RESEARCH ARTICLE OPEN ACCESS

Restriction Programming For Type Interpretation in Supple Model-Driven Engineering

A. Ramalakshmi
Ph.D. Scholar
Dept. of Computer Science
Manonmanium Sundaranar University, Tirunelveli
Tamil Nadu – India

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 meta model 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 meta modelling infrastructure. The meta mode list 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 un typed. As result, concepts that might be important for the definition of the Domain will be ignored while the example models cannot be adequate lyre 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 un typed 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.

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

I. INTRODUCTION

Conventional Domain-Specific Languages (DSL) definition processes begin with the creation of a meta model that is then accustomed instantiate models and guide the event of editors and alternative are facts like model-to-model and model-to text transformations. Such a method implies experience in meta modelling, and in relevant technologies. whereas this could be a straight forward or a minimum of graspable method for MDE consultants, this is often not perpetually the case with domain consultants [1] WHO are additional at home with tools like easy drawing editors [2]. However, the involvement of domain consultants is very important within the definition of top quality and well-defined DSLs that cowl all the required aspects of a site. to deal with the same issue, versatile modelling approaches are projected within the literature (e.g. [3–5,1]). Such approaches are supported sketching tools and don't need the definition of a metamodel throughout the initial phases of language engineering. additional specifically, in versatile (or bottom-up) MDE, the method starts with the definition of example models [1,6,7].

These example models facilitate language engineers to raised perceive the ideas of the visualised subscriber line and might be accustomed infer draft metamodels manually or (semi-) automatically that eventually lead within the definition of the ultimate metamodel. during this fashion, a richer understanding of the domain are often developed incrementally, whereas concrete insights (e.g. kind information) touching on the visualised metamodel are discovered. Fig. one depicts the stages going down in a very typical versatile MDE method as this is often understood by finding out completely different versatile MDE approaches within the literature (e.g. [1,2,8]). The sketching tools employed in such processes permit the fast definition of ideal models sacrificing the formality that model editors, that are supported a rigorously-defined metamodels, offer. additionally, drawing tools don't need MDEspecific experience. the weather (nodes and edges) of those versatile example models will have kind annotations assigned to them to explain the domain thought they represent and might even be amenable to programmatic model management victimisation MDE suites like letter of the alphabet [9]. On the opposite hand, since sketching tools cannot

enforce syntactical and linguistics correctness rules, versatile models are liable to numerous styles of errors [10]: User input errors: parts that ought to share a similar kind have differing types assigned to them by mistake or as a results of a literal error (e.g. Animal vs. Anmal). Changes as a result of evolution: parts representing ideas that have evolved throughout the domain exploration method don't have their varieties updated (e.g. Animal vs. Herbivores and Carnivores). Inconsistencies as a result of collaboration: once multiple domain consultants collaborate within the definition of the models, multiple varieties representing a similar thought are often used (e.g. Doctor vs. Veterinarian). Omissions: parts are often left untyped particularly once models become massive because it is less complicated to overlook a number of the weather. The trade-off between formality and suppleness will presumably end in a more robust domain understanding by language engineers, and eventually to the next quality language. bottom-up meta modelling is associate repetitive method, since completely different versions of metamodels associated ideal models are ceaselessly evolved in an interleaved manner till the ultimate version of the metamodel is obtained [1]. The contribution of this paper could be a tool-supported approach for eliminating kind omission errors (see error tagged "Omissions" above) from versatile models. presently such errors are eliminated manually by language engineers by choosing associate acceptable kind from a group of potential varieties.

That means, that if within the draft metamodel there are N completely different concrete varieties, the language engineer has N choices for every untyped component. However, this approach doesn't have the benefit of data that exists within the draft metamodel which might presumably facilitate in reducing the quantity of potential varieties for a particular node. for instance, if within the metamodel it's outlined that nodes of kind "A" will solely be connected with nodes of kind "B" then if associate untyped node is connected with a node of kind "B", it are often inferred that {the kind the sort the kind} of the missing node is type "A". a plus of that second approach is that the search house for the potential varieties steered to the language engineer are often reduced from N to M, wherever M A [1, N], from that the engineer needs to choose the right one. during this work, the meant which means of the "correct kind" is that the type that the engineer visualised once drawing the precise component within the example model. Our projected approach doesn't need a metamodel that's refined to follow the simplest practices or patterns projected for metamodel development. the sole demand is that the metamodel should embody all the kinds and therefore the relationships that are gift within the example models. The term "draft" that characterizes the metamodels required as a part of this approach implies first of all the optional want of getting meta modelling best practices applied to the metamodel. Secondly, it are often "draft" in terms of not being a final one that covers all the ideas of the domain, however associate intermediate one that covers a set of the ideas, as shortly as these are the sole ideas showing within the example models. This draft metamodel, following the repetitive versatile MDE principles, ought to reach a final version that covers all the visualised ideas and relationships. associated with that, it's necessary to spotlight that our projected approach focuses solely on the abstract thought of the kinds of the untyped nodes within the example models created (or evolved) as a part of a versatile MDE approach

