Towards Mapping the Semantics of References between Metamodels[4]

Thomas Reiter[1], Werner Retschitzegger[2]

Information Systems Group (IFS)
Johannes Kepler University Linz, Austria

Kerstin Altmanninger[3]

Department of Telecooperation (TK)
Johannes Kepler University Linz, Austria

Abstract

In model driven development, model transformation languages play an important role as a way to specify and execute transformations on models. Methods and processes that can guide the development of transformations, however, still have to be formulated. In this paper we take a first step towards describing such a process, which is built on a step-wise establishment of mappings between metamodels. In particular, we focus on the mapping of references, for which we propose an initial set of mapping constructs, that allow taking into account certain semantics of references. Consequently, such mappings enable to specify complex relationships between references in different metamodels, which is an important step towards the derivation of model transformations implementing these mappings.

Keywords: model transformation, semantic mapping, mapping language, metamodel matching.

1 Introduction

The emergence of model driven development leads to the increase of different model transformation languages and systems. The different approaches towards building transformation languages can basically be separated into graph-rewriting and transformation approaches [5], as well as responses to OMG's QVT RFP [8][1].

Although these approaches may differ in the way how transformations are specified or how they are actually executed, the basic commonality they share, is that transformations are specified on the metamodel level and carried out on the model

[1] Email:reiter@ifs.uni-linz.ac.at

[2] Email:werner@ifs.uni-linz.ac.at

[3] Email:kerstin@tk.uni-linz.ac.at

[4] This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.

level. Analogous to traditional general purpose programming languages, the mentioned model transformation languages basically allow for arbitrary manipulations upon models. In model driven development in general, however, several different kinds of transformations can be characterized, such as horizontal (PIM-to-PIM) or vertical (PIM-to-PSM) transformations in MDA, as well as more specific transformation scenarios like the merging of models or code generation. Compared to traditional programming, for various kinds of problems, different software engineering principles, heuristics and patterns [6] have been established that guide a developer towards solving a programming task, e.g. step-wise refinement [10].

In our opinion, similar methods that guide the development of transformations still have to be formulated. Hence, we sketch a view of how such a method may look like, specifically for the task of developing transformations that facilitate a translation between two different metamodels by mapping semantically equal concepts of one metamodel onto concepts of another metamodel. By mapping semantically equal concepts, we particularly refer not just to the mapping of classes, but to the mapping of references representing relationships between these classes, too.

The remainder of this paper is structured as follows. Section 2 introduces two transformation examples and illustrates the proposed mapping approach. Section 3 generalizes the insights gained in the examples and lays out various constructs for the mapping of references. Section 4 discusses related work and section 5 concludes with an outlook on future work.

2 Matching References

This section introduces two transformation scenarios which will serve as running examples throughout this paper and illustrate the development of two transformations through a mapping of classes and references. The metamodels chosen in these examples represent two frequently encountered transformation situations, namely a transformations of flows (e.g., directed graphs), and the transformation of containers and elements (e.g.; arrays or lists). In particular, the first example deals with the transformation between metamodels representing flows, thus necessitating a not straight forward graph traversal. The second example deals with a transformation between structurally equal metamodels that however represent differently structured models. Hence, these examples allow to show up transformations problems which are typically linked to the correct understanding of the semantics of references in metamodels. Many difficulties with transformation development are associated with correctly reifying the semantics of references of the source metamodel in the target metamodel. The process laid out in the following, specifically takes into account the semantics of references, and allows to establish mappings between references denoting their relationship to each other.

