Constraint Programming for Type Inference in Flexible Model-Driven Engineering

Athanasios Zolotas[a,*], Robert Clarisó[b], Nicholas Matragkas[c], Dimitrios S. Kolovos[a], Richard F. Paige[a]

[a] *Computer Science Department, University of York, YO10 5GH, York, United Kingdom*
[b] *IT, Multimedia and Telecommunication Department, Universitat Oberta de Catalunya, Barcelona, Spain*
[c] *Computer Science Department, University of Hull, HU6 7RX, Hull, United Kingdom*

Abstract

Domain experts typically have detailed knowledge of the concepts that are used in their domain; however they often lack the technical skills needed to translate that knowledge into Model-Driven Engineering (MDE) idioms and technologies. Flexible or bottom-up modelling has been introduced to assist with the involvement of domain experts by promoting the use of simple drawing tools. In traditional MDE the engineering process starts with the definition of a metamodel which is used for the instantiation of models. In bottom-up MDE example models are defined at the beginning, letting the domain experts and language engineers focus on expressing the concepts rather than spending time on technical details of the metamodelling infrastructure. The metamodel is then created manually or inferred automatically. The flexibility that bottom-up MDE offers comes with the cost of having nodes in the example models left untyped. As a result, concepts that might be important for the definition of the domain will be ignored while the example models cannot be adequately re-used in future iterations of the language definition process. In this paper, we propose a novel approach that assists in the inference of the types of untyped model elements using Constraint Programming. We evaluate the proposed approach in a number of example models to identify the performance of the prediction mechanism and the benefits it offers. The reduction in the effort needed to complete the missing types reaches up to 91.45% compared to the scenario where the language engineers had to identify and complete the types without guidance.

Keywords: Flexible modelling, Bottom-up modelling, Type inference, Constraint programming, Example-driven modelling

*Corresponding author
Email addresses: amz502@york.ac.uk (Athanasios Zolotas), rclariso@uoc.edu (Robert Clarisó), n.matragkas@hull.ac.uk (Nicholas Matragkas), dimitris.kolovos@york.ac.uk (Dimitrios S. Kolovos), richard.paige@york.ac.uk (Richard F. Paige)

1. Introduction

Conventional Domain-Specific Languages (DSL) definition processes start with the creation of a metamodel which is then used to instantiate models and guide the development of editors and other artefacts such as model-to-model and model-to-text transformations. Such a process implies expertise in metamodelling, and in relevant technologies. While this may be an easy or at least understandable process for MDE experts, this is not always the case with domain experts [1] who are more familiar with tools like simple drawing editors [2]. However, the involvement of domain experts is important in the definition of high quality and well-defined DSLs that cover all the needed aspects of a domain. To address the aforementioned issue, flexible modelling approaches have been proposed in the literature (e.g. [3, 4, 5, 1]). Such approaches are based on sketching tools and do not require the definition of a metamodel during the initial phases of language engineering.

More specifically, in flexible (or bottom-up) MDE, the process starts with the definition of example models [1, 6, 7]. These example models help language engineers to better understand the concepts of the envisioned DSL and can be used to infer *draft metamodels* manually or (semi-)automatically which eventually lead in the definition of the final metamodel. In this fashion, a richer understanding of the domain can be developed *incrementally*, while concrete insights (e.g., type information) pertaining to the envisioned metamodel are discovered. Figure 1 depicts the stages taking place in a typical flexible MDE process as this is interpreted by studying different flexible MDE approaches in the literature (e.g. [1, 2, 8]).

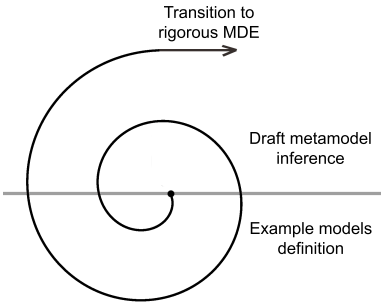


Figure 1: Stages of a typical flexible MDE approach.

The sketching tools used in such processes, allow the quick definition of exemplar models sacrificing the formality that model editors, which are based on a rigorously-defined metamodels, offer. In addition, drawing tools do not require MDE-specific expertise. The elements (nodes and edges) of these flexible example models can have type annotations assigned to them to describe the domain concept they represent and can also be amenable to programmatic model management using MDE suites like Epsilon [9].

On the other hand, since sketching tools cannot enforce syntactic and semantic correctness rules, flexible models are prone to various types of errors [10]:

1. *User input errors:* elements that should share the same type, have different types assigned to them by mistake or as a result of a typo (e.g. An**i**mal vs. Anmal).
2. *Changes due to evolution:* elements representing concepts that have evolved during the domain exploration process, do not have their types updated (e.g. Animal vs. Herbivores and Carnivores).
3. *Inconsistencies due to collaboration:* when multiple domain experts collaborate in the definition of the models, multiple types representing the same concept can be used (e.g. Doctor vs. Veterinarian).
4. *Omissions:* elements can be left untyped especially when models become large as it is easier to overlook some of the elements.

The trade-off between formality and flexibility can possibly result in a better domain understanding by language engineers, and eventually to a higher quality language. Bottom-up metamodelling is an iterative process, since different versions of metamodels and exemplar models are continuously evolved in an interleaved manner until the final version of the metamodel is obtained [1].

The contribution of this paper is a tool-supported approach for eliminating type omission errors (see error labelled "Omissions" above) from flexible models. Currently such errors are eliminated manually by language engineers by selecting an appropriate type from a set of possible types. That means, that if in the draft metamodel there are N different concrete types, the language engineer has N options for each untyped element. However, this approach does not benefit from information that exists in the draft metamodel and that could possibly help in reducing the number of possible types for a specific node. For example, if in the metamodel it is defined that nodes of type "A" can only be connected with nodes of type "B" then if an untyped node is connected with a node of type "B", it can be inferred that the type of the missing node is type "A". An advantage of that second approach is that the search space for the possible types suggested to the language engineer can be reduced from N to M, where $M \in [1,N]$, from which the engineer has to select the *correct* one. In this work, the intended meaning of the "correct type" is the type that the engineer envisioned when drawing the specific element in the example model.

Our proposed approach does not require a metamodel that is refined to follow the best practices or patterns proposed for metamodel development. The *only requirement* is that the metamodel must include all the types and the relationships that are present in the example models. The term "draft" which characterises the metamodels needed as part of this approach implies firstly the optional need of having metamodelling best practices applied to the metamodel. Secondly, it can be "draft" in terms of not being a final one that covers all the concepts of the domain, but an intermediate one that covers a subset of the concepts, as soon as these are the only concepts appearing in the example models. This draft metamodel, following the iterative flexible MDE principles,

should reach a final version that covers all the envisioned concepts and relationships. Related to that, it is necessary to highlight that our proposed approach focuses only on the inference of the types of the untyped nodes in the example models created (or evolved) as part of a flexible MDE approach (see "Example models definition" phase in Figure 1). The way that the draft metamodel is inferred (or evolves) is outside the scope of this work and it is an interesting area of future research; the draft metamodel should fulfil the aforementioned requirement, though.