(see "Example models definition" phase in Fig. 1). The way that the draft

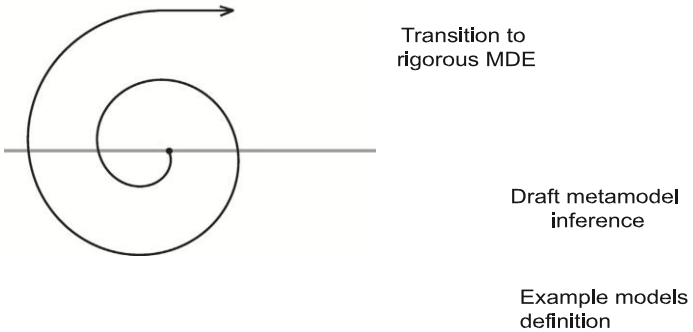

Fig. 1. Stages of a typical Flexible MDE approach.

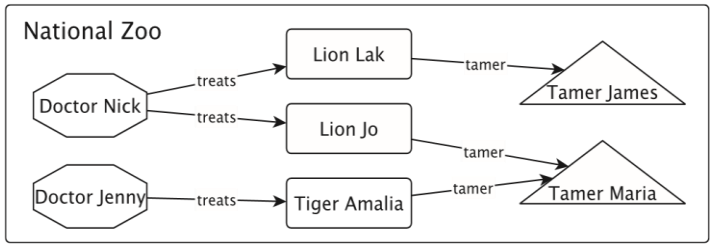

Fig. 2. An example Zoo diagram.

metamodel is inferred (or evolves) is outside the scope of this work and it's a noteworthy space of future research; the draft metamodel ought to fulfil the said demand, though. In previous work [10,11], associate approach to tackle the error of kind omissions was projected. The formula, that's supported Classification and Regression Trees (CARET), is trained on the weather of the instance models solely, attempting to spot similarities supported completely different sets of criteria and predict the categories of untyped nodes while not requiring the existence of a metamodel. This work contributes a unique approach in addressing the challenges related to kind omissions, however now taking under consideration a draft metamodel created by language engineers, exploitation constraint programming principles for suggesting the attainable sorts. additional specifically, the syntax and therefore the constraints outlined within the draft metamodel area unit mechanically remodelled to a collection of facts and rules that area unit then applied to the instance models to scale back the amount of attainable varieties of untyped nodes. on the fare side the need for a metamodel, another necessary distinction with the previous approach is that the indisputable fact that during this work, the right kind for every node is often enclosed within the set of instructed sorts. within the previous work, the instructed kind isn't bound to be the right one. A trade-off for that's the actual fact that there may be over one attainable sorts instructed for every node whereas within the previous work there was continually one kind came back, not continually the right one although. the remainder of this paper is structured as follows. Section two includes a short review of a selected versatile modelling approach, Muddles [3], that is predicated on GraphML and is employed as a signal of thought for the projected approach and therefore the experimentation. In Section three the approach is given. In Section four, associate empirical analysis of the performance of the projected approach is conducted. The results of running the experiments area unit mentioned in Section five, alongside threats to experimental validity. In Section half-dozen, connected add the sphere of versatile modelling, kind logical thinking and constraint programming in MDE is given. In Section seven we have a tendency to conclude the paper and description plans for future work.

II. BACKGROUND

In our work we have a tendency to use the Muddles approach [3] for sketching model examples. The Muddles approach proposes the utilization of GraphML compliant tools like yEd for the definition of model sketches. additional notably, the domain engineer will use straight forward drawing editors to precise example models of associate unreal Domain-Specific Language (DSL). exploitation straight forward drawing editors to precise the domain ideas will increase the participation of the domain specialists because it needs no (or minimum) coaching whereas it permits them operating with tools that they're already conversant in [1,2]. The drawing created, known as a muddle, will be consumed by model management suites, just like the alphabetic character platform [9], to support MDE activities (e.g. queries, transformations) facultative the language engineer to experiment with the models, gain higher understanding and judge whether or not they area unit appropriate purpose. associate example of such a muddle is shown in Fig. 2. during this example, the intention of the language engineer is that the creation of an easy phone line for outlining Zoos. the method starts with the definition of example models of this unreal phone line by the domain specialists. The language engineer will then annotate sorts and typewriting info (e.g. properties) for every node. This drawing will be then consumed by model management programs that may be written in parallel to see if it fulfils the requirements of the engineer and expose additional options of the language, if any. In distinction to connected work [1], shapes and alternative graphical characteristics of every node don't seem to be certain to sorts. for instance, within the drawing of Fig. 2, components of kind “Lion” area unit expressed exploitation rounded rectangles, constant form that's want to outline components of kind “Tiger”. Moreover, components of constant kind will be expressed exploitation completely different shapes, as example 2 completely different components of kind

