Developing eMoflon with eMoflon

Erhan Leblebici[1,⋆], Anthony Anjorin[1,⋆⋆], and Andy Schürr[2]

[1] Graduate School of Computational Engineering, Technische Universität Darmstadt
{leblebici,anjorin}@gsc.tu-darmstadt.de
[2] Real-Time Systems Lab., Technische Universität Darmstadt
andy.schuerr@es.tu-darmstadt.de

Abstract. eMoflon is a Model-Driven Engineering (MDE) tool that supports rule-based unidirectional and bidirectional model transformation. eMoflon is not only being used successfully for both industrial case studies and in academic research projects, but is also consequently used to develop itself. This is known as *bootstrapping* and has become an important test, proof-of-concept, and success story for us. Interestingly, although MDE technologies are inherently self-descriptive and higher-order, very few actively developed MDE tools are bootstrapped. In this paper, we (i) report on the current state and focus of eMoflon, (ii) share our experience with bootstrapping in an MDE context, and (iii) provide a scalability analysis of a core component in eMoflon implemented as both a unidirectional and bidirectional model transformation with eMoflon.

Keywords: eMoflon, MDE, model transformation, bootstrapping.

1 Introduction and Motivation

eMoflon[1] is a graph transformation tool that supports the rule-based specification of model transformations, which play a central role in Model-Driven Engineering (MDE). eMoflon builds upon the Eclipse Modelling Framework (EMF), using *Ecore* for metamodelling, *Story Driven Modelling* (SDM) [3] (a dialect of programmed graph transformations) for unidirectional model transformation, and *Triple Graph Grammars* (TGGs) [6] for bidirectional model transformation. eMoflon consists of an Eclipse plugin as backend, and two frontends: a set of Eclipse-based editors supporting a *textual* syntax, and a plugin for Enterprise Architect (EA), a professional UML tool, supporting a *visual* syntax.

Besides industrial case studies and academic research projects, an important proof-of-concept for eMoflon is its own self-development. This is often referred to as *bootstrapping* and will be used in the rest of this paper to present the main features supported by eMoflon. Figure 1 depicts a schematic overview of the chain of model transformations employed internally by eMoflon.

⋆ Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.

⋆⋆ The project on which this paper is based was funded by the German Federal Ministry of Education and Research, funding code 01IS12054. The authors are responsible for all contents.

[1] www.emoflon.org

D. Di Ruscio and D. Varró (Eds.): ICMT 2014, LNCS 8568, pp. 138–145, 2014.

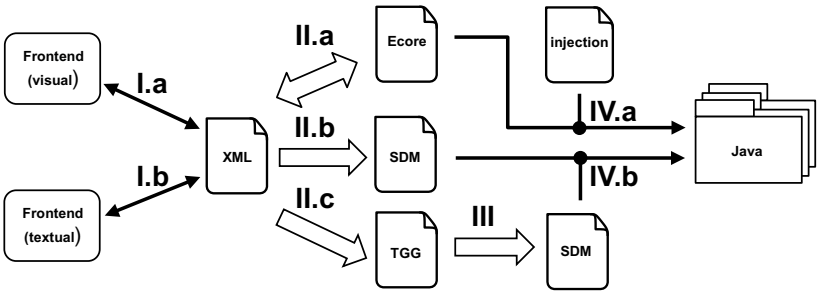

Fig. 1. An overview of the main model transformations used in eMoflon

Ecore, SDM and TGG models are specified in either a visual or textual concrete syntax using the respective frontend. The first step in the chain (marked as I.a, I.b in Fig. 1) maps the frontend-specific representation to and from a common, frontend-independent XML tree structure. This is realized with C# code in the case of EA, and with standard (un)parsers in the case of our textual syntax. The tree structure is used as a generic exchange format decoupling the backend from its different frontends. It is kept as simple as possible to shift the complexity of the transformation to the subsequent steps in the chain.