In previous work [10, 11], an approach to tackle the error of type omissions was proposed. The algorithm, that is based on Classification and Regression Trees (CART), is trained on the elements of the example models only, trying to identify similarities based on different sets of criteria and predict the types of untyped nodes without requiring the existence of a metamodel. This work contributes a novel approach in addressing the challenges associated with type omissions, but this time taking into account a draft metamodel constructed by language engineers, using constraint programming principles for suggesting the possible types. More specifically, the syntax and the constraints defined in the draft metamodel are automatically transformed to a set of facts and rules that are then applied to the example models to reduce the number of possible types of untyped nodes. Beyond the requirement for a metamodel, another important difference with the previous approach is the fact that in this work, the correct type for each node is **always** included in the set of suggested types. In the previous work, the suggested type is not guaranteed to be the correct one. A trade-off for that, is the fact that there might be more than one possible types suggested for each node while in the previous work there was always one type returned, not always the correct one though.

The rest of this paper is structured as follows. Section 2 includes a brief review of a specific flexible modelling approach, Muddles [3], which is based on GraphML and is used as a proof of concept for the proposed approach and the experimentation. In Section 3 the approach is presented. In Section 4, an empirical evaluation of the performance of the proposed approach, is conducted. The results of running the experiments are discussed in Section 5, along with threats to experimental validity.[1] In Section 6, related work in the field of flexible modelling, type inference and constraint programming in MDE is presented. In Section 7 we conclude the paper and outline plans for future work.

2. Background

In our work we use the Muddles approach [3] for sketching model examples. The Muddles approach proposes the use of GraphML compliant tools such as yEd[2] for the definition of model sketches. More particularly, the domain engi-

[1] The algorithms, the experimentation data and the results along with instructions can be found at `http://www.zolotas.net/type-inference-cp/`

[2] The yEd editor can be downloaded from `http://www.yworks.com/en/products_yed_about.html`

neer can use simple drawing editors to express example models of an envisioned Domain-Specific Language (DSL). Using simple drawing editors to express the domain concepts increases the participation of the domain experts as it requires no (or minimum) training while it allows them working with tools that they are already familiar with [1, 2]. The drawing produced, called a *muddle*, can be consumed by model management suites, like the Epsilon platform [9], to support MDE activities (e.g. queries, transformations) enabling the language engineer to experiment with the models, gain better understanding and decide they are fit for purpose.

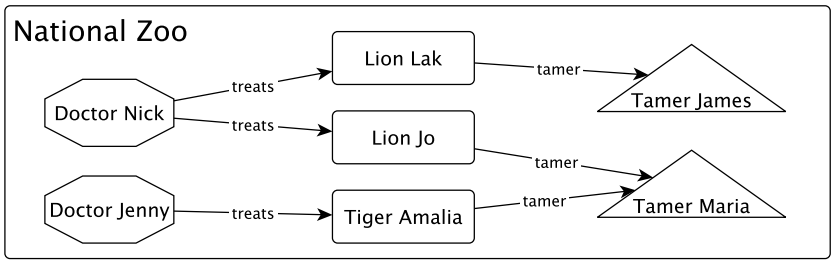


Figure 2: An example Zoo diagram

An example of such a muddle is shown in Figure 2. In this example, the intention of the language engineer is the creation of a simple DSL for defining Zoos. The process starts with the definition of example models of this envisioned DSL by the domain experts. The language engineer can then annotate types and typing information (e.g. properties) for each node. This drawing can be then consumed by model management programs that can be written in parallel to check if it fulfils the needs of the engineer and expose more features of the language, if any. In contrast to related work [1], shapes and other graphical characteristics of each node are not bound to types. For example, in the drawing of Figure 2, elements of type "Lion" are expressed using rounded rectangles, the same shape that is used to define elements of type "Tiger". Moreover, elements of the same type can be expressed using different shapes, for instance two different elements of type "Lion" can be expressed using the rounded rectangle for one of them and a circle for the second. By doing so, the domain expert is not constrained by a concrete syntax and can use any shape and arrow available from the tool's palette to express herself freely.

Types and type-related information for each element are defined using custom GraphML properties[3]. This is an extensibility mechanism provided by GraphML to support attaching arbitrary key-value information to graph elements. A short description and examples for each custom property is given below. More details on these properties can be found in [3].

[3]YEd's manual on custom properties can be found in `http://yed.yworks.com/support/manual/properties.html/`

Table 1: Element properties (based on Table taken from [3])

Extension	**For**	**Description**	**Example**
Type	Node, Edge	The type of the element	Lion, Doctor < Person (< denotes the extend relationship)
Properties	Node, Edge	Descriptors and values for primitive attributes of nodes/edges	String name = Jenny, Integer age = 25
Default	Node, Edge	The label of the node/edge	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	The role of the edge in its source node	patient 0..5, tamer 1
Role in target	Edge	The role of the edge in its target node	carer *, employee *

It is important to mention here that the proposed approach is not bounded with the Muddles approach, which is only used as proof of concept and for experimentation purposes, but can be adapted to and used in principle with any other flexible modelling approach which fulfils the following minimal set of requirements:

- allows users to assign types to nodes and provides a mechanism to extract these *types*
- provides a mechanism to allow the extraction of the *source and target nodes* of the *edges* in the example models

3. Proposed Type Inference Approach

3.1. Overview of the Approach

In this section the proposed approach for type inference in flexible MDE is presented. An overview is given in Figure 3.

The process starts with the language engineers having example models drawn using a flexible modelling approach (i.e. muddles in this scenario) to facilitate the process of defining the DSL they are interested in (step (1)). This may involve continuous changes to the example models, after which enough knowledge is acquired for the production of a first draft version of the DSL's metamodel (step (2)). The example models due to the lack of editors that are based on pre-defined metamodels (see Section 1) may have nodes that are left untyped. At this stage, the metamodel might be a partial one, that only describes the

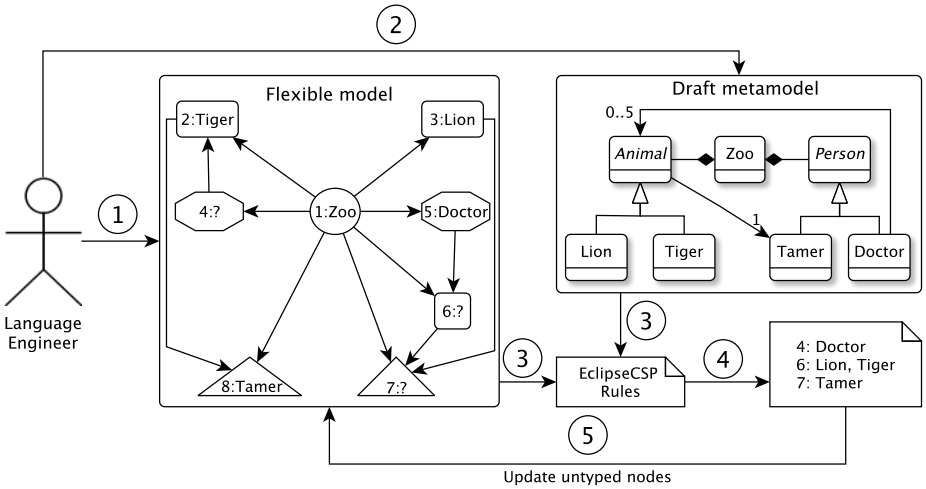


Figure 3: An overview of the proposed approach.