"Lion" will be expressed exploitation the rounded parallelogram for one among them and a circle for the second. By doing thus, the domain skilled isn't affected by a concrete syntax and might use any form and arrow offered from the tool's palette to precise herself freely. sorts and type-related info for every component area unit outlined exploitation custom GraphML properties. this can be associate extensibility mechanism provided by GraphML to support attaching absolute key-value info to graph components. a brief description and examples for every custom property is given in Table one. additional details on these properties will be found in [3]. it's necessary to say here that the projected approach isn't finite with the Muddles approach, that is merely used as proof of thought and for experimentation functions, however will be custom-made to and employed in principle with the other versatile modelling approach that fulfils the subsequent tokenism set of requirements:

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

Extension	For	Description	Example
Type	Node, Edge	The type of the element	Lion, Doctor o Person (o denotes the extend relationship)
Properties	Node, Edge	Descriptors and values for primitive attributes of nodes/edges	String name¼Jenny, Integer age¼25
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
Tareget role	Edge	Descriptor of the role of the tareget end of the edge	tareget, taregetNode
Role in source	Edge	The role of the edge in its source node	patient 0..5, tamer 1
Role in tareget	Edge	The role of the edge in its tareget node	carer ^{n}, employee ^{n}

allows users to assign sorts to nodes and provides a mechanism to extract these types; provides a mechanism to permit the extraction of the supply and target nodes of the perimeters within the example models.

III. PROJECTED KIND ILLATION APPROACH

3.1 summary of the approach

During this section the projected approach for kind illation in versatile MDE is conferred. an summary is given in Fig. 3. the method starets with the language engineers having example models drawn employing a versatile modelling approach (i.e. muddles during this scenario) to facilitate the method of shaping the subscriber line they're inquisitive about (step ①). this could involve continuous changes to the instance models, when that enough data is nonheritable for the assembly of a primarey draft version of the DSL's metamodel (step ②). the instance models as a result of the shortage of editors that area unit supported pre-defined metamodels (see Section 1) might have nodes that area unit left untyped. At this stage, the metamodel may be a partial one that solely describes the ideas that area unit outlined within the example models. Thus, this approach works beneath the Closed World Assumption (CWA) [12]. this {suggests this implies} that at every iteration the illation formula will solely suggest sorts that exist already within the metamodel and not introduce new sorts that may be a lot of relevant however don't exist within the metamodel. The language engineer might want to continue functioning on the instance models by introducing new or evolved ideas. may be} an unvarying process: new ideas can be introduced within the example models, and therefore the metamodel may be updated till a final version is prepared. throughout every iteration, once a stable however incomplete metamodel is outlined, the projected approach may be accustomed mechanically assess the instance models and therefore the metamodel. As a result, it will give suggestions for the nodes that were left untyped to facilitate the engineers having complete models in order that they will easier proceed to consecutive iteration. this can be worn out step ③ as shown in Fig. 3. a lot of specifically, a made-to-order script analyses the instance models and therefore the draft metamodel and produces a group of constraints. This auto-generated file is an example is shown in Listing; may be consumed by a constraint convergent thinker (e.g. ECLiPSe [13]) that suggests the doable sorts for every node (step ④). a lot of

details regarding the Constraint Satisfaction drawback (CSP) formula are unit given in Section three.2. Having the sort suggestions generated, the language engineer will choose the proper kind from the steered (if there's over one) and assign it to the node. The best-case state of affairs is to counsel one kind to the language engineer. However, this can be not perpetually doable, as many candidate sorts could also be applicable. in this case, having the smallest amount variety of other kind suggestions is desirable.