The second step (marked as II.a, II.b, and II.c. in Fig. 1) is to transform the generic tree structure to actual instances of our Ecore, SDM, and TGG metamodels. These transformations are bootstrapped (depicted as bold white arrows) meaning that they are implemented with eMoflon itself. The transformation II.a is bidirectional to enable importing external Ecore instance models (e.g., as provided by the Transformation Tool Contest[2]). A unidirectional version of II.a is also available in the XML-Ecore direction with SDM, as support for SDM in eMoflon was implemented *before* TGGs. The two versions of II.a provide for an interesting qualitative and quantitative comparison of SDM and TGGs, and we shall use excerpts of transformation II.a as our running example throughout the paper. The transformations II.b and II.c transform a tree structure to SDM and TGGs, respectively. These transformations are currently unidirectional, but bidirectionalizing them is work in progress as it would, for example, enable transforming generated models (result of III) back into the respective concrete syntax.

TGGs are *operationalized* by *compiling* them to SDM with the transformation III, which is bootstrapped with SDMs as a unidirectional model transformation. Bidirectionality is not absolutely necessary in this case as the SDM generated from a TGG represents low-level operationalization details and is not an artifact meant for further user adjustments. Finally, unidirectional model-to-text transformations IV.a and IV.b generate Java projects from Ecore and SDM, with the option of *injecting* hand-crafted (Java) code into the generated files.

In this paper, our contribution is to share and discuss our experience of bootstrapping in an MDE context. For this, we use excerpts from the import/export mechanism of eMoflon as our running example, which is developed with SDMs

[2] `http://www.transformation-tool-contest.eu/`

and TGGs in two different versions. We also provide a scalability comparison of these two versions. That is of particular interest in the context of bootstrapping eMoflon. The rest of the paper is structured as follows: Section 2 introduces eMoflon's support for metamodelling with Ecore. Support for unidirectional (SDMs) and bidirectional (TGGs) model transformation is presented in Sect. 3 and 4, respectively, together with an evaluation of runtime scalability in Sect. 5. Bootstrapping transformation tools in general, and eMoflon in particular, is discussed in Sect. 6 together with related work. Sect. 7 states our future focus and concludes the paper.

2 Metamodelling with Ecore

eMoflon supports Ecore-conform metamodelling used to specify the data structures to be manipulated with model transformations. An excerpt of the metamodel used to represent the generic exchange format in eMoflon is depicted as a class diagram to the left of Fig. 2, consisting basically of labelled Nodes with children and Attributes. To demonstrate how this tree structure is used, the tree metamodel itself is represented as a generic tree to the right of Fig. 2 (as an object diagram). Only the tree structure for representing `EClasses` and `EReferences` is shown, i.e., `EAttributes` as well as multiplicities and containment are omitted. The `EClasses` "Node" and "Attribute" are represented as nodes in the tree labelled as "EClass" with attributes for their names and a global ID used for cross references in the tree. `EReferences` are represented analogously, placed in the tree as children of a "references" node of the respective "EClass" node.

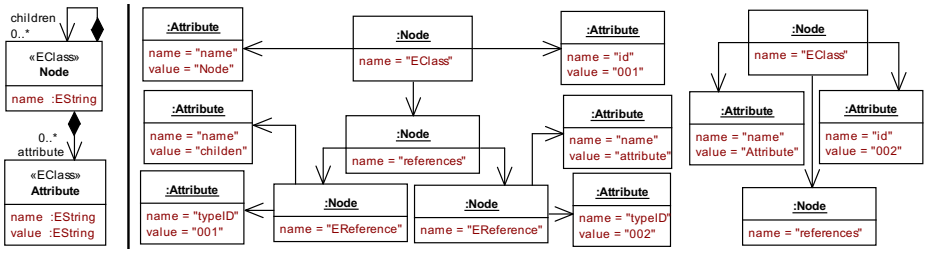

Fig. 2. Metamodel used as an exchange format and its representation as a generic tree

3 Unidirectional Model Transformations with SDM

Story Driven Modelling (SDM) [3] is used in eMoflon to specify unidirectional model transformation. SDM combines graph patterns with control flow structures consisting of a start node, connected activity nodes, and stop nodes. Figure 3 depicts the SDM `handleReferences` that transforms the tree structure representing a reference to an actual instance of `EReference` in Ecore. The SDM, simplified for presentation purposes, takes a related node and class (`classNode` and `eClass`) as parameters, and consists of two activity nodes.

