Type Inference Using Concrete Syntax Properties in Flexible Model-Driven Engineering

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

[1] Department of Computer Science, University of York, York, UK
[2] Department of Computer Science, University of Hull, Hull, UK
Email: {amz502, sam.devlin, dimitris.kolovos, richard.paige}@york.ac.uk,
n.matragkas@hull.ac.uk

Abstract. In traditional Model-Driven Engineering (MDE) models are instantiated from metamodels. In contrast, in Flexible MDE, language engineers initially create example models of the envisioned metamodel. Due to the lack of a metamodel at the beginning, the example models may include errors like missing types, typos or the use of different types to express the same domain concepts. In previous work [1] an approach that uses semantic properties of the example models to infer the types of the elements that are left untyped was proposed. In this paper, we build on that approach by investigating how concrete syntax properties (like the shape or the color of the elements) of the example models can help in the direction of type inference. We evaluate the approach on an example model. The initial results suggest that on average 64% of the nodes are correctly identified.

1 Introduction

In traditional MDE the models instantiated using editors, conform to a predefined metamodel. In contrast, in Flexible MDE, language engineers use drawing editors, like those proposed in [2], [3] and [17], to express example models that will be used to infer the envisioned metamodel. These tools are used as free-form whiteboards on which the domain experts sketch elements that represent concepts and assign types to them. As there is no metamodel to specify the semantics, this process is error prone. Drawn elements may be left untyped, or the same concept may be represented by using two or more types. The first could happen either unintentionally (the engineer forgot to assign a type to the element) or intentionally (engineers leave repeatedly-expressed concepts untyped). The latter may occur either because two or more engineers are involved and thus different terminology may be used or, if the process is long-term, the engineer may have forgotten the type used in the past for a specific concept.

This paper addresses the challenges associated with identifying and managing omissions during type assignment in flexible modelling by using a variation of the *type inference* approach proposed in [1]: missing types are inferred by computing and analysing matches between untyped and typed elements that share the same

graphical characteristics. The difference between the approach proposed in [1] is that in this work we are looking for concrete syntax properties of the drawings and not at semantic relations. More specifically, in this approach, features that represent graphical characteristics of the nodes (i.e. the shape, color, width and height) are fed to a Classification and Regression Tree (CART) algorithm which predicts the types of the untyped nodes. We present the approach in detail, using an illustrative flexible modelling approach based on GraphML and the flexible modelling technique called *Muddles* [2]. We demonstrate the approach's accuracy via experiments run on an example flexible model.

This approach is based on the assumption that language engineers tend to use, to the extent possible, the same concrete syntax to express the same concept in a diagram. However, in order to explore the capabilities of the approach in cases where this assumption does not stand (or stands partially) we add noise to the example model by applying changes to the graphical properties of its nodes (e.g. randomly changing the shape of some nodes).

2 Related Work

In [4], rules that should be taken into consideration to construct the graphical syntax of languages is proposed. The author claims that the importance of graphical notation is a neglected issue so far and he adapts the *theory of communication* by Shannon and Weaver [5] to highlight that the effectiveness of the graphical syntax can be increased by choosing the most appropriate notation conventions of these that the human mind can process. In [6], Bertin identified a set of 8 visual variables that can encode and reveal information about the elements they represent or their relations in a graphical way.

In the field of bottom-up metamodelling, Cho et al. [7] propose an approach for an semi-automatic metamodel inference using example models. In [8], example models created by domain experts can be used to infer the metamodel. In [9], evolved models that do not conform to their metamodel are used to recover the metamodel. In [2], users, using a drawing tool, define example models which are then amenable to programmatic model management scripts.

Type inference is used in programming languages. The Hindley-Milner [10] [11] algorithm and its extension by Milner and Damas [12] are the most common used in this domain. Code statements are reduced to simpler 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 is a *matching* problem: elements that are "sufficiently similar" may have similar or identical types; a classification was published in [13]. A model matching technique is used in [14] to generate traceability links. The nodes of the two models are matched by checking their *name similarity*. In [15] each element's unique MOF identifier is used to calculate the difference and union of models that belong to the same ancestor model. In [16] manually generated signatures are used to match elements and perform model merging. The last three approaches are of limited flexibility as they depend

on names or persistent identifiers for inference. Finally, in Flexisketch [17] the authors adapt the algorithm proposed in [18] to find possible matches between the shapes of hand-drawn elements that appear on the same canvas.

