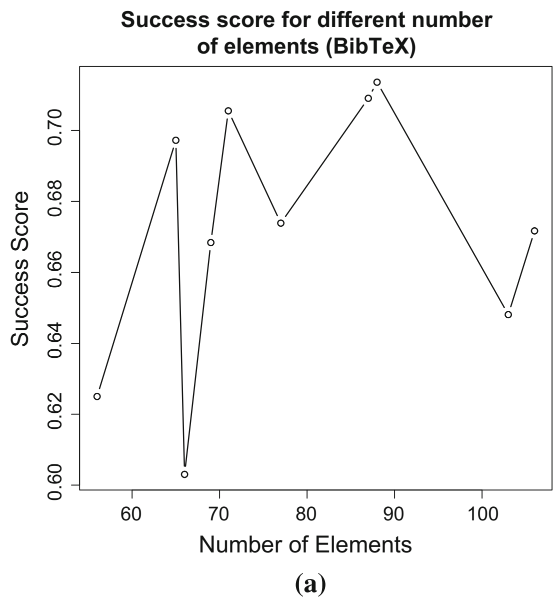
The approach presented here was tested (see Table 4) on models that have on average from 21 to 79 elements. These are "human-sized" models and not "super-sized" models of thousands of elements that would probably lead to better training with better results but are not considered realistic in scenarios where flexible MDE is used. We need to highlight that in our approach and the experiments, the learning algorithm is reset every time a new model is assessed, thus the results presented here are based on the algorithm trained each time on one "human-sized" model. It then starts from the beginning without any prior knowledge.

An experiment was conducted to explore if the size of the model affects the prediction accuracy. Two line graphs for the BibTeX and the Cobol metamodels are provided in Fig. 7 (the rest can be found on the paper's website).[7] These graphs show the accuracy for the 50% sampling of the 10 models of varying size for each of these two metamodels. As one can see, the accuracy is fluctuating and there are cases where biggest models score lower than smaller ones. Moreover, models of almost the same size have significantly different prediction accuracy. For example, in the Cobol metamodel (see Fig. 7b) two models of almost the same size (34 vs. 36 nodes) have 15% difference in the prediction accuracy (59–44%). What is interesting is that the smaller one have better accuracy. A possible explanation for this is the fact that the number of the elements is not the only variable that is changing when models grow in size. In reality, when new elements are added, more references (and containments) are created as well, which affects 4 out of the 5 variables in the experiment. Thus, when new nodes are added, the signatures of the already existing nodes are changing. As a result, the accuracy of the mechanism is affected not only by the addition of new elements (so, there is more training data for the approach) but from the fact that the features are changing as well, which is a very important (if not the most important) factor that affects the performance of the approach.

5.2 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 8 and 9, respectively. In the paper, we only include the results where the algorithm was trained using 1, 50 and 1000 trees. The data for all the values can be found at the paper's website.

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



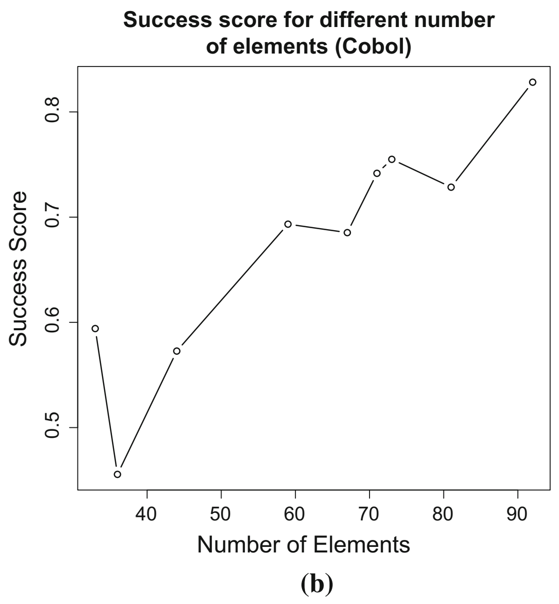


Fig. 7 Accuracy for different model sizes. **a** BibTeX—50% sampling, **b** Cobol—50% sampling

Table 8 Results summary table for N-RF

Model name	#Types	#Trees	Average accuracy for different sampling rates (N-RF)							Avg.	Cor. 1
			30%	40%	50%	60%	70%	80%	90%		
Chess	2	1	–	0.96	0.97	0.98	0.99	0.96	0.98	0.973	0.35
		50	–	1.00	1.00	1.00	1.00	1.00	1.00	1.000	–
		1000	–	1.00	1.00	1.00	1.00	1.00	1.00	1.000	–
Conference	4	1	0.71	0.78	0.78	0.81	0.82	0.82	0.85	0.796	0.93
		50	0.82	0.85	0.87	0.88	0.88	0.89	0.90	0.870	0.94
		1000	0.81	0.86	0.87	0.88	0.89	0.90	0.93	0.877	0.95
Profesor	4	1	0.85	0.87	0.91	0.90	0.91	0.92	0.93	0.899	0.92
		50	0.95	0.97	0.97	0.97	0.97	0.97	0.99	0.970	0.80
		1000	0.95	0.98	0.96	0.98	0.98	0.98	0.99	0.974	0.77
Zoo	5	1	0.61	0.64	0.64	0.66	0.73	0.75	0.73	0.680	0.93
		50	0.88	0.91	0.93	0.95	0.97	0.97	0.98	0.941	0.97
		1000	0.89	0.92	0.95	0.96	0.98	0.98	0.99	0.953	0.95
Ant	6	1	0.57	0.58	0.64	0.64	0.63	0.65	0.68	0.627	0.91
		50	0.67	0.68	0.71	0.71	0.72	0.70	0.72	0.701	0.79
		1000	0.67	0.69	0.71	0.71	0.72	0.71	0.74	0.707	0.91
Usecase	6	1	0.60	0.62	0.66	0.69	0.71	0.71	0.72	0.673	0.96
		50	0.74	0.75	0.76	0.78	0.78	0.76	0.75	0.760	0.35
		1000	0.75	0.75	0.77	0.79	0.78	0.77	0.76	0.767	0.41
Bugzilla	7	1	0.38	0.39	0.39	0.39	0.38	0.42	0.41	0.394	0.71
		50	0.45	0.43	0.46	0.46	0.43	0.44	0.45	0.446	− 0.06
		1000	0.46	0.43	0.47	0.47	0.44	0.45	0.45	0.453	− 0.10
BibTeX	8	1	0.56	0.57	0.58	0.57	0.57	0.60	0.57	0.574	0.49
		50	0.63	0.64	0.64	0.63	0.63	0.64	0.64	0.636	0.29
		1000	0.63	0.63	0.64	0.64	0.64	0.64	0.64	0.637	0.79
Cobol	11	1	0.43	0.49	0.53	0.55	0.55	0.61	0.60	0.537	0.95
		50	0.58	0.62	0.66	0.69	0.70	0.71	0.74	0.671	0.97
		1000	0.59	0.63	0.67	0.69	0.70	0.71	0.75	0.677	0.97