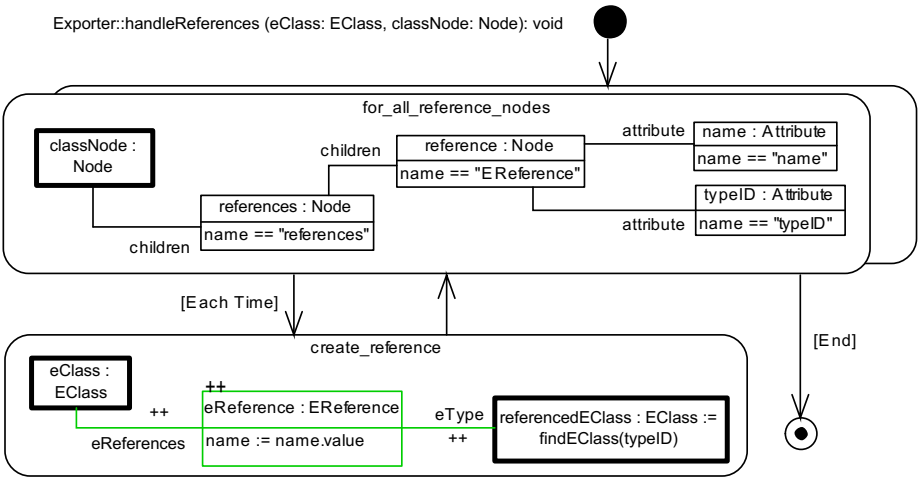

Fig. 3. SDM for exporting references of an EClass

Starting with a *for-each* activity node (`for_all_reference_nodes`) that determines *all* occurrences of the specified pattern in the tree, the SDM iterates over all subtree structures that represent references in the given root node `classNode`. Fixed elements in the pattern such as `classNode` (bound to the given parameter) are depicted with a bold frame, while all other elements are determined via pattern matching, such that all constraints are satisfied (e.g., `name == "typeID"`). For each occurrence of the pattern, the SDM executes the second activity `create_reference`. This activity creates a new `EReference` (depicted green with a "++" markup) between `eClass`, fixed to the given parameter, and `referencedEClass`, determined by invoking a helper method that returns the class referenced by `typeID`. Binding an object over a method call (possibly with parameters as in our case) is a standard language feature in SDM as defined in [4]. Such helper methods can be implemented again with SDM or with plain Java (e.g., using a pre-filled hash table for efficiency reasons). This enables recursion and the integration of hand-crafted code in SDM.

4 Bidirectional Model Transformations with TGGs

Triple Graph Grammars (TGGs) [6] are a declarative, rule-based technique to specify bidirectional model transformation. A TGG is a set of rules that describe how consistent *triples* of source and target models (*graphs*), connected by a correspondence model, are built up simultaneously. All *operational* transformations such as forward, backward and update propagation, are automatically derived from the single specification. In the following, the same transformation implemented with SDMs in Fig. 3, i.e., handling references in the tree, is presented as a TGG rule (depicted in Fig. 4). Black elements represent the pre-condition of the rule, i.e., an occurrence of these elements must be found in order to apply the