concepts that are defined in the example models. Thus, this approach works under the *Closed World Assumption (CWA)* [12]. This means that at each iteration the inference algorithm can only suggest types that already exist in the metamodel and not introduce new types that might be more relevant but do not exist in the metamodel. The language engineer may want to continue working on the example models by introducing new or evolved concepts. This is an *iterative* process: new concepts can be introduced in the example models, and the metamodel could be updated until a final version is ready. During each iteration, when a stable but incomplete metamodel is defined, the proposed approach can be used to automatically assess the example models and the metamodel. As a result, it can provide suggestions for the nodes that were left untyped to facilitate the engineers having complete models so they can easier proceed to the next iteration. This is done in step (3) as shown in Figure 3. More specifically, a custom-made script analyses the example models and the draft metamodel and produces a set of constraints. This auto-generated file (an example is shown in Listing 1) can be consumed by a constraint solver (e.g. ECL^{i}PS^e [13]) which suggests the possible types for each node (step (4)). More details about the Constraint Satisfaction Problem (CSP) algorithm are given in Section 3.2.

Having the type suggestions generated, the language engineer can pick the correct type from the suggested (if there is more than one) and assign it to the node. The best-case scenario is to suggest a single type to the language engineer. However, this is not always possible, as several candidate types may be applicable. In that case, having the least number of alternative type suggestions is desirable.

A number of factors can affect the number of proposed types of an untyped node. Some of them depend on the metamodel, such as the number of types (less is better), the multiplicity constraints of association ends (tighter is better)

or the relationship between inheritance hierarchies and associations (it works best when the classes participating in associations have few subclasses). Other factors depend on the specific model, such as the degree of untyped nodes and their adjacent nodes (more edges is better) or the location of untyped nodes within the model (it works best when untyped nodes are not adjacent).

As described above, the approach relies on the assumption that the language engineers have acquired enough knowledge from the example models to come up with a draft metamodel that describes the envisioned DSL. Thus, it is important to highlight, that in contrast with our approaches presented in [10] and [11], this work requires the existence of a tentative version of a metamodel.

3.2. Formalisation of the CSP

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

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

A *solution* to a CSP is an assignment of values to variables such that (a) each variable is given a value within its domain and (b) all constraints are satisfied by the assigned values. Depending on the CSP, there may be no solution (*unfeasible* problem), a single solution or more than one.

A CSP describes a problem *declaratively*, without considering how the solution will be computed. *Constraint solvers* that compute solutions to CSPs typically operate using *backtracking-based search*: at each step, one variable is considered and a legal value from its domain is selected, backtracking to previous variables if there are no feasible values available. This generic search procedure can be fine-tuned by defining two heuristics: (a) how the next variable to be assigned should be assigned next and (b) which value from its domain is selected next.

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

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

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

2. *Domain*: Untyped objects may be assigned any non-abstract type in the metamodel. Thus, the domain of each variable is the set of non-abstract types (line 6).
3. *Constraints*: The edges that connect nodes define some restrictions on the valid type assignments (lines 8-12):
   (a) All edges must belong to an association defined in the metamodel (line 9), i.e. the types of source and target nodes must be compatible with some association.

> **Formalization:** Let $\langle obj_1, obj_2 \rangle$ be an edge between two objects obj_1 and obj_2 in the model M. Any object obj is an instance of class $type(obj)$ in the metamodel MM. Pairs of classes may be related through associations, e.g. $\langle t_A, t_B \rangle$, or inheritance hierarchies. Let $super(t)$ denote the set of direct superclasses of a class t and let $ancestors(t)$ denote the set defined inductively as follows: $t \cup ancestors(super(t))$.
> Given an edge e and an association as, the edge is type-compatible with the association if the following holds:
>
> $$\begin{aligned} &compatible\big(e = \langle obj_1, obj_2 \rangle, as = \langle t_A, t_B \rangle\big) := \\ &\qquad (t_1 = type(obj_1)) \ \wedge \ (t_2 = type(obj_2)) \ \wedge \\ &\qquad (t_A \in ancestors(t_1)) \ \wedge \ (t_B \in ancestors(t_2)) \end{aligned}$$
>
> Then, the constraint can be expressed as follows:
>
> $$\forall edge \in M : \exists assoc \in MM : compatible(edge, assoc)$$

   (b) Edges must respect the multiplicity constraints of associations defined in the metamodel (line 12), i.e. the number of edges corresponding to a given association must be between the lower and upper bound.

> **Formalization:** Let l^t_{as} (respectively u^t_{as}) denote the lower (upper) bound on the multiplicity of role t in association as. Let $from(obj, as)$ (respectively, to) denote the number of edges in the model M that have object obj as a source (resp. target) and are compatible with association as:
>
> $$\begin{aligned} from(obj, as) &:= \big(\#e = \langle obj, obj' \rangle \in M : compatible(e, as)\big) \\ to(obj, as) &:= \big(\#e = \langle obj', obj \rangle \in M : compatible(e, as)\big) \end{aligned}$$
>
> Then, the constraint can be expressed as follows:
>
> $$\begin{aligned} &\forall obj \in M : \forall as = \langle t_A, t_B \rangle \in MM : \\ &\qquad (l^{t_A}_{as} \leq from(obj, as) \leq u^{t_A}_{as}) \ \wedge \ (l^{t_B}_{as} \leq to(obj, as) \leq u^{t_B}_{as}) \end{aligned}$$

Algorithm 1 Computing feasible types

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

3.3. Solving the CSP

A solution to this CSP is a type assignment that conforms to the metamodel. However, we are not interested in a single type assignment, but rather the set of potential types for each object for which there is a valid type assignment. Therefore, we will need to solve this CSP several times, once per each pair ⟨variable, type⟩. The existence of a solution to this CSP means that the type is eligible for that variable. The step 2 of Algorithm 1 describes this procedure.

To avoid redundant computations, if a pair ⟨variable, type⟩ has already appeared in the solution to any of the previous CSPs (line 23), then it can be skipped (line 19) as we already know that this type is eligible for that variable. Thus, considering a model with n untyped objects and a metamodel with m non-abstract types, the number of CSPs that need to be solved in the worst-case can be calculated as follows. The total number of ⟨variable, type⟩ pairs is $n \cdot m$. The solution to the first CSP will yield one potential type assignment per variable, i.e. n pairs. Hence, Algorithm 1 will require solving at most $(n \cdot m) - n + 1$ CSPs. The search space of each CSP has n^m potential solutions, even though in practice the majority of CSPs can be solved without exploring the entire search space.