Table 8 continued

Model name	#Types	#Trees	Average accuracy for different sampling rates (N-RF)							Avg.	Cor. 1
			30%	40%	50%	60%	70%	80%	90%		
Wordpress	19	1	0.29	0.36	0.40	0.42	0.45	0.47	0.46	0.407	0.94
		50	0.45	0.55	0.62	0.66	0.69	0.71	0.76	0.634	0.97
		1000	0.48	0.56	0.62	0.67	0.70	0.71	0.77	0.644	0.98
Muddle	20	1	0.44	0.45	0.45	0.48	0.46	0.51	0.52	0.473	0.91
		50	0.48	0.54	0.55	0.56	0.58	0.60	0.58	0.556	0.89
		1000	0.50	0.56	0.55	0.58	0.59	0.60	0.56	0.563	0.70
Avg.			0.58	0.70	0.73	0.74	0.74	0.75	0.76		

Table 9 Results summary table for S-RF

Model name	#Types	#Trees	Average accuracy for different sampling rates (S-RF)							Avg.	Cor. 1
			30%	40%	50%	60%	70%	80%	90%		
Chess	2	1	–	0.98	0.98	0.98	0.96	0.98	0.95	0.972	− 0.68
		50	–	1.00	1.00	1.00	1.00	1.00	1.00	1.000	–
		1000	–	1.00	1.00	1.00	1.00	1.00	1.00	1.000	–
Conference	4	1	0.77	0.80	0.83	0.81	0.82	0.85	0.86	0.820	0.91
		50	0.83	0.85	0.87	0.86	0.87	0.88	0.88	0.863	0.90
		1000	0.84	0.86	0.87	0.86	0.87	0.89	0.87	0.866	0.77
Profesor	4	1	0.75	0.77	0.84	0.86	0.87	0.87	0.87	0.833	0.89
		50	0.87	0.90	0.92	0.94	0.94	0.95	0.95	0.924	0.93
		1000	0.88	0.90	0.92	0.94	0.94	0.95	0.95	0.926	0.94
Zoo	5	1	0.48	0.56	0.60	0.58	0.59	0.63	0.66	0.586	0.91
		50	0.64	0.70	0.77	0.78	0.79	0.85	0.88	0.773	0.97
		1000	0.65	0.71	0.77	0.79	0.80	0.85	0.88	0.779	0.98
Ant	6	1	0.52	0.54	0.60	0.62	0.63	0.64	0.67	0.603	0.96
		50	0.64	0.69	0.72	0.75	0.75	0.77	0.78	0.729	0.95
		1000	0.66	0.71	0.74	0.75	0.75	0.77	0.78	0.737	0.93
Usecase	6	1	0.59	0.63	0.67	0.69	0.71	0.75	0.76	0.686	0.99
		50	0.71	0.75	0.78	0.80	0.80	0.84	0.82	0.786	0.93
		1000	0.72	0.76	0.78	0.80	0.80	0.82	0.83	0.787	0.96
Bugzilla	7	1	0.47	0.48	0.53	0.53	0.53	0.58	0.48	0.514	0.45
		50	0.55	0.57	0.60	0.60	0.59	0.65	0.55	0.587	0.33
		1000	0.55	0.57	0.59	0.61	0.59	0.64	0.55	0.586	0.33
BibTeX	8	1	0.59	0.59	0.59	0.58	0.59	0.61	0.59	0.591	0.34
		50	0.63	0.64	0.64	0.64	0.63	0.63	0.64	0.636	0.00
		1000	0.63	0.65	0.64	0.64	0.63	0.64	0.65	0.640	0.28
Cobol	11	1	0.32	0.39	0.39	0.45	0.46	0.48	0.47	0.423	0.93
		50	0.45	0.51	0.53	0.58	0.61	0.63	0.63	0.563	0.97
		1000	0.45	0.51	0.55	0.58	0.62	0.65	0.64	0.571	0.97
Wordpress	19	1	0.23	0.30	0.33	0.34	0.40	0.38	0.43	0.344	0.96
		50	0.39	0.47	0.52	0.52	0.56	0.60	0.62	0.526	0.97
		1000	0.40	0.48	0.52	0.54	0.58	0.59	0.63	0.534	0.97
Avg.			0.54	0.68	0.70	0.71	0.72	0.75	0.74		