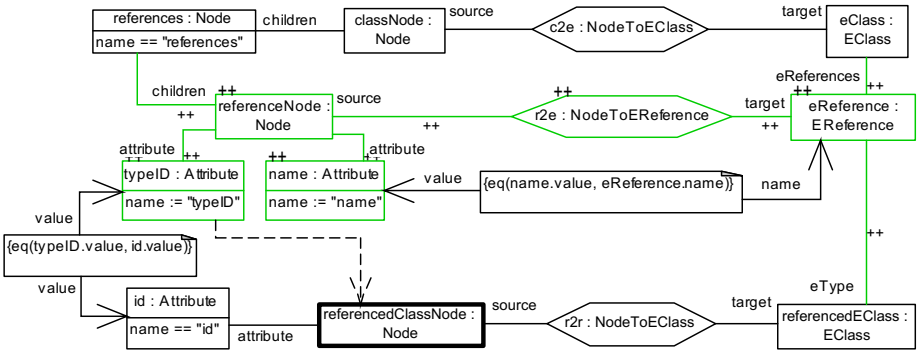

Fig. 4. TGG rule for handling references

rule. Green elements with a ++ markup state the post-condition that must hold after the rule has been applied. The rule, therefore, states that an `EReference` is created together with the depicted subtree structure. TGGs are declarative in the sense that no explicit control structure or rule dependencies are specified. The underlying algorithm figures out automatically the correct choice and sequence of rules to apply for each operational scenario. *Attribute constraints* such as `eq(name.value, eReference.name)` are specified with a bidirectional extensible textual constraint language, and ensure that eReference is named correctly using the appropriate attribute in the tree, and that referenceNode has the correct typeID value corresponding to the referenced class node in the tree.

In case of a forward transformation, the TGG rule in Fig. 4 is modified by adding all source elements to its context. This means that the required tree structure is "parsed" and only the correspondence link and the target elements are created when applying the rule. Unfortunately, finding the referenced class node might be very time-consuming as no direct connection exists from the reference node to the referenced class node in the tree. In the worst case, one must iterate over all class nodes in the tree to find the correct one. As an optimization technique for such cases, we propose *binding expressions* to *bind* an element directly from another via an auxiliary method, which can be implemented with SDM or plain Java. In our example, the binding expression (depicted as a dashed arrow in the rule) takes the type id attribute of the reference node as input and returns the referenced class node, which should have the same type id. Analogously to the helper method for the SDM (Fig. 3), this is realized in constant time as a table lookup in a lazy cache. Integrating such hand-crafted components seems to contradict the declarative nature of TGG rules, but they serve as a crucial and pragmatic means of dealing with performance issues at critical points.

In our example, a second rule is required to handle self-references and would only differ slightly from the rule depicted in Fig. 4. For such cases, eMoflon supports *rule refinements*, a modularity concept for TGGs. Using refinements, an *abstract* rule covering the commonalities of both rules can be specified and refined

appropriately in the concrete rules. Rule refinement avoids pattern duplication and greatly improves the readability and maintainability of TGG specifications.

5 Scalability

The plots on the left and right side of Fig. 5 show our runtime measurements in linear and logarithmic scale, respectively, for the import with TGGs and the export with TGGs and SDM. The y-axis shows the time in seconds, the x-axis the number of elements of randomly generated Ecore models. Vertical dashed lines indicate a change in step size in the x-axis. The logarithmic plot shows two additional measurement points for very large models containing up to 300.000 elements. All measurements were repeated 10 times (the median is plotted) and executed on an Intel i5-3550 (3.30 GHz) processor with 8 GB RAM running Windows 7 and Eclipse 4.3.

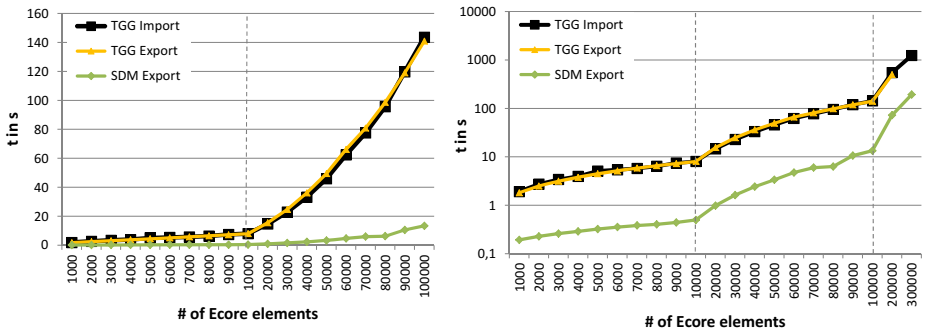

Fig. 5. Runtime measurements in linear and logarithmic scale

Our TGG algorithm is in theory polynomial with respect to model size, and our results back this claim (showing even almost linear behaviour for up to 10.000 elements). Our results also show that both directions (import and export) exhibit very similar runtime behaviour, reflecting the bidirectional and symmetrical nature of TGGs. On the other hand, the TGG-based transformations are 10-15 times slower than the SDM implementation and run out of memory as from 200.000 elements for the export, and 300.000 for the import (this difference is due to the tree being much larger than the corresponding Ecore model).