We approach the development of transformations in two steps. The first step deals with the mapping of semantically equivalent classes. Such a mapping could be supported by schema or ontology matching tools [4]. Those tools, however, are mainly built on discovering naming similarities and simple structural relationships, and aim at integrating large, but rather similar schemas and ontologies. Using such matchers on metamodels can yield undesirable results [7]. Hence, in our opinion

the matching of metamodels representing domain-specific languages poses a considerably different challenge, and basically relies on a user's domain knowledge, which lets one intuitively find corresponding concepts. Such a mapping between classes is assumed as a prerequisite in the following examples and should express semantic equivalence, for instance resulting in model transformation code causing the creation of a target model element for every source model element. The second step in transformation development deals with the correct setting of references in the target metamodel, and is often not a trivial task. Hence, we aim at offering support through a process guiding the step-wise discovery of a mapping between references based on an initial class mapping between metamodels. Thus, in a semi-automatic fashion, for every reference in the RHS metamodel, a most likely mapping towards references in the LHS metamodel can be proposed to the user. Heuristics (e.g., suggest shortest paths first) can be used to rank likely matches. Finally, a mapping expressing an often not straight forward relationship between references is established, which consequently can be used to derive the actual implementation of a transformation. The following examples will detail the above described process for the mapping of references.

2.1 Activity Diagram to Project Plan

The setting for the first example is depicted in Fig. 1, which shows a metamodel for *activity diagrams* (AD) as the left-hand side (LHS) metamodel, and a metamodel for Gantt-chart *project plans* (PP) as the right-hand side (RHS) metamodel [2].

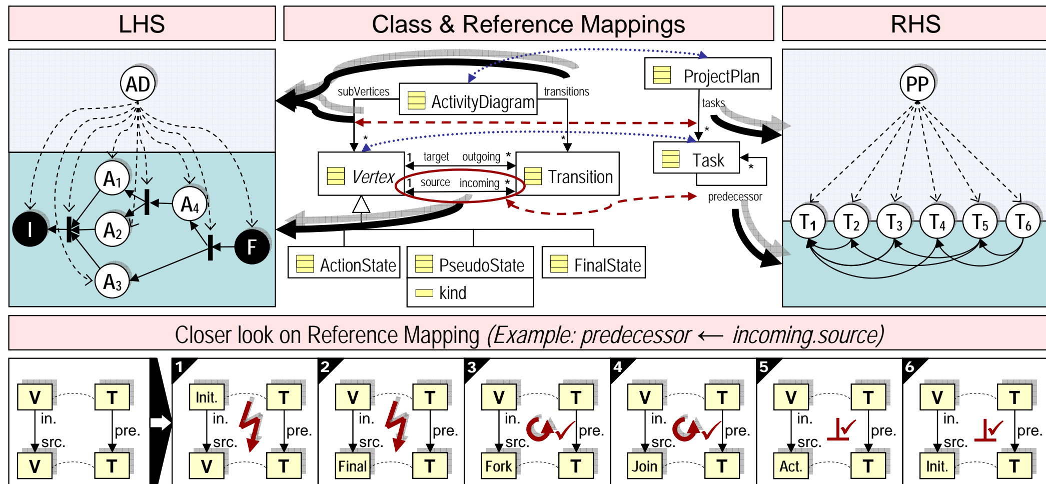

Fig. 1. Activity Diagram to Project Plan.

An *activity diagram* consists of *vertices* and *transitions* in between. *Vertex* is an abstract class with the concrete sub-classes *Action-*, *Final-* and *PseudoState*. The *kind* attribute in the *PseudoState* class differentiates *Fork-*, *Join-* and *InitialStates* [5].

[5] For reasons of simplicity we refer to a *PseudoState* with the *kind* attribute set to *'initial'*, *'fork'* or *'join'* as an *Initial-*, *Fork-* or *JoinState*.

A *ProjectPlan* consists of a number of *Tasks* and every *Task* has a *predecessor* reference to its previous *Task*.

Class Mapping. A mapping between classes denotes *StateMachine* and *ProjectPlan* to be semantically equivalent. Intuitively *ActionState* maps onto *Task*. We take a design decision and also map *FinalState* and *InitialState* onto *Task*. Control flow structures like *Fork*- or *JoinState* should however not be represented as a *Task*.