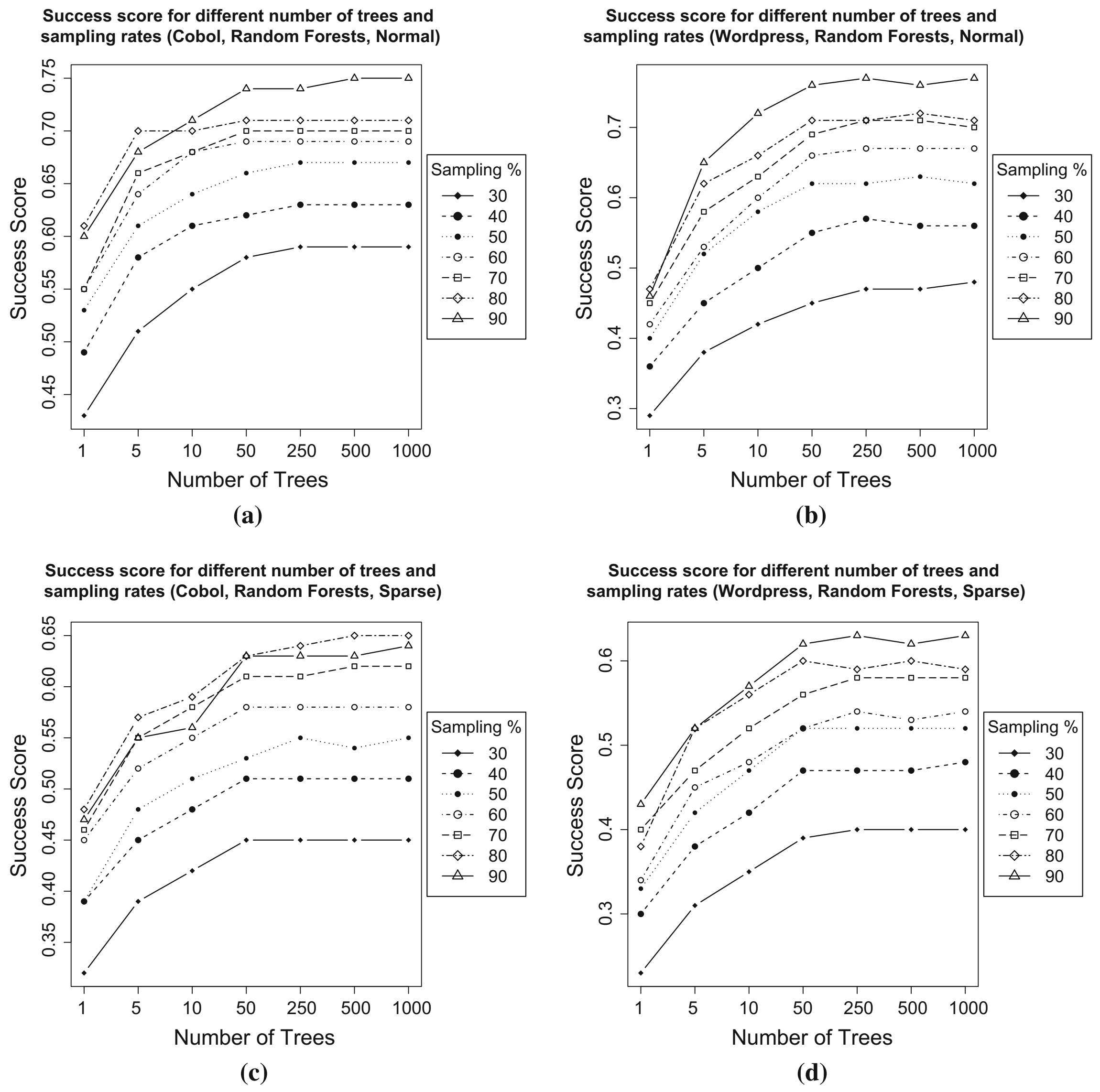


Fig. 8 Accuracy for different sampling rates and number of trees. **a** Cobol—Normal, **b** Wordpress—Normal, **c** Cobol—Sparse, **d** Wordpress—Sparse

The random forest 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 pattern 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 87.0 to 100.0% for models with less than 5 types (based on the 50 trees classification). The same values for the “Sparse” set are 77.3–100.0%. The lowest prediction value in the “Normal” set is for the Bugzilla metamodel, with 44.6% of the untyped nodes predicted correctly. The same value in the “Sparse” set is 58.7%. As with the CART approach, RF also outperforms the random allocation approach, the results of which are presented in Table 5.

Regarding the experiments related to the number of trees used and how this affects the prediction results it is clear from both the tables that the number of trees created as part of the random forest generation affects the accuracy: more trees lead to better accuracy. However, by analysing all the data for the 7 different tree values (1, 5, 10, 50, 250, 500, 1000) we identified that there is a point after which there is not significant improvement in the prediction. To make this more clear, we present the line graphs for the prediction accuracy based on the number of trees used, for 2 different models of the experiment (Cobol and Wordpress), for both the “Normal” set and the “Sparse” set.

Table 10 Accuracy difference trends between “Normal” and “Sparse” experiments

Model Name	# of “*” 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	↘	↘

As shown in Fig. 8, from 1 to 50 trees there is a rapid increase in the predictive ability of the RFs generated. After 50 trees, the accuracy does not improve any more but the computation needed to generate the ensemble 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.

5.3 Comparison

5.3.1 Normal versus Sparse

In this experiment, we adjusted the random model generator to produce less dense models by minimising the “*” or “0…1” references and containments instantiated. We summarise the results of the comparison in Table 10. In the table, the second column presents the number of “*” or “0…1” references in each model out of the total references. The third 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 (↗) in the “Sparse” set than the “Normal” set. In the same manner, the last column hosts the trend for the random forests equivalent.