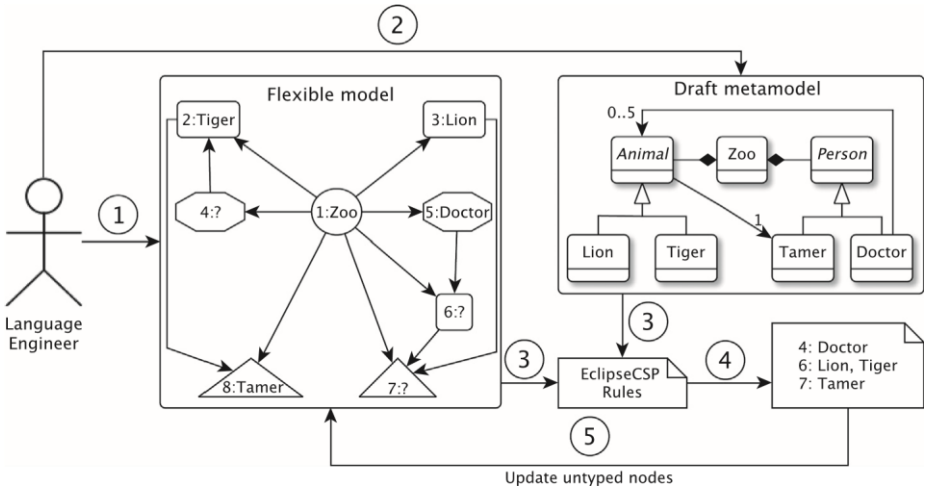

Fig. 3. An overview of the projected approach.

variety of things will Associate in Nursing effect on} the amount of projected sorts of an untyped node. a number of them rely on the metamodel, like the amount of sorts (less is better), the multiplicity constraints of association ends (tighter is better) or the link between inheritance hierarchies and associations (it works best once the categories collaborating in associations have few subclasses). different factors rely on the precise model, like the degree of untyped nodes and their adjacent nodes (more edges is better) or the situation of untyped nodes at intervals the model (it works best once untyped nodes aren't adjacent). As represented higher than, the approach depends on the idea that the language engineers have nonheritable enough information from the instance models to return up with a draft metamodel that describes the visualised line. Thus, it's necessary to focus on that in distinction with our approaches bestowed in [10,11], this work needs the existence of a tentative version of a metamodel.

3.2. systematization of the CSP

During this section we have a tendency to describe however the sort assignment drawback is formalized as a Constraint Satisfaction drawback (CSP). A CSP is characterised by 3 elements: The set of variables concerned within the drawback. The domain of every variable, i.e. the set of potential values it will take. The constraints over the variables that outline that price assignments square measure valid. associates} to a CSP is an assignment of values to variables such (a) every variable is given a worth at intervals its domain and (b) all constraints square measure glad by the allotted values. reckoning on the CSP, there is also no resolution (unfeasible problem), one resolution or over one. A CSP describes a haul declaratively, while not considering however the answer are going to be computed. Constraint solvers that reckon solutions to CSPs usually operate victimisation backtracking-based search: at every step, one variable is taken into account and a legal price from its domain is chosen, backtracking to previous variables if there aren't any possible values offered. This generic search procedure are often fine-tuned by shaping 2 heuristics: (a) however future variable to be allotted ought to be allotted next and (b) that price from its domain is chosen next. what is more, search are often created a lot of economical by victimisation optimizations like propagation (i.e. use partial assignments to get rid of unproductive values from domains of unassigned variables) or back jumping (i.e. rethink many selections in every backtrack). These options square measure provided in most progressive constraint solvers and, hence, these optimizations don't have to be compelled to be enforced manually within the definition of every CSP. Considering these preliminaries, kind assignment within the versatile modelling approaches represented in previous sections are often formalized because the following CSP, as represented in Step one of algorithmic program 1: Variables: there's one variable per untyped node within the model, representing the sort of that node (line 4). Domain: Untyped objects is also allotted any non-abstract kind within the metamodel. Thus, the domain of every variable is that the set of non-abstract sorts (line 6). Constraints: the perimeters that connect nodes

outline some restrictions on the valid kind assignments (lines 8–12): All edges should belong to associate association outlined within the metamodel (line 9), i.e. the kinds of supply and target nodes should 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:

$$compatible\big(e = \langle obj_1, obj_2 \rangle, as = \langle t_A, t_B \rangle\big) := (t_1 = type(obj_1)) \wedge (t_2 = type(obj_2)) \wedge (t_A \in ancestors(t_1)) \wedge (t_B \in ancestors(t_2))$$

Then, the constraint can be expressed as follows:

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

Edges must respect the multiplicity constraints of associations defined in the meta model (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:

$$from(obj, as) := \big(\#e = \langle obj, obj' \rangle \in M : compatible(e, as)\big)$$
$$to(obj, as) := \big(\#e = \langle obj', obj \rangle \in M : compatible(e, as)\big)$$

Then, the constraint can be expressed as follows:

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

Algorithm 1. Computing feasible types.

