Elementary Model Management Patterns

Sahar Kokaly[1], Zinovy Diskin[1,2], Tom Maibaum[1], Hamid Gholizadeh[1]

[1] NECSIS, McMaster University, Canada
{kokalys|diskinz|maibaum|mohammh}@mcmaster.ca
[2] University of Waterloo, Canada
zdiskin@gsd.uwaterloo.ca

1 Introduction

Design patterns appear in many areas of software engineering including Object Oriented (OO) design [5] and workflow languages [1], and can be used for model management (MMt) as well. Indeed, MMt tools address common issues, whose solutions are frequently reinvented for each tool. Platform specific solutions to common problems may vary by language, however, the high-level specifications could be the same. Design patterns seek to communicate these common specifications in an abstract way that enables reuse, improves understanding, and eases communication.

Although some work has been suggested in the literature relating to design patterns for MMt, most of it has focused on model transformation (MT) design patterns, and has been specific to certain MT languages. In a notable work [4], the authors propose a *platform independent* language for describing patterns for graph-based MTs. In [6], the author discusses transformation patterns for refactoring/improving/implementing OO designs. In our earlier work [2], we presented a library of abstract design patterns for MMt based on mathematical foundations. Rather than suggesting patterns for entire MMt scenarios (as is done for OO design in [5] and for MT in [4]), we proposed elementary building-block-patterns, from which complex patterns for MMt scenarios can be composed. The goal of this short paper is to introduce the library to the PAME community, and present it in a condensed and simplified way (Section 2). How our elementary patterns are composed into complex scenarios is shown by an example in Section 3.

2 Elementary Design Patterns

In our presented design patterns, mappings are made explicit and are defined as sets of links (block arrows) between model elements rather than single links between models. Moreover, the algebraic operations (appearing on chevrons) in the patterns generate models as well as explicit traceability maps.

Patterns 1-3 present a given configuration, and we call them *static*. In these patterns we distinguish between the information that is given (shown in black) and that being provided by a user or some heuristics (shown in green). In contrast, patterns 4-8 show operations, which take a configuration of models and mappings as their input (shown by shaded elements that are either black or green), and output a configuration of automatically computed models and mappings (blank blue elements). We call these patterns *dynamic*. Amongst them, patterns 4-6 are for source-to-target MTs, and patterns 7,8 are for the two types of model merge operations.

Pattern 1: Typing Mapping. A model is a *total typing mapping* (e.g., t_1 in Fig.1 from a model's *data* graph (D_1) to the model's *metadata* graph (MM_1) (ovals in Fig.1). Typing must satisfy the constraints (C_1) declared in the metamodel (we write $t_1 \models C_1$). Three vertical arrows in Fig.1 specify three models.

Pattern 2: Model Mapping. A *model mapping* (e.g., f_1 in Fig.1) is a pair (f_{1D}, f_{1MM}) of total *correspondence* mappings between the respective data and metadata parts of the models. Together with the respective typing mappings, they form a commutative square of graph mappings. E.g., in Fig.1, we have two commutative squares (note the marker [=]) specifying two model mappings: $f_i : M_0 \to M_i, i = 1, 2$.

Pattern 3: Model Overlap. An *overlap* of two models is a *span* of model mappings. E.g, Fig.1 specifies a span (f_1, f_2) between models M_1 and M_2. Model MM_o, the head of the span, comprises correspondence links between models, and mappings $f_i, i = 1, 2$ point to the ends of the links.

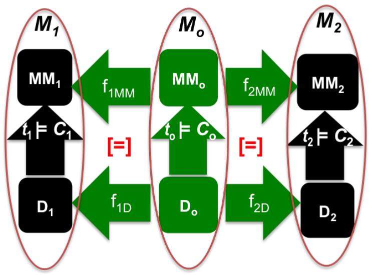

Fig. 1. Patterns 1,2,3

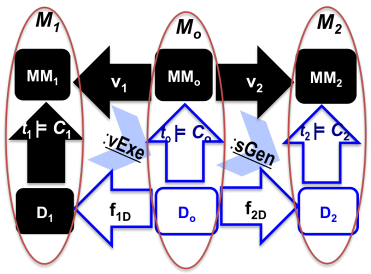

Fig. 2. Patterns 4,5,6

Pattern 4: Descriptive Views. A *view definition* is a metamodel mapping $v_1 : MM_1 \leftarrow MM_o$ (see Fig.2). Its execution for model M_1 is given by invoking the operation `vExe` (note the chevron), which takes instances of MM_1 and outputs instances of MM_o.