As one can see from the table, in 9 cases the accuracy wasn’t affected at all, while in 8 others the accuracy dropped, sometimes significantly. There were three cases where the accuracy was increased. There are models which have all their relationships marked as “*” (e.g. Bugzilla, Usecase, Wordpress) and have different trends in their prediction scores (↗, ∼, ↘, 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” model generation also affects the results: the elements that are created only through one “*” relationship will be instantiated less times in the “Sparse” experiment. If these elements, on their turn, are responsible to instantiate other types that are only hosted by them, then the latter have significantly decreased chances in appearing in the set. For example, in a naive model with 3 types (Grandparent, Parent, Children) with 2 “*” relationships (Grandparent -*-> Parent -*-> Children), the “Sparse” set will have fewer “Parent” nodes and even less (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 lack of presence 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 the feature signatures of the “Grandparent” and “Parent” nodes were affected).

Finally, as mentioned in Sect. 3.2, 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 to pick any other from the rest 4 features (i.e. F1, F3, F4 or F5) to differentiate them. This way, the noise injection has no effect in the accuracy of the prediction mechanism.

5.3.2 CART versus 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 11 for the trends in the average scores for each metamodel). This was an unexpected outcome for the study, given that RF typically outperform CART and more generally that ensembles of classifiers typically outperform individual classifiers [10].

Table 11 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	$\nearrow$	$\sim$
Chess	$\sim$	$\sim$
Cobol	$\sim$	$\sim$
Conference	$\searrow$	$\searrow$
Profesor	$\sim$	$\sim$
Usecase	$\sim$	$\sim$
Wordpress	$\searrow$	$\searrow$
Zoo	$\sim$	$\searrow$

Our expectation 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 $\sqrt(p)$ where p is the number of features in the input. Given that our feature signature contains only 5 features, the package chose only 2 features to train each tree in the ensemble potentially harming the accuracy achievable by the resultant models. This default behaviour of RF does not allow it to outperform CART when datasets have restricted number of features as is the case in our approach. A possible solution that would help not only RF perform better but CART as well is the introduction of new, extra features on top of those five presented in this work. Plan for future work is described in Sect. 7. Furthermore, considering the high accuracy of CART on almost all metamodels (particularly those with 6 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 this method may be applied to, we conclude these results support our decision in the previous study that for this application to type inference CART is both sufficiently accurate and preferable to more complex classification algorithms due to its interpretable output.

5.4 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, to the best of our knowledge, this is the first time classification algorithms are used for type inference, and it is of interest to assess if any of the 5 proposed features 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 description of how the calculation is performed follows.

As discussed in Sect. 3.2, CART calculates the split performance of all the features (variables) available and selects the one that has the best goodness of split to place it as a condition on a node. The selected feature is called *primary* feature for the node [40]. However, sometimes there might be more than one features that would produce the identical splitting with the one selected to appear on the node (primary). These "clone" variables are called *surrogate* variables for the node [40]. The rpart package (responsible for classifying elements based on the CART algorithm in our approach) calculates the importance of each variable by accumulating the goodness of split measures of each variable whenever it participated as a primary or a surrogate variable in a node [33]. The final sums are then scaled to 100 to appear as percentages.

Regarding RFs [3], the importance of each variable is calculated by summing up the impurity decreases [24] for all the nodes that the variable participated. This sum is then divided by the total number of trees in the forest [24]. The Gini index is used as impurity decreases measure (also referred as the Mean Decrease Gini [24]) in the randomForest package used for classification in this work [22].

A summary of the results for the four different experiments is shown in the pie charts of Fig. 9. These values are the average importance of different variables for all the runs of the experiments expressed as percentages. We also include a table with the variable importance values of each feature for each metamodel in the N-CART experiments, in Table 12. The tables for the rest 3 experiments (S-CART, N-RF and S-RF) are available at the paper's website.

The pie charts suggest that in the 4 different experiments, the first feature (F1), that of *Number of Attributes*, is the one that is important 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 occupy feature 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 at all. Thus this specific feature value (i.e. F2 and F3 if there are no refer-