This work builds on the approach presented in [1]. A CART algorithm is used to infer the types of nodes. However, the data fed to the classification algorithm consist of features that are related to semantic aspects of the example models (number of attributes, unique incoming and outgoing references and unique parents and children of each node). In this work, we propose the use of features that are related to the concrete syntax of the drawn example models.

3 Background: Muddles

In Muddles [2], a GraphML compliant drawing editor called yEd[3] is used to allow language engineers draw example models that are amenable to model management suites. The drawn elements are annotated with their types. Attributes can also be added by using the appropriate node properties input boxes. The drawing is automatically transformed to the intermediate Muddle model and can then be consumed by a model management suite like the Epsilon platform [19].

In this work, due to the fact that we are interested in the graphical properties of the drawings, like the shape, the color and the size of the drawn elements we use the extended version of the Muddles approach that was presented in [20] that extracts such information. A *muddling* example follows for better understanding.

3.1 Example

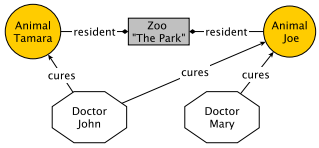

Fig. 1. Example

The language engineer is interested in creating a language for expressing zoos. The process starts by drawing an example zoo diagram (Fig. 1(b)). Next, diagram elements are annotated with basic type information. For instance, the type of both hexagon shapes is defined 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 hexagon can be of type *Doctor* while a different hexagon can be of type *Animal*. Types and properties of the types (e.g. attributes, multiplicity of edges) can also be specified. More details about these properties are presented in [2].

Model management programs can then be used to manage this diagram. For example, the following Epsilon Object Language (EOL) [21] script returns the