This algorithm has been implemented in the ECL^{i}PS^e [13] Constraint Logic Programming System, which uses a prolog-based syntax to define and solve CSPs. The finite domain (fd) library has been used as the underlying constraint solver.

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

Listing 1: An example file containing the prolog-based syntax automatically generated from the metamodel and example model shown in Figure 2

Listing 1 shows a (partial) example of the generated file for the metamodel and the muddles presented in Figure 2 containing the prolog-based constraints. In lines 3-5 the relationships (both references and aggregations are treated the same way) that appear in the metamodel are listed with the cardinalities. For technical reasons, the many (*) upper limit is set to the value 500 but this could change to anything that it is thought to be a large number for each example. In lines 8-10 all the classes (both abstract and concrete) are instantiated as objects in the problem while in line 13-15, concrete classes are defined. In lines 17-19 the inheritance relationships between the classes are defined. This is all the

information needed from the metamodel. In lines 24-26, each node is assigned with a type using its distinctive identifier. If the type is unknown, meaning that the node has been left untyped, then an "_" underscore is used, prompting the algorithm to assign any valid type to this node. Finally, the links between the nodes in the muddle are defined in lines 29-31. The links are defined using the unique ids of the nodes.

4. Experimental Evaluation

This section presents the experiment run to evaluate the proposed approach. The following are the research questions considered through the experiment:

- **RQ1:** Is there a reduction in the size of the set of candidate types when using the proposed approach? How large is that reduction?
- **RQ2:** Is there a reduction in the size of the set of candidate types when using the proposed approach if isolated nodes are not taken into account? How large is that reduction?

As described in Section 1, when the proposed approach is not deployed the set that includes the candidate types for each untyped node is the number of total concrete types available in the metamodel. The types included in this set will be interchangeably called as "possible" or "candidate" types in this work as they represent the types that a node can take. For instance, in the Zoo example of Figure 3 the number of candidate types for each untyped node is 5, which is the number of concrete types that appear in the metamodel. The purpose of the proposed approach is to prune the size of this set of types by applying the constraints that the draft metamodel includes. This way the approach helps language engineers to select the correct type from smaller sets. All the candidate types are "correct" in terms of not violating the metamodel but we remind the reader here that in our work the term "correct type" is defined as the one that the drawn element actually represents (i.e. the type that the engineer had in mind when drawing the specific node).

Taking all the above into account, the control value for our experiment (i.e. the value that the results of our approach compare with) is the total number of concrete types in the metamodel. In order to assess the performance of the proposed approach and compare it with the control value we introduce the following metric:

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

where n is the number of nodes that are left untyped in the example model, TST stands for the Total Suggested Types returned by the CSP algorithm for the i-th node and TCT stands for the Total Concrete Types in the metamodel. In the Zoo example of Figure 3, this value is 73.3% as the individual savings values

for nodes 4 and 7 are 0.8 (1 - (1/5)) and for node 6 is 0.6 (1- (2/5)). The highest the value, the better the performance of the approach as this is interpreted as the reduction in effort required by the engineer to pick the correct type of each node. A value of 0 means that the algorithm offered no savings at all, as the suggested types are equal to all the concrete types in the metamodel (i.e. the size of the set containing the candidate types remained the same after applying the proposed approach).

The experiment, an overview of which is shown in Figure 4, aims in answering the above research questions. The results of running the experiment are presented in Section 5.

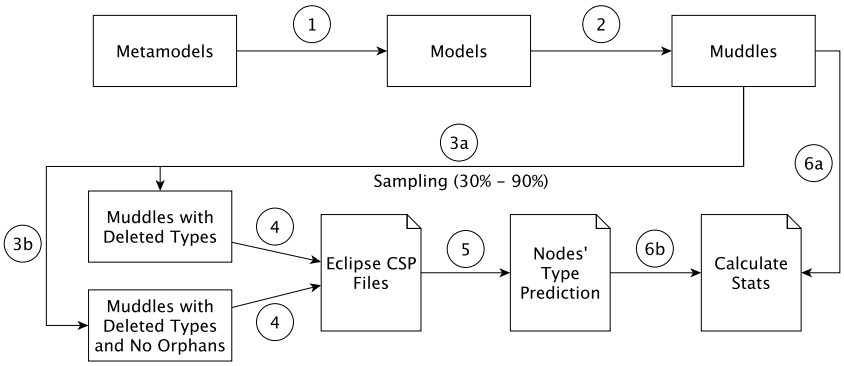


Figure 4: An overview of the experimentation process.

For the purpose of evaluation we applied the proposed approach to a number of randomly generated models, each of which is instance of one of the ten metamodels that were selected in this experiment. The metamodels are those used in the previous work [10], [11]. These metamodels represent the draft/intermediate metamodel that the language engineers came up with from sets of muddle drawings. For each of the metamodels, 10 models were randomly generated using the *Crepe* approach proposed in [14] (step (1) in Figure 4). These 100 randomly generated models are of varying size. The values of the attributes of the different classes of each model were randomly picked from a pool of characters/integers, as they do not affect the performance of the proposed approach. The fact that there is no muddles corpus available led us to the decision of using synthetic muddles based on random generated models. Any threats to validity of that decision are discussed in Section 7.

Step (2) consists of the process of transforming these models into muddles using a custom build model-to-text transformation. At this point, the constructed muddles contain nodes that have their types assigned.

In order to simulate the scenario of having muddles with untyped nodes, a script parses the GraphML files and randomly deletes types of nodes (step (3a) and (3b)). It is of interest to identify whether the proportion of untyped nodes affects the performance (in terms of the ability to reduce the number of

proposed types) of the approach. Thus, 7 different sampling rates, from 30% to 90%, were selected. A 30% sampling rate means that 30% of the nodes had a type assigned to them, leaving the rest 70% being the set of the nodes that are left untyped. The selected rates are the same as the ones used in [10] and [11]. In order to control for sampling bias which can affect the results positively or negatively (e.g. the type of the nodes picked for a simulated type deletion are those whose type can be easily predicted), the type deletion was performed 10 times for each sampling rate for each muddle. At the end of this step 7,000 different muddles were created (10 metamodels x 10 models = 100 x 7 sampling rates = 700 x 10 sampling repetitions = 7,000). In addition, we considered the scenario where some nodes are not only left untyped but are also left isolated (i.e. are not connected with any other node). Our approach cannot currently infer the type of such *orphan* nodes: there is no type assigned to them and they have no relationship with any other node on the diagram. In the future we plan to use other structural features for the inference of orphan nodes such as number and type of attributes. For the second experiment (step (3b)), these nodes were removed from the muddle before commencing the prediction mechanism.