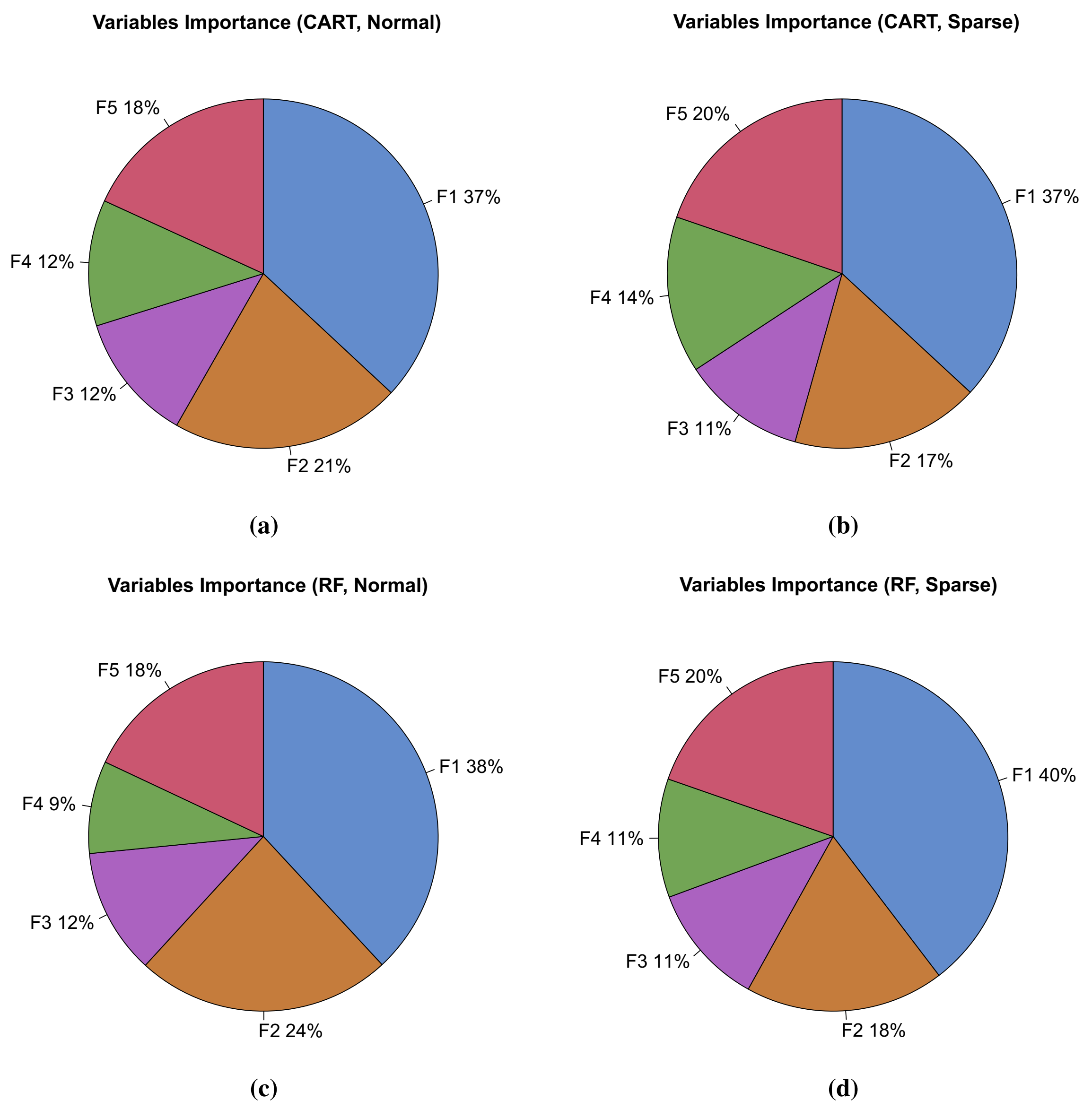


Fig. 9 Variables importance. **a** CART—Normal, **b** CART—Sparse, **c** RF—Normal, **d** RF—Sparse

Table 12 Variable importance table CART normal

Model name	F1	F2	F3	F4	F5
Ant	17.20	9.58	6.76	3.08	6.97
BibTeX	21.96	0.00	0.00	12.02	13.24
Bugzilla	10.69	5.37	0.00	4.81	5.38
Chess	6.12	0.00	0.00	0.92	6.10
Cobol	12.83	12.03	8.05	4.55	7.44
Conference	8.04	8.76	0.29	7.51	7.36
Profesor	7.64	1.99	1.84	0.66	6.43
Usecase	7.41	10.99	11.74	5.57	5.94
Wordpress	15.81	13.57	7.75	0.39	2.61
Zoo	17.29	9.85	3.85	0.00	0.00

ences, 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 12, in the “Chess” metamodel, which has no reference relationships, the values of F2 and F3 are 0.

5.5 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, we found out 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 same class "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 [26] (i.e. no differentiating point with the parent class as they have no extra attributes, references or containment relations assigned to them). As a result, the constructed feature signature is identical for all of the four types and thus the classification algorithm is unable to find a distinctive characteristic to split them into different classes/leaves.

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.

This behaviour is one of the reasons why the results are not getting close to the maximum possible value (that of 100% prediction accuracy) when the training set is high (90%): there are some cases like those described above where even the language engineer would not be able to identify the intended type of an untyped node as some types are not differentiated. A second reason that explains why the prediction is not maximised even when high training sets are used, which is also related with the first reason, is the number and the type of the features used in the proposed approach. These features are not able to find differences between nodes that have the same attributes and incoming/outgoing references or containments.

A way to address this problem and improve the prediction accuracy could be the introduction of other features, atop the five used in this study, which are calculated based on other characteristics that are not always the same in such situations. In [44], four features that are based on concrete syntax are proposed. Combining these features with the five proposed in this work might help in the direction of tackling the problems appearing in the aforementioned cases. In addition, including string similarity measurements in the prediction (like checking the name of the extra added attribute in the BibTeX example) will help, too. Finally, in a usual flexible modelling approach, a draft metamodel that explains the current concepts in the example models might already exists. This draft metamodel includes some constraints (e.g, multiplicities of references) that can be exploited to improve type inference in scenarios like the aforementioned where types have no differentiation point. Consider for example the following scenario. A draft metamodel which was inferred in one of the iterations of the Bugzilla metamodel development using a flexible MDE approach contained a reference from "Bug" to "Keyword" with multiplicity of "1". If in the example model, a "Bug" node is already connected with a "Keyword" node, then all the other remaining nodes connected with the same "Bug" node could not be of type "Keyword". In contrast with our approach that mixes the types "Keyword" and "Block" as they have no differentiation characteristics, a "Block" node that has been left untyped could not be incorrectly been predicted as of type "Keyword" following the new suggested approach. Such an approach that is based on Constraint Programming principles has been implemented in [42]. Plans in the direction of combining the above solutions with the proposed approach are discussed in Sect. 7.

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.