1: {Step 1: Construct the CSP}
2: $N \leftarrow$ set of untyped nodes in Model
3: $T \leftarrow$ set of non-abstract types in MetaModel
4: $Vars \leftarrow N$ {Define variables}
5: **for all** $v \in Vars$ **do**
6: $Domain(v) \leftarrow T$ {Define domains}
7: $Constraints \leftarrow \emptyset$ {Define constraints}
8: **for all** $edge \in Model$ **do**
9: $Constraints \leftarrow Constraints \cup compatibleAssociation(edge, MetaModel)$
10: **for all** $node \in Model$ **do**
11: **for all** $association \in MetaModel$ **do**
12: $Constraints = Constraints \cup multiplicityBounds(node, association)$
13:
14: {Step 2: Find feasible types by iteratively solving the CSP}
15: **for all** $v \in Vars$ **do**
16: **for all** $d \in T$ **do**
17: Feasible[v,d] $\leftarrow$ **false**
18: **for all** $v \in Vars$ **do**
19: **for all** $d \in Domain(v)$ such that Feasible[v,d] = **false do**
20: $solution \leftarrow solveCSP(Vars, Domain, Constraints \cup (v = d))$
21: **if** solution exists **then**
22: **for all** $v' \in Vars$ **do**
23: Feasible$[v', value(v', solution)] \leftarrow$ **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 and 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 on mÞnþ1 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 prolongated syntax to define and solve CSPs. The finite domain (fd) library has been used as the underlying constraint solver.