As discussed in Section 3, a file that contains the constraints associated with a model-metamodel pair is used by the ECL^{i}PS^e [13] solver to identify the possible types for each untyped node. In step (4) the ECL^{i}PS^e file for each of the 7,000 muddles (and the 7,000 muddles with no orphan nodes) is automatically generated. These files are consumed by the ECL^{i}PS^e solver (step (5)) and the results are stored in a text file. A single text file is generated for each muddle. The text file contains the mapping between the id of each node and the set of all the suggested types (e.g. 1: A, B, C where 1 is the unique id of the node and A, B, C are the possible types returned).

The performance of the approach is calculated by comparing the correct type of each node and the types that are contained in the set of suggested nodes (step (6b)). The correct type of each node is kept in the text file before commencing the type deletion (step (6a)) to facilitate the comparison. The measures that are used in this work to assess the performance of the approach are discussed in the next section (Section 5) along with the results.

5. Results

Table 5 summarises the corpus of metamodels and the generated models used for the experiment. More specifically, the number of concrete types for each metamodel is given (column "#Types"). The minimum and maximum number of instantiated classes for the 10 generated models of each metamodel are provided in columns "Min" and "Max" respectively, followed by the average number of elements in these 10 models (column "Average #Elements in instances"), for both the experiments with and without including the orphan nodes. As shown in the data, the smallest metamodel is the one used to describe

a university professor, consisting of 4 concrete types. The largest metamodel, is that of describing Wordpress CMS websites with 19 different metaclasses.

Table 2: Data summary table

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

5.1. Quantitative Findings

Table 3 presents the results of the average total savings for each of the 10 metamodels, for the 7 different sampling rates. The results are averaged as for each metamodel there were 10 random models generated and for each sampling rate the type deletion was performed 10 times to avoid the case of a lucky or an unlucky sampling. The first row for each metamodel (entries that are not in italics) includes the results for the scenario where the orphan nodes were taken into account and thus there results give answer to research question RQ1. For example, the highlighted value of 67.38 means that the average savings percentage for 100 example models (10 models x 10 type deletion sessions for each) which have 60% of their types known for the Use Case metamodel is 67.38%, if the orphan nodes are taken into account. As an answer to the research question RQ2 the results for the "no orphans" scenario are also given in Table 3 typed in italics. Thus, for the "no orphans" scenario the effort saving for the above example is 77.57% (highlighted in Table 3). Three metamodels are marked with asterisks. These are the metamodels for which not all the 700 runs were finished due to large statespace. The reasons behind that and more details are explained in Section 5.2.

Two conclusions can be extracted from the results shown in Table 3. Firstly, the savings percentage is not correlated to the size of the metamodel (Orphans: ρ_{Pearson}=0.34, P=0.33, No Orphans: ρ_{Pearson}=0.12, P=0.71). There are large metamodels (e.g. Wordpress, Cobol) where the scores are high while in others (e.g. BibTeX) the score is significantly lower. In the same manner, there are small metamodels (e.g. Professor, Zoo) where the savings are high while in other small metamodels (e.g. Chess) the results are lower. The same applies to the mid-sized metamodels (e.g. Ant Scripts, Use Case and Conference vs. Bugzilla). Secondly, the savings are not affected by the sampling rate. That

Table 3: Average saving results table

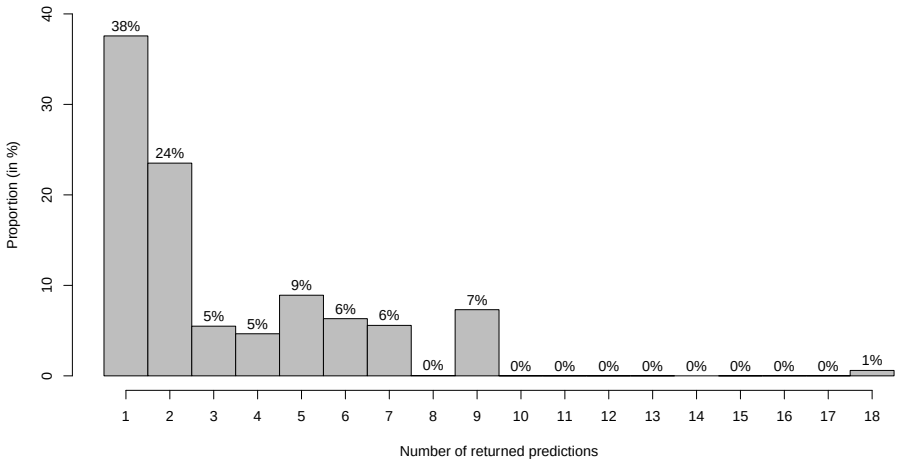
		Average Total Savings Percentage for Different Sampling Rates						
Model	**Types**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**
Professor	4	63.18	63.08	62.92	63.11	63.35	63.32	63.12
		66.28	*66.39*	*66.49*	*66.31*	*66.31*	*65.88*	*67.46*
Chess	5	39.95	39.78	40.33	40.03	40.10	41.60	38.48
		80.00	*80.00*	*80.00*	*80.00*	*80.00*	*80.00*	*80.00*
Zoo	5	69.05	68.38	68.90	68.54	69.25	68.75	69.59
		73.06	*73.18*	*73.14*	*72.99*	*73.67*	*73.52*	*73.88*
Ant*	6	60.57	62.29	61.90	63.08	62.54	62.61	61.97
		70.91	*71.32*	*71.77*	*71.95*	*72.35*	*72.95*	*72.00*
Use Case	7	67.34	67.35	67.89	**67.38**	67.35	67.32	67.64
		77.67	*77.57*	*77.73*	***77.57***	*77.48*	*77.35*	*77.96*
Conference	7	67.11	67.78	67.27	67.76	67.80	66.87	67.23
		74.03	*73.98*	*73.54*	*73.69*	*73.92*	*74.11*	*74.26*
Bugzilla	7	19.67	18.98	19.89	20.19	19.67	19.21	18.93
		31.82	*32.46*	*32.67*	*31.22*	*31.74*	*30.19*	*31.10*
Cobol*	12	75.28	75.69	75.72	76.19	76.04	75.80	78.34
		75.67	*76.21*	*76.80*	*76.71*	*77.36*	*77.38*	*76.75*
BibTeX	14	49.28	49.01	49.47	48.85	49.74	48.76	48.70
		44.34	*44.23*	*44.75*	*43.88*	*44.54*	*43.35*	*44.56*
Wordpress*	19	79.12	80.18	80.76	82.07	81.90	80.23	81.83
		88.05	*89.93*	*90.36*	*90.85*	*91.17*	*91.18*	*91.45*

means that no matter the number of nodes that were left untyped, the performance remains the same. This is an expected result as the CSP algorithm used is not based on machine learning techniques, thus the amount of knowledge that it is available (i.e. the numbed of known nodes) does not affect its performance. This behaviour was identified in our previous work ([10], [11]) where there was a significant improvement in the prediction scores in bigger sampling rates.