Pattern 5: Prescriptive Views. Implementation of an instance of metamodel MM_o within a platform specified by MM_2 is an operation opposite to view execution (see Fig.2), and unfolds over a view definition mapping $v_2 : MM_o \to MM_2$. We call this operation *source generation* `sGen`. As view models (e.g., M_o) usually contain less information than source models (e.g., M_2), a policy is required to choose a unique implementation amongst all possibilities.

Pattern 6: Model Transformation. A *model transformation definition* is, in general, a span (v_1, v_2) of metamodel mappings, which can be executed in both directions. Fig.2 shows execution of the span in the direction from left to right, which transforms model M_1 into model M_2. The span can be also executed in the opposite direction with `vExe` for v_2 and `sGen` over v_1. View execution and source generation are two special cases, when either the right or the left "leg" of the transformation definition span is the identity mapping.

Pattern 7: Additive Model Merge. An *Additive Merge* is the result of executing a *colimit* operation on two models and their correspondence span, resulting in a model together with two embedding mappings $e_i, i = 1, 2$ (as seen in Fig.3), whose inverse mappings provide traceability. This operation accurately merges elements of the models without redundancy: if an element x from model M_1 and y from model M_2 are linked in the correspondence span O, only one element appears in the colimit $M_1 \vee M_2$. Real model merge typically requires some post-processing for conflict resolution and normalization (see [2] for details).

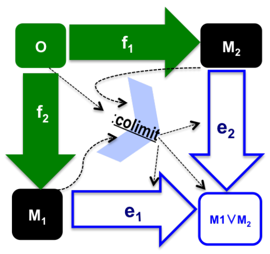

Fig. 3. Pattern 7

Pattern 8: Multiplicative Model Merge. A *Multiplicative Merge* is the result of executing a *limit* operation on two models and their correspondence *cospan*, resulting in a model together with two *traceability mappings* (see Fig.4) to each of the input models. This pattern appears in scenarios that involve model parallel composition, intersection or join. Details can be found in [3].

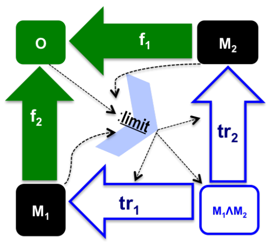

Fig. 4. Pattern 8

3 Example

Consider the following MMt scenario: two models $M1$,$M2$ are matched, then merged, and the merge is translated into code. In our framework, the workflow is encoded as a composition of operations shown in Fig.(5). Semi-automatic (green) operation `Match` produces the correspondence span O, operation `Merge` (comprising colimit with some post-processing not shown in the figure) results in a triple $(M3, e1, e2)$, and operation `Transf` (whose definition is not shown) produces $M4$ with traceability mapping tr. Finally, applying operation `limit` to the cospan $(e1, tr)$ outputs M_{14}; the part of $M4$ that depends on $M1$.

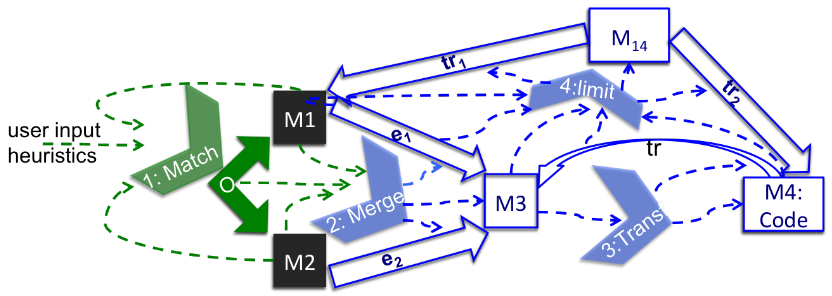

Fig. 5. Workflow example.

4 Concluding Remarks

As an exercise, we encourage the workshop participants to consider the MDE design patterns they proposed in light of the patterns we presented here.

References

1. van der Aalst, W.M.P.: Workflow patterns. In: Encyclopedia of Database Systems, pp. 3557–3558 (2009)
2. Diskin, Z., Kokaly, S., Maibaum, T.: Mapping-Aware Megamodeling: Design Patterns and Laws. In: SLE. pp. 322–343 (2013)
3. Diskin, Z., Maibaum, T.: A model management imperative: being graphical is not sufficient, you have to be categorical. (To appear in ECMFA'15)
4. Ergin, H., Syriani, E.: Towards a language for graph-based model transformation design patterns. In: Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, York, UK, July 21-22, 2014. pp. 91–105 (2014)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Education (1994)
6. Lano, K., Kolahdouz-Rahimi, S.: Design patterns for model transformations. In: ICSEA 2011, The Sixth International Conference on Software Engineering Advances. pp. 263–268 (2011)