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

names of all the elements of Type *Animal* (the nodes typed as "Animal" have a String attribute named *name* assigned to them).

```
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 the type inference approach followed is presented (an overview is presented in Fig. 2). The engineer creates an example diagram of the envisioned DSL using a GraphML compliant tool (yEd). Some of the elements may be left untyped for the reasons discussed in Section 1. The mechanism proposed in this approach, analyses the drawing and collects graphical information from it which is then fed into a classification algorithm to classify the untyped elements.

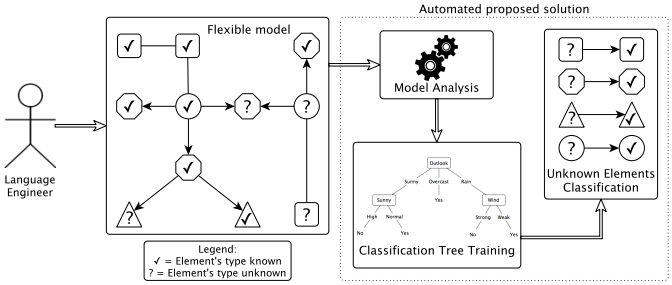

Fig. 2. An overview of the proposed approach (taken from [1])

4.1 Features Collection

The classification of the elements is based on characteristics that each element has. These are called *features*. We call the set of all the characteristics that are collected from each node as *feature signature*. In addition, the type of each node is attached to the last position of the feature signature. If the element is left untyped, then this place is left empty. In this approach we propose the use of 4 graphical characteristics as features for each node. The selected characteristics are presented in Table 1. Example signatures follow for better understanding.

Table 1. Signature features for nodes

Name of Feature	**Description**
Shape	The shape of the node (e.g. rectangle, ellipse, etc.).
Color	The color of the filling of the node in HEX (e.g. #FFCC00, etc.).
Width	The width of node expressed in pixels.
Height	The height of the node expressed in pixels.

For example, the feature signature for the node named "Animal Tamara" and annotated with the type "Animal" is [ellipse, #FFCC00, 114, 112, Animal]. The first value is the shape of the node (ellipse) while the second is the color of the filling. The third and fourth features are the width and the height. The last value is the annotated type for this element. Similarly, the feature signature for

the node named "Animal Joe" is [ellipse, #FFCC00, 107, 105, Animal]. From these two signatures one can see that elements of the same type may or may not have the same signatures. This justifies the selection of a classification algorithm and not a simple matching algorithm, as classification algorithms do not only look for perfect matches but can also classify the items by picking each time the features that are more important in the set that they are trained on.

A script that parses the diagram and constructs the feature signature for the nodes was created. The signatures are stored in a file that is fed to the CART.

4.2 Classification

Classification algorithms are a supervised machine learning method for finding functions mapping input features to a discrete output class from a finite set of possible values. They require a dataset to train on, after which they can generalise from the previous examples given in the training set to predict the class of new unseen instances.

Many classification algorithms exist, some of the most established being decision trees, random forests, support vector machines and neural networks [22]. For this work we chose to use decision trees. Specifically, we used the rpart package (version 4.1-9)[4] that implements the functionality of (CART) [23] in R[5]. In practice other classification algorithms (i.e. support vector machines or neural networks) can often have higher accuracy, but will produce a hypothesis in a form that is not as easily interpreted. Given the exploratory nature of this work, 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. For example, a possible decision tree for type inference is illustrated in Figure 3. Internal nodes represent features (e.g. Shape, Colour, etc.), each branch from a node is labelled with values of the feature and leaf nodes represent the final classification given. To classify a new instance, start at the root node of the tree 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. The predicted classification of the new instance is the value of that leaf node. For example, given the tree in Fig. 3, a new instance which shape is not an ellipse and its colour is different than white (#FFFFFF) classified as Zoo (path is highlighted in Fig. 3).

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 which can be used to classify (identify the type of) the untyped nodes using their feature signatures. 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

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

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 [24]. This approach generates k different splits of the data into training and test data sets and returns the mean accuracy generated from k repeats with each repeat using a unique split of the data.

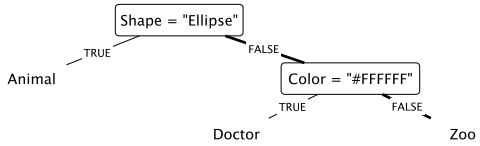

Fig. 3. Example Decision Tree

5 Experimentation Process

In this section the experimentation process used to evaluate the performance of the proposed approach is presented. An overview is given in Fig. 4[6].

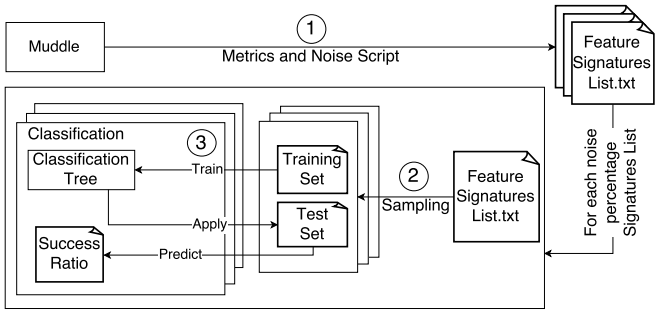

Fig. 4. The experimentation process

In order to test the proposed approach we applied the classification algorithm to a muddle. This muddle was created before commencing this research as part of a side project to express requirements for a web application. Our experience working with Muddles suggests that it is a fairly complicated example as it consists of more than 100 elements of 20 different types. This muddle was also one of the 11 models that were part of the experiment for the approach presented in [1]. A comparison with the 10 other models used in the experiment of the previous approach is not possible as these were automatically generated using mechanisms that are biased in the selection of the 4 features that we assess in this work: all the nodes were of the same shape, color and size.