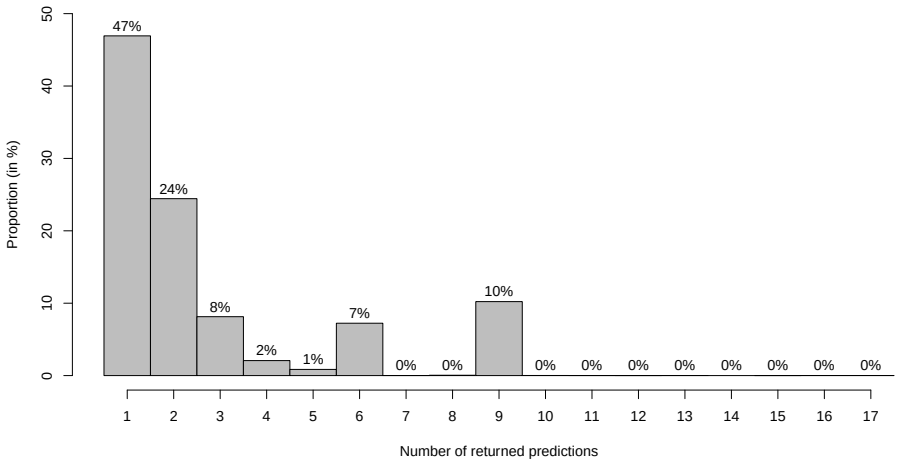
As described in Section 3, one of the differences of this work with the previous is that the algorithm returns a set of suggested types for each node rather than a prediction for the most probable one. The trade-off is that the correct type is guaranteed to be in the list of the suggested types. In the "with orphans" experiment, there were about 155,000 nodes left untyped (about 109,000 for the "no orphans" scenario). In all these cases, the set of the candidate types included the correct type verifying the previous argument. Related to that, it is of interest to assess how many types are returned as suggestions for each of the untyped nodes. Figure 5 presents a histogram to help explore this. For 38% of the untyped nodes in the "with orphans" scenario (RQ1) there is exactly one type returned (for the "no orphans" scenario - RQ2 - this is 47%). This is very important because for more than the one third of the nodes this approach automatically predicted correctly the type of the node without the need of verification or extra help by the language engineer. Although in previous work

80% of the node types were predicted correctly, there was no guarantee about the correctness of the prediction meaning that the language engineer had to verify the prediction manually for each single node.

In addition, in the same histogram one can see that for 24% of the untyped nodes the language engineer has to select between 2 types reducing the amount of effort needed to a minimum (for the "no orphans" scenario this value is also 24%).



(a) With orphans scenario.



(b) No orphans scenario.

Figure 5: Histograms for the number of suggesting types for each node in the experiment that is left untyped

5.2. Qualitative Findings

The time needed for the algorithm to predict the possible types for all the missing nodes in an example model takes from a few milliseconds up to a few seconds. A few experiments for three of the metamodels were not completed in reasonable time. Table 4 presents the number of experiments that were not finished and thus not included in the results. In all the 6 cases, the unfinished experiments are mostly part of the 30% or 40% sampling rate simulations. Any threats to validity are discussed in Section 5.3.

Table 4: Number of unfinished experiments.

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

In the following, we discuss the causes of these timeouts. The execution time of Algorithm 1, depends on the size of the model and the associations and multiplicity constraints. Regarding model size, considering a model with n untyped nodes and a metamodel with m concrete types, the number of potential type assignments grows exponentially, in the order of n^m. Thus, the number of untyped nodes and types has an impact on the efficiency of the solver. This is why the majority of experiments with a timeout are those with the highest rate of untyped objects (e.g. scenarios like the 30% or 40%). Furthermore, the fact that the number of CSPs that needs to be solved in Algorithm 1 grows with the size of the (untyped elements in the) model and (concrete types in the) metamodel makes the impact of the problem size even more significant.

Nevertheless, there is another factor which plays a larger role in the efficiency of the solver: the number and restrictiveness of the constraints. This happens because the solver takes into account the constraints when searching for a solution, using them to prune the search space. Hence, in CSPs where constraints are very tight (e.g. tight bounds for multiplicities), the solver will be able to discard large sections of the search space without needing to explore them. Conversely, in CSPs with few constraints or where constraints are loose, most solutions will satisfy the CSP and the solver will again complete the search quickly.

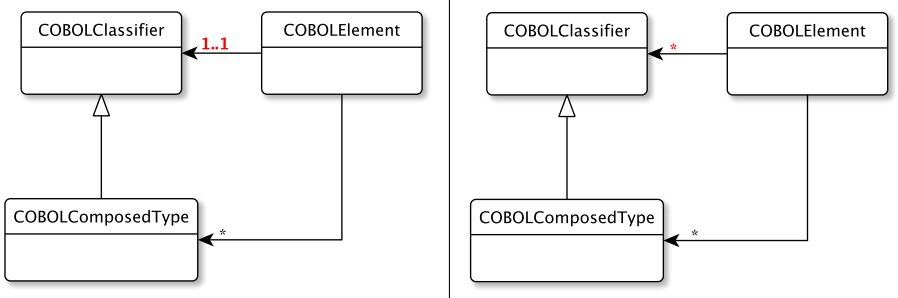
However, there is a certain threshold between those two extremes where solving the CSP becomes more complex and requires exponential runtimes. Within this threshold, constraints discard many solutions, forcing the solver to evaluate many candidates but at the same time pruning is not effective enough to avoid exploring most of the state space. This phenomenon is well known and has been observed empirically in random CSPs, i.e. CSPs with random constraints. For restricted forms of random CSPs [15, 16, 17], it is possible to characterize this threshold in order to detect "hard" CSPs a priori. Still, these results have no generalization to arbitrary CSPs so there is no available method to predict the execution time of the solver for a specific CSP.

5.3. Threats to Validity

As discussed in Section 5.1, the effort saving results are not affected by the number of missing types. In addition, the number of the missing experiments are not of a significant amount for 5 out of 6 of cases where the experiments did not finish within reasonable time (with the exception of the "with orphans" scenario for the Ant Scripts metamodel). Thus, we do not have reasons to believe that the experiments that did not complete affect the validity of our experimental evaluation.

In the cases where a class is extended by one or more other classes we need to accumulate the existence of this class' children in the drawing to check if its upper and lower cardinalities are fulfilled. For instance, in the example of Figure 3, each element of type "Doctor" must treat no more than 5 elements of type "Animal". The constraint programming algorithm aggregates the instances of "Lion" and "Tiger" that each "Doctor" is connected with to validate if the constraint for the "Animal" class is fulfilled. However, the way the example models are represented in our algorithm does not make it feasible to include a corner case where a class/type is connected with the parent class and in parallel it is also connected with one of its children with a reference that has fixed cardinalities.