5.6 Threats to validity

The data used to evaluate the performance of the proposed approach were generated using a random model generator. The first issue of that is that we are using models which conform to a metamodel, and not real muddles. We wanted to evaluate the feasibility of the proposed approach for type inference first before carrying out more detailed experiments on user-created flexible models. We followed this approach of synthetic muddles creation for pragmatic reasons: firstly, we have a model generator that uses genetic algorithms to produce random models (no random muddle generator currently exists) and secondly, a library of example models created as part of a flexible MDE approach does not exist. It is not possible to be sure if the synthetic muddles created as part of this approach are representative examples of real muddles because flexible modelling is a relatively new technology which still positions itself in the MDE world. We do not believe that the use of models instead of muddles will have significant impact on the experimental results as the accuracy of our classification algorithm depends only on the features identified in Table 2; 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. In addition, we introduced noise in four out of five features to include structural inconsistencies in our data. The results of running the approach on this "Sparse" set revealed that the performance is slightly reduced.

A second issue related to the use of the 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 workaround 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. Plans for future work are described in Sect. 7.

In the experiment, 10 metamodels were used in total from which a number of muddles were generated. The metamodels were picked randomly from a zoo of 500 metamodels with no specific criterion other than that of describing a domain that most of the readers are familiar with. The number of types in these varied from 2 up to 19 as shown in Table 4. 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.

6 Related work

Flexible modelling is one of the methods proposed in the literature for tackling the *symmetry of ignorance* gap [7] in DSL development, i.e. the fact that domain experts do not usually possess language development expertise, and language engineers do not have domain knowledge.

One of the common activities of flexible modelling is the inference of a modelling language from a set of examples. This language can be textual or graphical. Roth et al. [30] propose an approach to the bottom-up development of textual DSLs. More particularly, 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.

Cuadrado et al. [6] propose an interactive and tool-supported approach to metamodel inference for DSLs. Their main goal is to actively engage the domain experts during the DSL development process. Domain experts can use either sketching tools such as Dia[8] or a dedicated textual notation to specify model fragments, which capture domain knowledge. Such fragments consist of untyped nodes and relations. Once a fragment is defined, a language engineer can enhance its semantics by annotating it with additional information (e.g. typing information). Semantically enhanced model fragments can then be consumed by the provided tool in order to infer a metamodel.

A similar approach is the *MLCBD* process [5]. This process consists of three phases. First, domain experts use shapes and connectors to define model examples. These examples are then annotated with domain-specific information, and finally these annotated examples guide the metamodel inference.

The two aforementioned approaches to metamodel inference rely on simple rectangular shapes and connectors between them for expressing model fragments. Kuhrmann [20] and Wuest et al. [39] propose more flexible approaches in which model fragments can be expressed in free-form shapes. Type annotations can be assigned to the various elements of the model fragment, and a metamodel can be inferred from the annotated example. The novelty of these approaches lies in sketch recognition algorithms, which assign typing information to new free-form shapes by matching them to annotated ones.

Metamodel inference is not used only in the context of flexible modelling. The *MARS* tool [14] supports metamodel inference from a set of models after migrating or losing their metamodel. This tool relies on a transformation engine, which converts models expressed in XMI to a domain-specific representation, and on an inference engine, which uses grammar inference techniques on the new representation in order to infer the metamodel.

The main goal of the aforementioned approaches is quite different from ours. Their goal is to infer a metamodel from a set of model fragments, which are assumed to be correct and complete. However, in our work we assume that model fragments in the context of flexible modelling can be incorrect and incomplete, since their correctness and completeness is not enforced by a modelling tool. Therefore, our approach is complementary to the aforementioned approaches. It can support the user during the definition of model fragments, and once correct and complete fragments are defined metamodels can be inferred automatically.

Work of partial modelling is also relevant to our work. In the literature there are different definitions of model partiality. In [8], 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 [8], in our work 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 [29,31].

Rabbi et al. [29] propose a diagrammatic approach to the completion of partial models based on category theory. Similarly, in [31] 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 metamodel. The aim of both approaches is to provide model completion in order to reduce modelling effort in the same manner that code completion provided by programming language editors reduces

[8] http://projects.gnome.org/dia/.

coding effort. Moreover, these approaches rely on a metamodel and they are not directly applicable in the context of flexible modelling.

Antkiewicz et al. [1] propose an approach to partial model completion based on the *Clafer* [2] 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 and then an inference engine uses a metamodel and the initial set of examples in order to derive a complete model. Compared to this work, our approach does not rely on a metamodel. Moreover, our approach is more generic, since it does not depend on a dedicated modelling environment.

In the context of MDE, model examples are used not only for metamodel inference but also for other MDE-related activities. Faunes et al. [9] propose an approach for inferring metamodel well-formedness rules from sets of valid and invalid models. The rule inference is based on genetic programming, and the derived rules are in the form of OCL invariants.

Furthermore, model transformation by-example (MTBE) approaches (e.g. [17,21,32,34]) have been proposed for automatically deriving model transformation rules. These approaches rely on user-defined examples of input and output models, and the inference is based on various techniques such as metaheuristics, model comparison and induction. A literature survey, which summarises the research in this area, is provided in [16].

7 Conclusions and future work

In this paper, we built atop our previous work presented in [43] to support type inference in flexible MDE approaches, providing support for moving from partially typed example models to more complete ones. In our previous work, we used a single classification algorithm, CART. In this work, we used a second algorithm based on random forests to assess how this will affect the prediction accuracy. The results showed that CART has already maximised the prediction performance and the use of an algorithm that belongs to the same category does not improve the results. For the random forests algorithm, we used 7 different values for the number of trees that the algorithm is trained with, identifying a point (50 trees) after which the prediction accuracy reaches a plateau. In addition, in this work, we injected noise in 4 out of the 5 variables used by creating more “Sparse” example models. The results showed that this has an impact in some metamodels. Finally, we calculated the importance of each variable in both algorithms.