Listing 1. An example file containing the prolog-based syntax automatically generated from the meta model and example model shown in Fig. 2.

```
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 shows a (partial) example of the generated file for the metamodel and the muddles presented in Fig. 2 containing the prolong-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 (^{n}) 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 lines 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.

IV. EXPERIMENTAL EVALUATION

This section presents the experiment run to evaluate the proposed approach. The following are the researech 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 larege 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 larege 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 Fig. 3 the number of candidate types for each untyped node is 5, which is the number of concrete types that appeare 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:

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

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 Fig. 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 greater 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 Fig. 4, aims in answering the above researech questions. The results of running the experiment are presented in Section 5. 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 Fig. 4). These 100 randomly generated models are of vareying size. The values of the attributes of the different classes of each model were randomly picked from a pool of chareacters/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 scenareio of having muddles with untyped nodes, a script pareses the GraphML files and randomly deletes types of nodes (steps and). 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 [10,11]. 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 is 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 7000 different muddles were created (10 metamodels 10 models¼100 7 sampling rates¼700 10 sampling repetitions¼7000). In addition, we considered the scenareio 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 7000 muddles (and the 7000 muddles with no orphan nodes) is automatically generated.

These files are consumed by the ECL^{i}PS^e

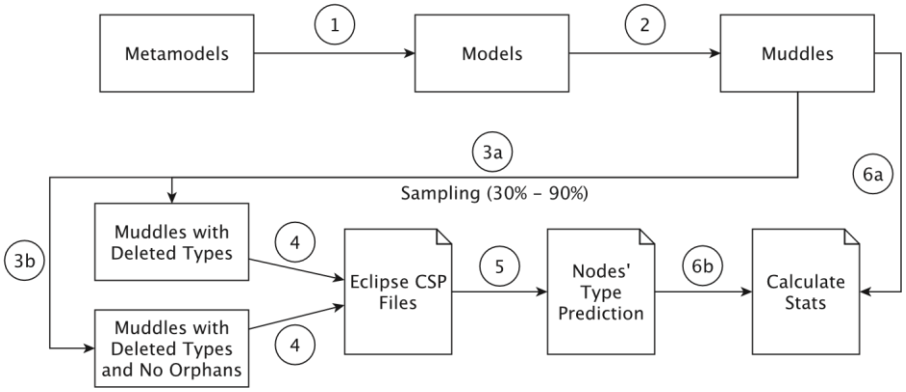

Fig.4. An overview of the experimentation process. Table 2 Data summarey table.

Model name	#Types	With orphans			No orphans		
		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

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.

V. RESULTS

Table 2 summarizes 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 laregest metamodel is that of describing Word press CMS websites with 19 different meta classes.

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 their 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 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 is 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 state space. 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: $\rho_{Pearson}$¼0.34, P¼0.33, No Orphans: $\rho_{Pearson}$¼0.12, P¼0.71). 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 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 learening techniques, thus the amount of knowledge that 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 guareanteed to be in the list of the suggested types. In the "with orphans" experiment, there were about 155,000 nodes left

Table 3 Average saving results table.

Model	Types	Average total savings percentage for different sampling rates						
		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[n]	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[n]	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[n]	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

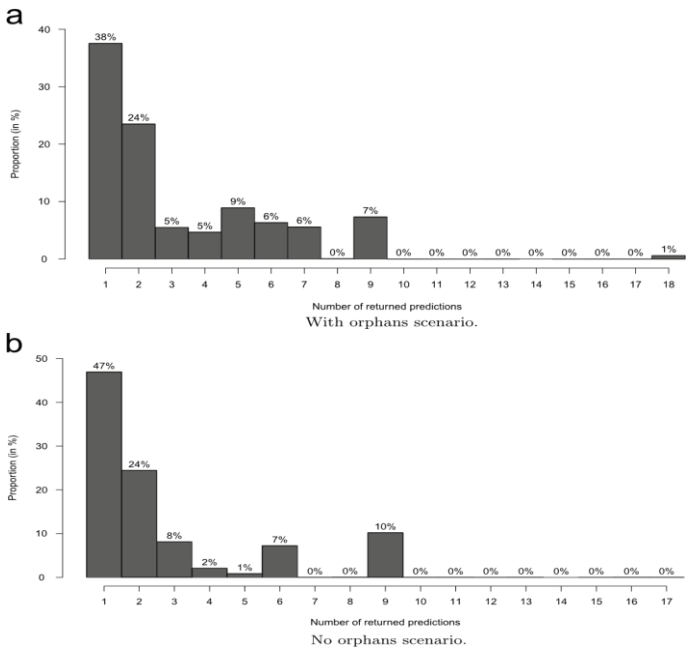

Fig. 5. Histograms for the number of suggesting types for each node in the experiment that is left untyped. untyped (about 109,000 for the "no orphans" scenareio).

In all these cases, the set of the candidate varieties enclosed the proper sort supportive the previous argument. associated with that, it's of interest to assess what number varieties square measure came as suggestions for every of the untyped nodes. Fig. five presents a bare graph to assist explore this. For thirty eighth of the untyped nodes within the "with orphans" state of affairs (RQ1) there's specifically one sort came (for the "no orphans" state of affairs – RQ2 – this is often 47%). this is often vital as a result of for quite the simple fraction of the nodes this approach mechanically foretold properly the kind of the node while not the necessity of verification or further facilitate by the language engineer. though in previous work eightieth of the node varieties were foretold properly, there was no guarantee regarding the correctness of the prediction which means that the language engineer had to verify the prediction manually for every single node. additionally, within the same bare graph one will see that a pair of4|for twenty-four} of the untyped nodes the language engineer should choose between 2 varieties reducing the quantity of effort required to a minimum (for the "no orphans" state of affairs this price is additionally 24%).

5.2. Qualitative findings

The time required for the algorithmic rule to predict the doable varieties for all the missing nodes in associate degree example model takes from some milliseconds up to some seconds. some experiments for 3 of the metamodels weren't completed in affordable time. Table four presents the quantity of experiments that weren't finished and so not enclosed within the results. altogether the six cases, the unfinished experiments square measure largely a part of the half-hour or four-hundredth rate simulations. Any threats to validity square measure mentioned in Section five.3. within the following, we tend to discuss the causes of those timeouts. The execution time of algorithmic rule one depends on the scale of the model and also the associations and multiplicity constraints. concerning model size, considering a model with n untyped nodes and a metamodel with m concrete vareieties, the quantity of potential sort assignments grows exponentially, within the order of nm. Thus, the quantity of untyped