In addition, we tested the resilience of the proposed approach to human error and the bias that our muddling habits may cause: we tend to use the same shape when we express a specific type. We need to highlight here, that in some cases all elements of the same type share same features but there are types where this is not the case. In contrast, some feature values (e.g. rectangle) were not used only for one type: elements of different types share common features in our

[6] The code, data, results and a guide on how to use the approach can be found in `http://www.zolotas.net/type-inference-graphical`

experimentation example. Arguably, this is the case with any other relatively large muddle as the number of available shapes and colors is in practice limited. Thus, in order to check if adhering to some basic conventions when drawing an example model is important for the accuracy of the prediction, we performed a second experiment by adding noise to some of the elements by explicitly changing their features. We did that gradually by altering randomly *one* feature of none (0%) up to all (100%) of the elements of the muddle using a step of 20% (0%, 20%, ..., 100%). 40% noise addition means that 40% of the nodes have *exactly one* feature (randomly selected) changed to something else (e.g. shape is changed from rectangle to ellipse). A step-by-step description of the experiment follows.

Initially a script collects the features from the muddle and places them in a list that includes the feature signatures for each element (step (1)). As the example has 105 nodes, there are 105 feature signatures in this list. This process is repeated 6 times; a new signatures list file is created for each of the noise levels. Next, each list is randomly separated into a training and a test set (step (2) in Figure 4). The training set contains the nodes for which in a realistic scenario the types are known while the test set contains those nodes that are left untyped. In order to reach unbiased results (due to an unlucky or lucky random sampling) we perform the random sampling 10 times for each file (10-Fold). It is also of interest to identify if the amount of knowledge that the algorithm has on each diagram is of importance to the success ratio. For that reason we use 7 different sampling rates; from 30% to 90%. For example, a 40% sampling rate means that 40 % of the nodes are thought to be of known type while the rest (60%) are the nodes for which the type is unknown. The generated couples of training and test sets (420 in total) are then fed to the CART algorithm (step (3)). The algorithm is trained on a training set and predicts the types of the elements of the coupling test set. The success ratio of the prediction is then calculated.

6 Results

Table 2 summarises the results for all the 420 runs. Each cell in the table contains the average accuracy of the classification for the 10 runs for the specific added-noise level and sampling rate. For instance, the highlighted value **0.64** indicates that on average, 64% of the missing types were successfully predicted for the 10 samples of the 20% added-noise model, using a 70% sampling rate.

The results suggest that the accuracy scores for the original (0% noise) model on average are similar to the success rates scores of the inference approach proposed in [1] (last row of Table 2). As expected, increasing the level of noise results in worse prediction scores. The same trend is also visible when decreasing the sampling rate. The correlation coefficients Corel. 1 and Corel.2, which definition follows, confirm that visual observation.

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

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

The values for Corel. 1 indicate a strong correlation for all the added-noise levels: prediction scores increase as training sets (the nodes with known types) become larger. The same behaviour was also observed in [1]. Regarding the second correlation (Corel. 2) we observe a perfect (negative) correlation between the number of nodes in a drawing that have altered features and the success score across all the sampling rates. This is evidence that following specific rules in the graphical syntax of the drawing increases the chances for correct type inference. By the term "specific rules" it is not implied that these rules should be strict. As discussed in previous sections, in the 0% added-noise example the authors use the same shapes to express the same concepts or in other cases the same color but not in a rigorous manner: same graphical properties are used in different concepts while the same concepts may have different graphical properties. We have also identified that the results related with the added noise experimentation (especially those of 80% and 100%) are influenced a lot by the randomness in picking the feature that each time will be altered. More specifically, if the random algorithm picked a lucky noise injection (changing in each node a feature that it is not distinctive for this type) then the results were better.

Table 2. Results summary table

	Average Success Ratio (Accuracy) for Different Sampling Rates								
Noise Level	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**Avg.**	**Corel. 1**
0%	0.55	0.58	0.61	0.63	0.67	0.71	0.71	0.637	0.99
20%	0.50	0.53	0.58	0.61	**0.64**	0.63	0.74	0.604	0.96
40%	0.47	0.53	0.52	0.56	0.57	0.62	0.63	0.557	0.97
60%	0.42	0.46	0.45	0.49	0.46	0.56	0.53	0.481	0.85
80%	0.35	0.38	0.44	0.43	0.50	0.44	0.56	0.443	0.89
100%	0.35	0.35	0.35	0.37	0.42	0.42	0.40	0.380	0.85
Corel. 2	-0.98	-0.98	-0.99	-0.99	-0.95	-0.98	-0.93	-	-
Muddle [1]	*0.55*	*0.56*	*0.59*	*0.65*	*0.59*	*0.67*	*0.64*	*0.607*	-