An example is shown in Figure 6. This corner case is found in two types in total in our experiments (one type in the Cobol metamodel and one type in the Wordpress metamodel). For this corner case, the Muddle-to-Text generator that produces the prolog constraints and commands raises the fixed multiplicity constraint for the class-to-parent reference to many (*). This works against the effectiveness of the proposed approach as suggested types that normally would be rejected due to this multiplicity constraint are included in the list of suggested types decreasing the savings effort figures presented in the results of the experiment.



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

Figure 6: Example of a corner case scenario.

6. Related Work

Various researchers advocate the need for bottom-up or example-driven metamodelling. In [18], the authors argue over the usefulness of bottom-up metamodelling and they identify a set of challenges in using such an approach. Their hypothesis is that bottom-up metamodelling can potentially bridge the *symmetry of ignorance* gap [19] 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. Similarly, in [6] the authors advocate the use of explicit model examples for improving the communication between language engineers and domain experts during the DSL development process.

One of the cornerstones of bottom-up metamodelling is metamodel inference. A number of tool-supported approaches, which provide this functionality, have been proposed in the MDE literature. In [7, 1], an interactive approach for defining metamodels is proposed. Its main goal is to enable the collaboration of domain experts with language engineers during the DSML development process. This approach is supported by a collaborative software tool, which performs metamodel induction from model fragments. A domain expert can specify such fragments either in a sketching tool such as Dia[4] or in a compact textual notation. Model fragments are untyped and they consist of nodes and relations. Once a fragment is defined, the language engineer can annotate it with additional information in order to enhance its semantics. Finally, the enhanced fragments can be consumed by the tool in order to infer the metamodel.

Similarly, in [20] the tool-supported process *MLCBD* is proposed. This process consists of three phases. In the first phase domain experts capture domain knowledge by defining concrete model examples. To do so they use simple shapes and connectors, which are then annotated with domain-specific information. This information will then guide the metamodel inference, which takes place in phase three of the process.

While in the previous works predefined shapes are used for expressing model fragments, in [8] the author proposes a tool-supported process for the definition of DSLs, which supports the inference of metamodels from examples expressed in free-form shapes. Aligned with this work is *FlexiSketch* [21], which is a tool mimicking a white board. It allows users to draw free-form shapes and connections between them. Type annotations can be assigned to the various shapes and then the tool can infer the metamodel. A sketch recognition algorithm matches new free-form shapes without typing annotations to existing shapes, and assigns typing information accordingly.

Clafer [22] is a modeling language with first class support for feature modeling. In [23], the authors use Clafer for example-driven modeling. They argue that model examples can improve domain comprehension among various stakeholders. In their approach incomplete models are expressed in Clafer and then an inference engine uses a metamodel and the initial set of examples in order to

[4] `http://projects.gnome.org/dia/`

derive a set of complete model. To achieve this, the approach takes advantage of Clafer's support for variability modeling. Compared to this work, our approach is more generic, since it does not depend on a dedicated modeling environment.

MARS [24] is another system, which supports metamodel inference. However, the objective of MARS is not to support example-driven metamodelling, but to enable metamodel inference from a set of models after migrating or losing their metamodel. The system relies on a transformation tool, which converts the models from XML to a domain specific language, and on an inference engine, which uses grammar inference techniques on the new representation.

The approaches presented so far support metamodel inference for graphical DSLs. On the other hand, [25] proposes an approach for 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.

Although the overall goal of the aforementioned works is similar to ours (i.e. to support collaboration between domain experts and language engineers during the early stages of language engineering), their focus is quite different. In these approaches it is assumed that model examples are correct and complete before the metamodel inference phase. However, in our work we assume that such model examples can be incomplete, since their completeness is not enforced by a modeling tool. Moreover, we assume that the definition of concrete examples and the metamodel inference is interleaved and one action informs the other. Therefore, in our work we use a version of a derived metamodel in order to improve the quality of a set of model examples.

The approach presented in this work tries to address the same issue (i.e. the problem of type inference in flexible modeling) as the approaches proposed in some of our previous work ([10], [11]). What differentiates our current approach is that it returns a set of possible types for an untyped node, and this set always contains the correct type. The approaches proposed in the past always returns a single type, and the correctness of this type is not guaranteed. This guarantee does not come for free though. For the proposed approach to be effective, the existence of a draft metamodel is required. Finally, the approach proposed in [11] is based on the concrete syntax of the elements appearing in the example models: following drawing conventions improves the inference results. However, domain experts and language engineers may not be interested or ready to express the concrete syntax of the envisioned DSL, especially at the early stages of the language development. The approach proposed here does not take into account the concrete syntax of the language, thus allowing engineers to focus on the abstract syntax.

Bottom-up metamodelling is only one aspect of the collaboration between domain-experts and language engineers during the development of DSLs. In [26] the *Collaboro* approach is proposed. Collaboro is supported by a dedicated DSL, which can model collaborations among stakeholders. More particularly, it can model language change proposals, solution proposals, comments, and rational. Therefore, this approach provides the necessary infrastructure for managing the

incremental development process of a DSL.

Examples are used in MDE not only for inferring metamodels but also for other MDE-related activities. In [27], metamodel well-formedness rules are automatically inferred 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. Moreover, several Model Transformation By-Example (MTBE) approaches (e.g. [28, 29, 30, 31]) 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 [32].

Another line of related work concerns partial modelling. In the literature there are different definitions of model partiality. In [33], 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. In contrast to [33], in our work model partiality means that a model fragment does not need to fully conform to a metamodel. Certain information such as element types can be missing, and constraints imposed by the metamodel such as multiplicities can be ignored. Our notion of model partiality is close to the one of [34] and [35].

More particularly, in [34] the authors propose a diagrammatic, declarative approach to partial model completion based on category theory. In this approach rewriting of partial models is used to support both addition and deletion of models elements. Similarly, in [35] the authors use Constraint Logic Programming (CLP) to assign appropriate values for every missing property in the partial model so that it satisfies the structural requirements imposed by the meta-model. The goal of these two approaches is to enable model completion of partial models in the same way that source code is completed automatically in the editor of an integrated development environment (IDE). On the other hand, in our work we focus on providing support for the early stages of language engineering in a bottom-up metamodelling context.

Apart from model completion, CLP is applied in other MDE scenarios. In [36], the *EMFtoCSP* tool is proposed, which is used for the verification of EMF [37] models annotated with OCL constraints. This tool checks for the following constraints: strong satisfiability, weak satisfiability, lack of constraint subsumptions and lack of constraint redundancies. Verification is performed by translating the EMF model and its accompanying constraints into a constraint satisfaction problem, which is then solved by a constraint solver. In a similar manner the *UMLtoCSP* [38] tool uses CLP for the formal verification of UML class diagrams.