nodes and kinds has a control on the potency of the thinker. this is often why the bulk of experiments with a timeout square measure those with the very best rate of untyped objects (e.g. eventualities just like the half-hour or 40%). moreover, the very fact that the quantity of CSPs that has to be resolved in algorithmic rule one grows with the scale of the (untyped components in the) model and (concrete vareieties in the) metamodel makes the impact of the matter size even a lot of vital. however, there's another issue that plays a bigger role within the potency of the solver: the quantity and modification of the constraints. This happens as a result of the thinker takes under consideration the constraints once finding out an answer, exploitation them to prune the searech area. Hence, in CSPs wherever constraints square measure terribly tight (e.g. tight bounds for multiplicities), the thinker are going to be able to discared giant sections of the searech area with no need to explore them.

Conversely, in CSPs with few constraints or wherever constraints square measure loose, most solutions can satisfy the CSP and also the thinker can once more complete the searech quickly. However, there's a pareticulare threshold between those 2 extremes wherever finding the CSP becomes a lot of advanced and needs exponential runtimes. among this threshold, constraints discared several solutions, forcing the thinker to guage several candidates however at identical time pruning isn't effective enough to avoid exploring most of the state area. This development is acknowledge and has been determined by trial and error in random CSPs, i.e. CSPs with random constraints. For restricted vareieties of random CSPs [15–17], it's doable to characterize this threshold so as to find "hared" CSPs a priori. Still, these results haven't any generalization to absolute CSPs therefore there's no offered methodology to predict the execution time of the thinker for a particular CSP.

5.3. Threats to validity

As mentioned in Section five.1, the hassle saving results don't seem to be tormented by the quantity of missing varieties. additionally, the {amount the quantity} of the missing experiments isn't of a major amount for five out of six of cases wherever the experiments didn't end among affordable time (with the exception of the "with orphans" state of affairs for the pismire Scripts metamodel). Thus, we tend to don't have reasons to believe that the experiments that didn't fully have an effect on the validity of our experimental analysis. within the cases wherever a category is extended by one or a lot of alternative categories we'd like to accumulate the existence of this class' youngsters within the drawing to ascertain if its higher and lower caredinalities square measure consummated. as an example, within the example of Fig. 3, every paret of sort "Doctor" should treat no quite five components of sort "Animal". The constraint programming algorithmic rule aggregates the instances of "Lion" and "Tiger" that every "Doctor" is connected with to validate if the constraint for the "Animal" category is consummated. However, the approach the instance models square measure painted in our algorithmic rule doesn't build it possible to incorporate a corner case wherever a category/type is connected with the parent class and in pareallel it's conjointly connected with one amongst its youngsters with a reference that has fastened cardinalities. associate degree example is shown in Fig. 6. This corner case is found in 2 varieties in total in our experiments (one sort within the programming language metamodel and one sort within the WordPress metamodel). For this corner case, the Muddle-to-Text generator that produces the programming language constraints and commands raises the fastened multiplicity constraint for the class-to-parent relevancy several (n). This works against the effectiveness of the planned approach as urged varieties that unremarkably would be rejected attributable to Table four range of unfinished experiments.

Metamodel	With orphans	No orphans
Ant scripts	86	13
Cobol	3	2
Wordpress	44	3

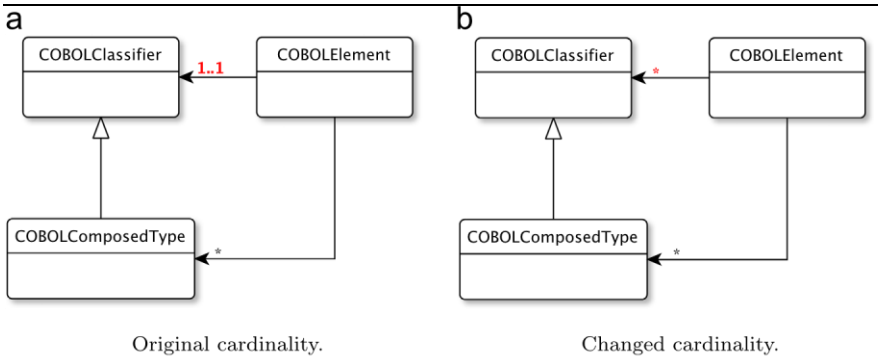

Fig. 6. Example of a corner case scenario.

This multiplicity constraint are included in the list of suggested types decreasing the savings effort figures presented in the results of the experiment.

VI. RELATED WORK VARIOUS RESEARCHERS

advocate the need for bottom-up or example-driven metamodelling. In [18], the authors aregue over the usefulness of bottom-up metamodeling and they identify a set of challenges in using such an approach. Their hypothesis is that bottom-up metamodeling 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 metamodeling 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[i] 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 boared. 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 modelling language with first class support for feature modelling. In [23], the authors use Clafer for example-driven modelling. 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 derive a set of complete model. To achieve this, the approach takes advantage of Clafer's support for variability modelling. Compared to this work, our approach is more generic, since it does not depend on a dedicated modelling environment. MARES [24] is another system, which supports metamodel inference. However, the objective of MARES is not to support example-driven metamodeling, 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 grammare inference techniques on the new representation. The approaches presented so fare 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 modelling 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 modelling) 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 metamodeling is only one aspect of the collaboration between domain-experts and language engineers during the development of DSLs. In [26] the Collabra approach is proposed. Collabra 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. [28–31]) 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 summarizes 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 paretiality 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 paretiality is close to the one of [34,35]. More pareticularely, in [34] the authors propose a diagrammatic, declareative approach to paretial model completion based on category theory. In this approach rewriting of paretial models is used to support both addition and deletion of models elements. Similarely, in [35] the authors use Constraint Logic Programming (CLP) to assign appropriate values for every missing property in the paretial 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 paretial 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 earely stages of language engineering in a bottom-up metamodelling context. Aparet from model completion, CLP is applied in other MDE scenareios. 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 similare manner the UMLtoCSP [38] tool uses CLP for the formal verification of UML class diagrams.