6.1 Threats to Validity

One example, created before commencing this research, was used to evaluate the approach. Although having one example may not be the best way to extract safe results we believe that it gives at least preliminary evidence that concrete syntax can be used to infer types of nodes in flexible modelling. A threat related to that which works against the approach is that the model consists of 20 different types. According to our experience with Muddles, this is a marginal one as engineers tend to use more rigorous editors as the models increase in size. The results in the experiments run in the previous approach [1] suggest that as the number of types increases the prediction accuracy decreases. In addition, the example model consisted of a number of highly repeated elements of the same type (e.g. nodes of type "MenuItem"). If these elements can be correctly identified, the fact that they participate a lot in the example leads to better success scores. However, a balancing fact is that there were also 5 types which appear less than 2 times, reducing the chances of having them predicted correctly in case all of

their instances end in the testing set (there will be no appearance in the training set so the algorithm doesn't know about the existence of this specific type).

7 Conclusions and Future Work

In this work we assessed whether the *concrete syntax* of flexible models can be used to infer the types of the elements that are left untyped using CART. Experiments suggest that on average 64% of the types were correctly identified. We also experimented with the intentional addition of noise in the diagram to check how this affects the prediction accuracy. A strong correlation between the percentage of altered nodes and the accuracy was identified providing evidence that this approach is more successful if it is used under the assumption that modellers tend to use, to the extent possible, the same graphical notation for elements of the same concept. We believe that this behaviour can be "unintentionally" replicated because of the "copy-paste" nature of muddling (e.g. create an animal node once and then copy & paste the node when you need it again). This way the same graphic notation is used for all the elements of the same type. It is important to highlight that if the type of the node was typed before the "copy-paste" event took place, which is not necessarily always the case, then the type is also transferred to the newly pasted nodes.

In the future, we plan to introduce and test additional features like font size and color, border size, orientation etc. In addition, in order to check if the prediction accuracy can be further increased, our intention is to combine the concrete syntax feature with the semantic related features proposed in [1]. Finally, we plan to run user studies to further evaluate the approach on more example models developed by language engineers.

Acknowledgments

This work was carried out in cooperation with Digital Lightspeed Solutions Ltd, and was part supported by the Engineering and Physical Sciences Research Council (EPSRC) through the Large Scale Complex IT Systems (LSCITS) initiative, and by the EU, through the MONDO FP7 STREP project (#611125).

References

1. 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. Volume 9153 of Lecture Notes in Computer Science. Springer International Publishing (2015) 75–91
2. Kolovos, D.S., Matragkas, N., Rodríguez, H.H., Paige, R.F.: Programmatic muddle management. XM 2013–Extreme Modeling Workshop (2013) 2
3. 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)

4. Moody, D.L.: The physics of notations: toward a scientific basis for constructing visual notations in software engineering. Software Engineering, IEEE Transactions on **35**(6) (2009) 756–779
5. Shannon, C.E., Weaver, W.: The mathematical theory of communication. (2002)
6. Bertin, J.: Semiology of graphics: diagrams, networks, maps. (1983)
7. 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
8. Sánchez-Cuadrado, J., De Lara, J., Guerra, E.: Bottom-up meta-modelling: An interactive approach. Springer (2012)
9. 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
10. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the american mathematical society (1969) 29–60
11. Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3) (1978) 348–375
12. 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
13. 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
14. Grammel, B., Kastenholz, S., Voigt, K.: Model matching for trace link generation in model-driven software development. Springer (2012)
15. Alanen, M., Porres, I.: Difference and union of models. Springer (2003)
16. Reddy, R., France, R., Ghosh, S., Fleurey, F., Baudry, B.: Model composition-a signature-based approach. In: Aspect Oriented Modeling (AOM) Workshop. (2005)
17. Wüest, D., Seyff, N., Glinz, M.: Flexisketch: A mobile sketching tool for software modeling. In: Mobile Computing, Applications, and Services. Springer (2013) 225–244
18. Coyette, A., Schimke, S., Vanderdonckt, J., Vielhauer, C.: Trainable sketch recognizer for graphical user interface design. In: Human-Computer Interaction–INTERACT 2007. Springer (2007) 124–135
19. 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
20. Zolotas, A., Kolovos, D.S., Matragkas, N., Paige, R.F.: Assigning semantics to graphical concrete syntaxes. In: XM 2014–Extreme Modeling Workshop. 12
21. 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
22. Jiawei, H., Kamber, M.: Data mining: concepts and techniques. San Francisco, CA, itd: Morgan Kaufmann **5** (2001)
23. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and regression trees. CRC press (1984)
24. Mitchell, T.M.: Machine learning. 1997. Burr Ridge, IL: McGraw Hill **45** (1997)