6 Discussion and Related Work

MDE technologies are inherently self-descriptive and higher-order but, to the best of our knowledge, very few model transformation tools are actually developed with bootstrapping. ATL [5] and FUJABA [3], however, are examples for tools/toolsuites that do practice bootstrapping. Although the bootstrapped FUJABA code generator CodeGen2 is actually reused in eMoflon to generate

Java code from SDMs, it is only used as a well-tested black-box component and is no longer bootstrapped. Figure 1 in the introduction reflects our pragmatic decision on what is to be bootstrapped in eMoflon after considering our current research foci and the advantages/challenges of bootstrapping.

Bootstrapping is a common technique in compiler construction for General Purpose Languages (GPLs) such as C++. SDM and TGGs, however, are Domain Specific Languages (DSLs) for model transformation, and cannot replace a GPL. Nevertheless, we are convinced that it is just as advantageous to use such transformation languages for defining suitable parts of their compilers. Barzdins et al. [1] demonstrate this by obtaining model transformation languages from existing ones via bootstrapping. A transformation language L_i is compiled to a lower-level language L_{i-1} with a compiler written in L_{i-1}. This corresponds to TGGs being compiled to SDMs with SDMs (cf. Fig. 1). In addition to their arguments for usability and efficiency of bootstrapped languages, our experience shows the following advantages: (i) the tool itself is a non-trivial test that cannot be skipped, (ii) a proof-of-concept is established regarding the capabilities of the developed transformation languages, and (iii) both functional and non-functional requirements are equally considered due to intensive self-usage. Regarding the last point, language-related features such as binding expressions and modularity concepts (cf. Sect. 4), as well as non-functional qualities such as user-friendliness and performance are constantly being improved on the basis of our self-usage experience.

Buchmann et al. [2] challenge the added value of graph-based model transformations in general and SDMs in particular, referring to the bootstrapping of CodeGen2. Some of the drawbacks they identify are indeed relevant for our bootstrapping, including a lack of means for low-level details such as exception handling, and missing modularity concepts for patterns. Moreover, our experience reveals further challenges of bootstrapping with SDMs: (i) increased complexity when making changes as they must be tested before and after building a new version of the tool, (ii) an increased dependency on underlying code generators and their shortcomings, and (iii) redundant implementations of components (initial versions with Java, later versions with SDMs, and in some cases finally with TGGs).

7 Conclusion and Future Focus

In this paper, we have reported on the current state of eMoflon, conducted a scalability analysis of a core component in eMoflon implemented with eMoflon, and shared our experience with bootstrapping. For the future, the focus of TGGs in eMoflon will be *synchronization* of concurrently changed models, a special case of model transformation where models are no longer created from scratch, but are updated incrementally to reflect the changes. Moreover, work on a new pattern matching engine is in progress to replace CodeGen2 and improve the code generation capabilities of eMoflon and, therefore, our development experience.

References

1. Barzdins, J., Kalnins, A., Rencis, E., Rikacovs, S.: Model Transformation Languages and their Implementation by Bootstrapping Method. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 130–145. Springer, Heidelberg (2008)
2. Buchmann, T., Westfechtel, B., Winetzhammer, S.: The Added Value of Programmed Graph Transformations A Case Study from Software Configuration Management. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 198–209. Springer, Heidelberg (2012)
3. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Transformations. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
4. Heinzemann, C., Rieke, J., Detten, M.V., Travkin, D., Lauder, M.: A new Meta-Model for Story Diagrams. In: 8th International Fujaba Days, pp. 2–6 (2011)
5. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
6. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) Graph-Theoretic Concepts in Computer Science. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)