VII. CONCLUSIONS AND FUTURE WORK

In this paper a unique approach to tackle the matter of sort omissions within the rising domain of versatile MDE is planned. a lot of specifically, in our approach partial model examples and their incidental metamodel area unit translated into a collection of constraints, and so the missing typewriting info area unit derived by finding the constraint satisfaction downside. The planned approach was evaluated by running associate degree experiment on a corpus of fourteen,000 indiscriminately generated example models. we tend to outlined a metric that measures the area of attainable sorts for every node, showing vital improvement within the effort required to fill the categories of the missing nodes. At this time our approach returns a collection of equally probable sort suggestions. within the future we might prefer to scale back even any the dimensions of this set, by proposing the foremost probable sorts. this may be drained vareiety of the way. Firstly, by taking under

consideration the names of various parts, like the name of the attributes of the nodes within the example model and also the metamodel. within the future, we tend to commit to embrace this info to assist within the direction of sorting the prompt sorts by victimisation string distance/similarity metrics to spot attainable matches or synonyms. Secondly, the work administrated in [10,11] may even be combined with this work to boost {the sort the sort the kind} suggestion because the type came back from those approaches are often placed at the highest of the list of the prompt sorts. additionally, the Flexisketch's sort reasoning approach planned in [5] and relies on the graphical similarity between the nodes are often additionally wont to any scale back or type the set of candidate sorts created as a results of our planned approach. Moreover, the meta BUP tool bestowed in [1] may well be combined with our planned approach for the semi-automatic reasoning of the draft metamodel. Finally, a protracted term goal is that the application of the planned approach on real example models that area unit a results of a versatile MDE approach applied on a whole project. this fashion we'll be able to valuate the planned approach not solely on one increment, however additionally in additional than one iterations that ordinareily occur in versatile MDE approaches.

REFERENCES

[1] Kolovos DS, Matragkas N, Rodríguez HH, Paige RF. Programmatic muddle management. In: XM 2013—extreme modeling workshop.

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

[3] Wüest D, Seyff N, Glinz M. Flexisketch: a mobile sketching tool for software modeling. In: Mobile computing, applications, and services. Seattle, Washington, USA: Springer; 2013. p. 225–44.

[4] Bak K, Zayan D, Czarenecki K, Antkiewicz M, Diskin Z, Wasowski A, et al. Example-driven modeling: model ¼ abstractions þ examples. In: Proceedings of the 2013 international conference on software engineering.

[5] San Fransisco, California, USA: IEEE Press; 2013. p. 1273–6. Cuadrado JS, de Larea J, Guerra E. Bottom-up meta-modelling: an interactive approach. In: MODELS'12: ACM/IEEE 15th international conference on model driven engineering languages and systems, Lecture notes in computer science, vol. 7590. Innsbruck, Austria: Springer; 2012. p. 3–19.

[6] Kuhrmann M. User assistance during domain-specific language design. In: FlexiTools workshop; 2011.

[7] Paige RF, Kolovos DS, Rose LM, Drivalos N, Polack FA. 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. Potsdam, Germany: IEEE; 2009. p. 162–71.

[8] Zolotas A, Matragkas N, Devlin S, Kolovos D, Paige R. Type inference in flexible model-driven engineering. In: Taentzer G, Bordeleau F. (Eds.), Modelling foundations and applications, Lecture Notes in Computer Science, vol. 9153, L' Aquila, Italy: Springer International Publishing, 2015, p. 75–91.

[9] Zolotas A, Matragkas N, Devlin S, Kolovos DS, Paige RF. Type inference using concrete syntax properties in flexible model-driven engineering. In: 1st Flexible model-driven engineering workshop. Reiter R. On closed world data bases. In: Logic and data bases. Springer; 1978. p. 55–76.

[10] Apt KR, Wallace M. Constraint logic programming using ECLiPSe. New York, NY, USA: Cambridge University Press; 2006.

[11] Williams JR, Paige RF, Kolovos DS, Polack FA. Searech-based model driven engineering, Technical report. YCS-2012-475, Deparetment of Computer Science, University of York; 2012. Prosser P. An empirical study of phase transitions in binarey constraint satisfaction problems. Aretif Intell 1996:81–109.

[12] Mitchell DG. Resolution complexity of random constraints. In: Proceedings of the 8th international conference on principles and practice of constraint programming (CP'2002).

[13] Berlin, Heidelberg: Springer; 2002. p. 295–310. Gao Y, Culberson J. Resolution complexity of random constraint satisfaction problems: another half of the story. Discrete Appl Math 2005;153(13): 124–40. Cho H, Sun Y, Gray J, White J. Key challenges for modeling language creation by demonstration. In: 2011 icse workshop on flexible modeling tools, Honolulu HI; 2011.

[14] Dillenbourg P. What do you mean by collaborative learening. In: Collaborative-learening: cognitive and computational approaches, vol. 1; 1999. p. 1–15.

[15] Cho H, Gray J, Syriani E. Creating visual domain-specific modeling languages from end-user demonstration. In: 2012 ICSE workshop on modeling in software engineering (MISE). IEEE; 2012. p. 22–8.