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, for instance, to infer types for an already typed model, which may potentially reveal poor or incorrect type assignments or misuses of the metamodel.

In the future, we plan to make use of additional features. Work in this direction was presented in [44] where 4 spatial/-graphical related features are used (i.e. colour of the node, width and height, shape). A user study in which domain experts will create real example models using a flexible MDE approach (e.g. Muddles) is of interest. This will also allow the combination of the four features based on spatial characteristics mentioned above with those presented in this work. This is not possible at this point as all the nodes of the synthetic muddles created as part of this work have exactly the same graphical characteristics (i.e. shape, colour and dimensions).

In addition, the names that the domain experts choose to assign to the semantic characteristics (e.g. types, references or attributes) could also be assessed to improve the predictions. We could in principle enumerate all known types/references/attributes, calculate the distance of the label to each other and then use all of these distances within the input features. However, this would significantly increase the dimensionality of the feature space, which would then likely decrease the accuracy of the predictive model due to the low number of sample data points. Another possible approach would be that of the direct use of string similarity algorithms where elements are matched based on the differences between their types, references or attributes. Initial work in this direction has been carried out in [41, Chapter 5] using a widely used similarity algorithm, called similarity flooding [25]. Results suggest that a combination with the approaches presented in this work is possible and could improve prediction results. As mentioned before, we base this work on the assumption that domain experts may use different naming conventions to express the same structural information. However, we could overcome this by assigning weights to the importance of name-matching feature: if the examples are generated by more than one domain experts then decrease the impact of the name matching in the prediction.

Acknowledgements 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. Antkiewicz, M., Bak, K., Czarnecki, K., Diskin, Z., Zayan, D., Wasowski, A.: Example-driven modeling using clafer. In: MDEBE@MoDELS, vol. 1104, pp. 32–41. CEUR-WS.org (2013)
2. Bak, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wasowski, A.: Clafer: unifying class and feature modeling. Softw. Syst. Model. **15**(3), 1–35 (2015)
3. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324
4. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and regression trees. CRC Press, Boca Raton (1984)
5. Cho, H., Gray, J., Syriani, E.: Creating visual domain-specific modeling languages from end-user demonstration. In: 2012 ICSE Workshop on Modeling in Software Engineering (MISE), pp. 22–28. IEEE (2012)
6. Cuadrado, J.S., de Lara, J., Guerra, E.: Bottom-up meta-modelling: an interactive approach. In: MODELS'12: ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, LNCS 7590, pp. 3–19. Springer, Berlin (2012)
7. Dillenbourg, P.: What do you mean by collaborative learning. Collab. Learn. Cogn. Comput. Approach. **1**, 1–15 (1999)
8. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: 34th International Conference on Software Engineering (ICSE), pp. 573–583. IEEE (2012)
9. Faunes, M., Cadavid, J., Baudry, B., Sahraoui, H., Combemale, B.: Automatically searching for metamodel well-formedness rules in examples and counter-examples. In: Model-Driven Engineering Languages and Systems, pp. 187–202. Springer, Berlin (2013)
10. Friedman, J., Hastie, T., Tibshirani, R.: The elements of statistical learning. Springer series in statistics, vol. 1, pp. 587–604. Springer, Berlin (2001)
11. 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, vol. 22 (2011)
12. Izquierdo, J.L.C., Cabot, J.: Community-driven language development. In: 2012 4th International Workshop on Modeling in Software Engineering (MISE), pp. 29–35. IEEE (2012)
13. Izquierdo, J.L.C., Cabot, J.: Enabling the collaborative definition of DSMLs. In: Advanced Information Systems Engineering, pp. 272–287. Springer, Berlin (2013)
14. Javed, F., Mernik, M., Gray, J., Bryant, B.R.: MARS: a metamodel recovery system using grammar inference. Inf. Softw. Technol. **50**(9), 948–968 (2008)
15. Jiawei, H., Kamber, M.: Data mining: concepts and techniques, vol. 5. Morgan Kaufmann, San Francisco (2001)
16. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Düsterhöft, A., Klettke, M., Schewe, K.-D. (eds.) Conceptual Modelling and Its Theoretical Foundations, pp. 197–215. Springer, Berlin (2012)
17. Kessentini, M., Sahraoui, H., Boukadoum, M., Omar, O.B.: Search-based model transformation by example. Softw. Syst. Model. **11**(2), 209–226 (2012)
18. Kolovos, D.S., Matragkas, N., Rodríguez, H.H., Paige, R.F.: Programmatic muddle management. In: XM 2013—Extreme Modeling Workshop (2013)
19. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon object language (EOL). In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture—Foundations and Applications, pp. 128–142. Springer, Berlin (2006)
20. Kuhrmann, M.: User assistance during domain-specific language design. In: FlexiTools Workshop (2011)
21. Langer, P., Wimmer, M., Kappel, G.: Model-to-model transformations by demonstration. In: Tratt, L., Gogolla, M. (eds.) Theory and Practice of Model Transformations, pp. 153–167. Springer, Berlin (2010)
22. Liaw, A., Wiener, M.: Randomforest: Breiman and Cutler's random forests for classification and regression. Version: 4.6-12. https://cran.r-project.org/web/packages/randomForest/index.html (2015)
23. López-Fernández, J.J., Cuadrado, J.S., Guerra, E., de Lara, J.: Example-driven meta-model development. Softw. Syst. Model. **14**(4), 1–25 (2013)
24. Louppe, G., Wehenkel, L., Sutera, A., Geurts, P.: Understanding variable importances in forests of randomized trees. In: Advances in Neural Information Processing Systems, pp. 431–439 Curran Associates, Inc. (2013). http://papers.nips.cc/paper/4928-understanding-variable-importances-in-forests-of-randomized-trees.pdf
25. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: Proceedings of the 18th International Conference on Data Engineering, 2002, pp. 117–128. IEEE (2002)
26. Meyer, B.: Object-Oriented Software Construction, vol. 2. Prentice Hall, New York (1988)
27. Mitchell, T.M.: Machine learning, vol. 45. McGraw Hill, Burr Ridge (1997)
28. 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: 14th IEEE International Conference on Engineering of Complex Computer Systems, pp. 162–171. IEEE (2009)
29. Rabbi, F., Lamo, Y., Yu, I.C., Kristensen, L.M., Michael, L.: A diagrammatic approach to model completion. In: 4th Workshop on the Analysis of Model Transformations (AMT)@ MODELS, vol. 15 (2015)
30. Roth, B., Jahn, M., Jablonski, S.: On the way of bottom-up designing textual domain-specific modelling languages. In: Proceedings of the ACM Workshop on Domain-Specific Modeling, pp. 51–56 (2013)
31. Sen, S., Baudry, B., Precup, D.: Partial model completion in model driven engineering using constraint logic programming. In: International Conference on the Applications of Declarative Programming. Citeseer (2007)
32. Strommer, M., Wimmer, M.: A framework for model transformation by-example: concepts and tool support. In: Paige, R.F., Meyer, B. (eds.) Objects, Components, Models and Patterns, pp. 372–391. Springer, Berlin (2008)
33. Therneau, T.M., Atkinson, E.J., et al.: An introduction to recursive partitioning using the rpart routines. Technical report Mayo Foundation (2015)
34. Varró, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: Proceedings of the 2007 ACM Symposium on Applied Computing, pp. 978–984. ACM (2007)
35. Volter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering—Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013). http://www.dslbook.org
36. 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)
37. 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**, 55–60 (2013)
38. Wüest, D., Seyff, N., Glinz, M.: Flexisketch: a mobile sketching tool for software modeling. In: Uhler, D., Mehta, K., Wong, J.L.