Reference Mapping. After the mapping of classes, one has to deal with finding a mapping for each reference in the target model. Firstly, as a most likely candidate, the *subVertices* reference is chosen to map onto the *tasks* reference. Secondly, the *predecessor* reference has to be mapped, which could be achived either by an *incoming.source* query as well as an *outgoing.target* query. Choosing the appropriate query lies in the responsibility of the user, who needs to interpret the word *"predecessor"* correctly. Once the decision for *incoming.source* is made, a straightforward assignment of *incoming.source* to *predecessor* would however be wrong. A correct setting of *predecessor* requires a traversal of the input model by repeatedly calling *incoming.source* to collect all *ActionStates* and *InitialStates* that are equivalent to the respective *Task* objects. The set of objects returned by the *incoming.source* query can contain *Fork*- or *JoinStates* which are not equivalent to *Task*, hence a repeated traversal of *incoming.source* is necessary. Due to the fact that *Vertex* is an abstract class, and not all of its subclasses map to *Task*, all possible instances of domains and ranges of the *incoming.source* query have to be considered in the way how they map onto *predecessor*. Fig. 1 shows how the discovery of a mapping towards the *predecessor* reference can go about, by specifying special relationships between all possible instances of *incoming.source* links.

First, we state that an *incoming.source* query originating from an *InitialState* (1) or going towards a *FinalState* (2) is invalid. An *incoming.source* query originating from a *Vertex* towards a *Fork*- or *JoinState* causes a repeated traversal of the query (3,4). *ActionStates* yielded by by an *incoming.source* query should be collected as a result without further traversal of *incoming.source* (5,6).

This information about the relationship between the *incoming.source* query and the *predecessor* reference can further be used to derive model transformations to actually implement the specified relationship, informally given as follows.

$$predecessor \leftarrow pred(start) = foreach(n \in start.incoming.source)\{p(n)\}$$

$$p(n)\begin{cases} break() & start.isTypeOf(InitialState) \vee n.isTypeOf(FinalState) \\ n & n.isTypeOf(InitialState) \vee n.isTypeOf(ActionState) \\ pred(n) & n.isTypeOf(ForkState) \vee n.isTypeOf(JoinState) \end{cases}$$

2.2 Priority Sequence to Priority Sequence.

The second example depicted in Fig. 2 shows a transformation between two metamodels representing sequences of prioritized tasks. The LHS metamodel has a *contains* reference indexing all tasks and a *next* reference where the first task of a given priority points directly to the first task of the next lower priority within the sequence. The RHS metamodels has a *successor* reference to impose an order on the tasks, similar to a linked list. The *priorityClasses* reference indexes the first

task in a given priority class. It is to note, that both metamodels are structurally equal and can represent the same state of affairs, but due to the different semantics of their references produce different object graphs.

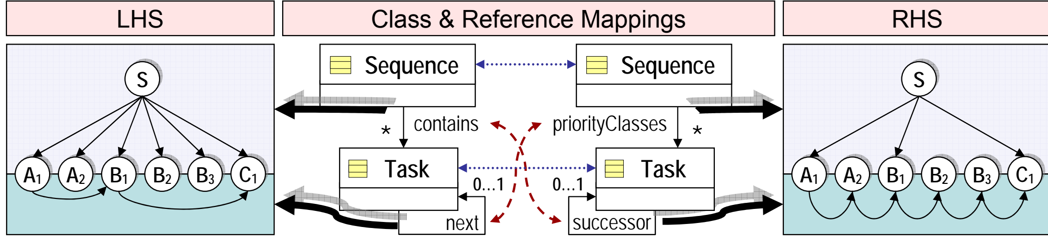

Fig. 2. Priority Sequence to Priority Sequence.

Class Mapping. An initial mapping of classes would denote the respective *Sequence* classes and the *Task* classes as equivalent.