7. Conclusions and Future Work

In this paper a novel approach to tackle the problem of type omissions in the emerging domain of flexible MDE is proposed. More specifically, in our approach partial model examples and their accompanying metamodel are translated into

a set of constraints, and then the missing typing information are derived by solving the constraint satisfaction problem. The proposed approach was evaluated by running an experiment on a corpus of 14,000 randomly generated example models. We defined a metric that measures the space of possible types for each node, showing significant improvement in the effort needed to fill the types of the missing nodes.

At this point our approach returns a set of equally probable type suggestions. In the future we would like to reduce even further the size of this set, by proposing the most probable types. This can be done in a number of ways. Firstly, by taking into account the names of different elements, such as the name of the attributes of the nodes in the example model and the metamodel. In the future, we plan to include this information to help in the direction of sorting the suggested types by using string distance/similarity metrics to identify possible matches or synonyms. Secondly, the work carried out in [10] and [11] could also be combined with this work to improve the type suggestion as the type returned from those approaches can be placed at the top of the list of the suggested types. In addition, the Flexisketch's type inference approach proposed in [5] and is based on the graphical similarity between the nodes can be also used to further reduce or sort the set of candidate types produced as a result of our proposed approach. Moreover, the metaBUP tool presented in [1] could be combined with our proposed approach for the semi-automatic inference of the draft metamodel.

Finally, a long term goal is the application of the proposed approach on real example models which are a result of a flexible MDE approach applied on a complete project. This way we will be able to evaluate the proposed approach not only on a single increment, but in more than one iterations that normally take place in flexible MDE approaches.

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). Special thanks to Mr. Carlo Capelli and Mr. Sotirios Banatas for the help provided during the initial phase of the Prolog implementation.

8. References

[1] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, J. de Lara, Example-driven meta-model development, Software & Systems Modeling (2013) 1–25.

[2] H. Ossher, R. Bellamy, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, M. Desmond, J. de Vries, A. Fisher, S. Krasikov, Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges, ACM Sigplan Notices 45 (10) (2010) 848–864.

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

[4] G. Gabrysiak, H. Giese, A. Lüders, A. Seibel, How can metamodels be used flexibly, in: Proceedings of ICSE 2011 workshop on flexible modeling tools, Waikiki/Honolulu, Vol. 22, 2011.

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

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

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

[8] M. Kuhrmann, User assistance during domain-specific language design, in: FlexiTools workshop, 2011.

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

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

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

[12] R. Reiter, On closed world data bases, in: Logic and data bases, Springer, 1978, pp. 55–76.

[13] K. R. Apt, M. Wallace, Constraint logic programming using ECLiPSe, Cambridge University Press, 2006.

[14] J. R. Williams, R. F. Paige, D. S. Kolovos, F. A. Polack, Search-based model driven engineering, Tech. rep., Technical Report YCS-2012-475, Department of Computer Science, University of York (2012).

[15] P. Prosser, An empirical study of phase transitions in binary constraint satisfaction problems, Artificial Intelligence (1996) 81–109.

[16] D. G. Mitchell, Proc. of the 8th International Conference on Principles and Practice of Constraint Programming (CP'2002), Springer, Berlin, Heidelberg, 2002, Ch. Resolution Complexity of Random Constraints, pp. 295–310.

[17] Y. Gao, J. Culberson, Resolution complexity of random constraint satisfaction problems: Another half of the story, Discrete Applied Mathematics 153 (13) (2005) 124 – 140.

[18] H. Cho, Y. Sun, J. Gray, J. White, Key challenges for modeling language creation by demonstration, in: 2011 ICSE Workshop on Flexible Modeling Tools, Honolulu HI, 2011.

[19] P. Dillenbourg, What do you mean by collaborative learning, Collaborative-learning: Cognitive and computational approaches 1 (1999) 1–15.

[20] H. Cho, J. Gray, E. Syriani, Creating visual domain-specific modeling languages from end-user demonstration, in: 2012 ICSE Workshop on Modeling in Software Engineering (MISE), IEEE, 2012, pp. 22–28.

[21] D. Wuest, N. Seyff, M. Glinz, Semi-automatic generation of metamodels from model sketches, in: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 664–669.

[22] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, A. Wasowski, Clafer: unifying class and feature modeling, Software & Systems Modeling (2015) 1–35.

[23] M. Antkiewicz, K. Bak, K. Czarnecki, Z. Diskin, D. Zayan, A. Wasowski, Example-driven modeling using clafer., in: MDEBE@MoDELS, Vol. 1104, CEUR-WS.org, 2013, pp. 32–41.

[24] F. Javed, M. Mernik, J. Gray, B. R. Bryant, MARS: A metamodel recovery system using grammar inference, Information and Software Technology 50 (9) (2008) 948–968.

[25] B. Roth, M. Jahn, S. Jablonski, On the way of bottom-up designing textual domain-specific modelling languages, in: Proceedings of the ACM workshop on Domain-specific modeling, 2013, pp. 51–56.

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

[27] M. Faunes, J. Cadavid, B. Baudry, H. Sahraoui, B. Combemale, Automatically searching for metamodel well-formedness rules in examples and counter-examples, in: Model-Driven Engineering Languages and Systems, Springer, 2013, pp. 187–202.

[28] M. Kessentini, H. Sahraoui, M. Boukadoum, O. B. Omar, Search-based model transformation by example, Software & Systems Modeling 11 (2) (2012) 209–226.

[29] M. Strommer, M. Wimmer, A framework for model transformation by-example: Concepts and tool support, in: Objects, Components, Models and Patterns, Springer, 2008, pp. 372–391.

[30] P. Langer, M. Wimmer, G. Kappel, Model-to-model transformations by demonstration, in: Theory and Practice of Model Transformations, Springer, 2010, pp. 153–167.

[31] D. Varró, Z. Balogh, Automating model transformation by example using inductive logic programming, in: Proceedings of the 2007 ACM symposium on Applied computing, ACM, 2007, pp. 978–984.

[32] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer, Model transformation by-example: a survey of the first wave, in: Conceptual Modelling and Its Theoretical Foundations, Springer, 2012, pp. 197–215.

[33] M. Famelis, R. Salay, M. Chechik, Partial models: Towards modeling and reasoning with uncertainty, in: 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 573–583.

[34] F. Rabbi, Y. Lamo, I. C. Yu, L. M. Kristensen, L. Michael, A diagrammatic approach to model completion, in: 4th Workshop on the Analysis of Model Transformations (AMT)@ MODELS, Vol. 15, 2015.

[35] S. Sen, B. Baudry, D. Precup, Partial model completion in model driven engineering using constraint logic programming, in: International Conference on the Applications of Declarative Programming, Citeseer, 2007.

[36] C. A. González, F. Buettner, R. Clarisó, J. Cabot, EMFtoCSP: A tool for the lightweight verification of EMF models, in: Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches, IEEE Press, 2012, pp. 44–50.

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

[38] J. Cabot, R. Clarisó, D. Riera, On the verification of UML/OCL class diagrams using constraint programming, Journal of Systems and Software 93 (2014) 1–23, http://dx.doi.org/10.1016/j.jss.2014.03.023.