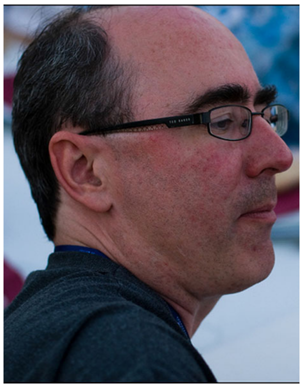
(eds.) Mobile Computing, Applications, and Services, pp. 225–244. Springer, Berlin (2013)
39. Wuest, D., Seyff, N., Glinz, M.: Semi-automatic generation of metamodels from model sketches. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 664–669. IEEE (2013)
40. Yohannes, Y., Webb, P.: Classification and regression trees, CART: a user manual for identifying indicators of vulnerability to famine and chronic food insecurity, vol. 3. International Food Policy Research Institute, Washington, D.C. (1999)
41. Zolotas, A.: Type inference in flexible model-driven engineering. Ph.D. Thesis, University of York, York, United Kingdom. http://etheses.whiterose.ac.uk/16380/1/main.pdf (2016)
42. Zolotas, A., Clariso, R., Matragkas, N., Kolovos, D.S., Paige, R.F.: Constraint programming for type inference in flexible model-driven engineering. Comput. Lang. Syst. Struct. (2016). https://doi.org/10.1016/j.cl.2016.12.002
43. Zolotas, A., Matragkas, N., Devlin, S., Kolovos, D., Paige, R.: Type inference in flexible model-driven engineering. In: Taentzer, G., Bordeleau, F. (eds.) Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 9153, pp. 75–91. Springer, Berlin (2015)
44. Zolotas, A., Matragkas, N., Devlin, S., Kolovos, D.S., Paige, R.F.: Type inference using concrete syntax properties in flexible model-driven engineering. In: 1st Flexible Model-Driven Engineering Workshop (2015)

Athanasios Zolotas is a Research Associate at the Computer Science department of University of York, UK. He is member of the Enterprise Systems research group. Athanasios received his EngD in Large-Scale Complex IT Systems from the University of York in 2017. His research interests are in model-driven engineering, safety critical systems and requirements engineering.

Nicholas Matragkas is a Lecturer of Software Engineering at the University of Hull, UK. He is a member of the Dependable Systems research group. Nicholas received his Ph.D. in Computer Science from the University of York in 2011. His current research interests include model-driven engineering, model management, software analytics and software testing.

Sam Devlin received an M.Eng. degree in Computer Systems and Software Engineering from the University of York, UK, in 2009. In 2013, he completed his Ph.D. on multi-agent reinforcement learning at the University of York and visited Oregon State University funded by a Santander International Connections Award. His research interests are focused on machine learning and artificial intelligence. He was a Research Associate from 2013–2015, working on data mining for collective game intelligence. He now holds a permanent academic role as a transitional fellow in the Digital Creativity Labs.

Dimitrios S. Kolovos is a Professor at the University of York. He has co-authored more than 150 scientific papers in international journals, conferences and workshops in the field of model-driven software engineering and has been an Eclipse Foundation committer leading the development of the Epsilon opensource project (eclipse.org/epsilon) since 2006, and the Emfatic project (eclipse.org/emfatic) since 2010.



Richard F. Paige is Professor of Enterprise Systems at the University of York, UK, where he leads the Enterprise Systems research group that specialises in Model-Driven Engineering. He has chaired numerous leading software engineering conferences and workshops, is on the editorial boards of Software and Systems Modeling, the Journal of Object Technology and Empirical Software Engineering. His research interests are in model management, formal methods, software processes, agile methods and safety critical systems.