Reference Mapping. In a first step, candidates for the mapping of references have to be found. Similar to the previous example this could be carried out by finding likely matches based upon the structure of references and classes in the context of an existing class mapping. This might seem trivial as the metamodels are structurally equal. Seemingly equivalent references like *contains* and *priorityClasses*, however, have very different semantics. The *contains* and *successor* references actually address the ordering of tasks, whereas the *next* and the *priorityClasses* reference serve to denote classes of priorities. Hence, the *successor* reference has to be mapped to a correctly indexed *contains* query, and *priorityClasses* is mapped onto the *contains* reference followed by a repeated traversal of the *next* reference. In this example, the indexing of references becomes of special importance, for traversing the correct *contains* links, as well as for correctly iterating the *nextPriority* reference. For instance, the resulting relationship for the priorities references is given as follows.

$$priorityClasses \leftarrow prior(n) = \begin{cases} contains[0] & \text{n=0} \\ prior(n-1).next & \text{n>0} \end{cases}$$

3 Mapping Constructs

The examples in the previous section show how important it is to be aware of the semantics of references, as references forming loops can necessitate an often counter-intuitive recursive traversal, and even transformations between structurally equal metamodels may not be as straightforward as it may seem at first look. This section builds upon the previously introduced examples and generalizes a number of mapping constructs to specify relationships between references of two metamodels. Similar to a mapping between classes, these constructs can denote special semantics for mappings between references, which are finally derived into model transformations.

We associate these mapping constructs with different patterns observable in the structure of references in the context of a mapping between classes. As depicted in Fig. 3 we differentiate three such patterns, dealing with the composition and

recursion of queries. Followed by that, a brief description and an overview of the mapping constructs is given.

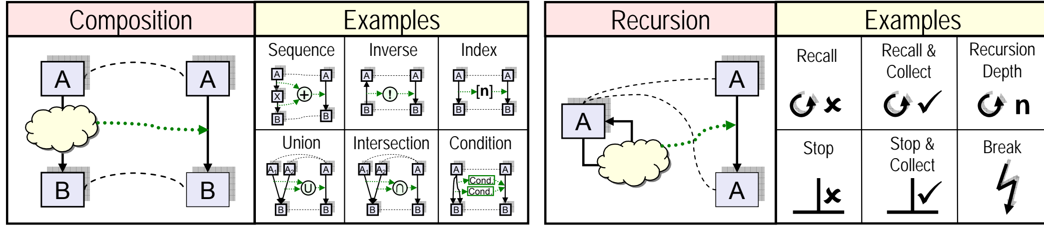

Fig. 3. Patterns categorized by the structure of references.

Composition. It is often the case that a reference on the RHS is equivalent to a query consisting of a sequence of references on the LHS. Furthermore, a RHS reference is often equivalent to a union or an intersection of queries on the LHS. Besides these basic navigational constructs, useful helper constructs could allow to specify that two references express inverse semantics (e.g., contains/containedBy) or use indexing mechanisms to express relationships between ordered references. The requirements for the activity diagram to project plan example could change and require that a *Task* is equivalent to *Vertices* except *Initial*- and *FinalStates*. Then, the *tasks* reference could be mapped onto a *subVertices* query (yields all *Vertices*) intersected with a *transitions.source* query (yields all *Vertices* without *FinalStates*) intersected with a *transitions.target* query (yields all *Vertices* without *InitialStates*), which would yield all *Vertices* except *Initial*- and *FinalStates*. A condition between references can denote a selection of a certain link, for instance when mapping from a higher cardinality to a lower cardinality reference. Furthermore, how a reference has to be set can differ in certain situations, which can be expressed by a conditional mapping between references. Referring to the activity diagram to project plan example for instance, one could decide not to set the *tasks* reference equivalent with the *subVertices* reference, but instead denote an equivalence with a *transitions.source* query to reference nodes in general, and an equivalence with a *transitions.target* query in case of the last node. This also shows that finding a matching reference is not always unambiguous, but often rather a matter of choice.
Recursion. References having the same domain (source class) and range (target class), either directly or through a composed query, often require a repeated invocation resulting in a traversal of the model. Such recursive references often occur in metamodels representing containment hierarchies or navigations in a flow. In such a case it is necessary to specify the query representing the recursion and conditions that will either cause a stop or a repeated traversal of the query, and what kind of queried model elements should join the result set (cf. Fig. 3). Furthermore, it can be useful to specify a certain maximum recursion depth or conditions causing a traversal's break off.

