Computer Languages, Systems & Structures ∎ (∎∎∎∎) ∎∎∎–∎∎∎

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

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*

A R T I C L E   I N F O

Article history:
Received 14 February 2016
Received in revised form
10 December 2016
Accepted 11 December 2016

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

A B S T R A C T

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.

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

* Corresponding author.
E-mail addresses: amz502@york.ac.uk (A. Zolotas), rclariso@uoc.edu (R. Clarisó), n.matragkas@hull.ac.uk (N. Matragkas), dimitris.kolovos@york.ac.uk (D.S. Kolovos), richard.paige@york.ac.uk (R.F. Paige).

http://dx.doi.org/10.1016/j.cl.2016.12.002
1477-8424/© 2016 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

issue, flexible modelling approaches have been proposed in the literature (e.g. [3–5,1]). 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. Fig. 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]).

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 characterizes 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 Fig. 1). The way that the draft

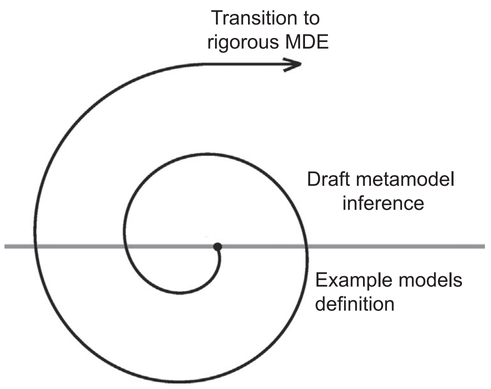
Transition to rigorous MDE
Draft metamodel inference
Example models definition

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