It has to be noted that we are not claiming completeness of the described patterns and the semantics we associated with mappings between references, as this paper focuses on laying out the idea of mapping references and fostering discussion about the establishment of a method that can guide the definition of transforma-

tions. Further patterns and mapping constructs could deal with situations involving cardinality restrictions and inheritance relationships.

4 Related Work

In [3] the idea of patterns and reusable idioms for the area of graph transformations has been introduced to foster the build up of pattern catalogues that transformation developers can rely on. Although it seems that our matching constructs for references can be used to instantiate these patterns, our work is more general as we rather focus on creating a semi-automatic process to guide transformation development.

As our approach to establish mappings between references in different metamodels is based on the mapping between classes, schema and ontology matching tools [4] are of relevance to our approach. To the best of our knowledge, however, these tools are not immediately usable for eliciting equivalent references in metamodels. For instance, the initial set of mapping constructs proposed in this paper, specifically takes into account recursive references, which are a frequently recurring pattern in model transformations and are therefore paid special attention.

The approach to finding the mapping of references between two metamodels can also be compared to the construction of wrappers in federated databases [9], which translate DB queries between different schemas. The final goals however differ, as wrappers aim at translating DB queries between schemas, whereas in our case, the knowledge of equivalent references serves our focus to finally support the implementation of model transformations between two metamodels.

5 Conclusion and Future Work

In this paper we have stated a position that encourages the importance of the mapping of references as a step in the definition of model transformations. We have laid out an initial set of mapping constructs to denote special semantics between references.

In our view, a structured approach discovering the mapping of classes and references can help the development of transformations between metamodels. An advantage of capturing mappings between classes and references in such a way, is that such specifications are declarative and basically technology independent, which can ease the porting of transformations between various model transformation languages and systems.

Future work will deal with extending our initial experiments to build a tool to support a user in finding equivalent references. Furthermore, we plan to detail the initial set of mapping constructs by specifically taking into account more complex patterns addressing cardinalities and inheritance relationships.

A final goal would be to come forward with a method and tool support that could guide the establishment of transformations between metamodels representing domain specific languages. Such a method would be based on an integrated matching tool for classes and references, and a mapping language to declaratively express complex relationships between references.

References

[1] Atl homepage. http://www.eclipse.org/gmt/atl/, 2006.

[2] Uml activity diagram to msproject. http://www.eclipse.org/gmt/atl/atlTransformations, 2006.

[3] Aditya Agrawal, Attila Vizhanyo, Zsolt Kalmar, Feng Shi, Anantha Narayanan, and Gabor Karsai. Reusable idioms and patterns in graph transformation languages. In T. Mens, A. Schrr, and G. Taentzer, editors, *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 181–192, October 2004.

[4] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with coma++. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 906–908, New York, NY, USA, 2005. ACM Press.

[5] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[7] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting metamodels to ontologies: A step to the semantic integration of modeling languages. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML), Genova, Italy*, October 2006.

[8] Object Management Group (OMG). Mof qvt final adopted specification.

[9] Philippe Thiran, Jean-Luc Hainaut, and Geert-Jan Houben. Database wrappers development: Towards automatic generation. volume 00, pages 207–216, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[10] Niklaus Wirth. Program development by stepwise refinement. In *Communications of the ACM, Vol. 14, No. 4, April 1971, pp. 221-227*. ACM, April 1971.