GRACE TECHNICAL REPORTS

An Algebraic Approach to Bidirectional Model Transformation

S. Hidaka Z. Hu H. Kato K. Nakano

GRACE-TR-2008–02 September 2008

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

An Algebraic Approach to Bidirectional Model Transformation

Soichiro Hidaka Zhenjiang Hu Hiroyuki Kato
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo 101-8430, Japan
{hidaka,hu,kato}@nii.ac.jp

Keisuke Nakano
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi
Tokyo 182-8585, Japan
ksk@cs.uec.ac.jp

Abstract

Bidirectional model transformation plays an important role in maintaining consistency between two models, and has many potential applications in software development, including model synchronization, round-trip engineering, software evolution, multiple-view software development, and reverse engineering. However, unclear bidirectional semantics, weak bidirectionalization method, and lack of systematic development framework are known problems that prevent it from being practically used. To remedy this situation, in this paper, we propose a novel algebraic framework for bidirectional model transformation, by integrating the state-of-the-art technologies on bidirectional tree transformations and algebraic graph querying. We make a significant extension from bidirectional tree transformation to bidirectional graph transformation, and give a powerful automatic bidirectionalization method to derive a backward graph transformation from a forward graph transformation. Moreover, we demonstrate how our algebraic framework can support systematic development of efficient large-scale bidirectional model transformations in a compositional manner. Our experimental results show promise of the new approach.

1 Introduction

Model transformation plays an important role in model-driven software development, which aims to introduce significant efficiency and rigor to the theory and practice of software development. In model-driven software development, models are the key artifacts in all phases of development, from system specification and analysis, to design and testing. The use of models and the application of model transformations open up new possibilities for creating, analyzing, and manipulating systems through various types of tools and languages.

Bidirectional model transformation [25, 2], being an enhancement of model transformation with bidirectional capability, is an important requirement on OMG's Queries/Views/Transformations (QVT) standard [22] recommended for defining model transformation languages. It describes not only a forward transformation from a source model to a target model, but also a backward transformation showing how to reflect the changes on the target model to the source model so that consistency between two models is maintained. Bidirectional model transformation has many potential applications in software development, including model synchronization [2, 27, 12], round-trip engineering [1], software evolution by keeping different models coherent to each other [20, 8], multiple-view software development [13, 11], and traceability and reverse engineering [5].

Despite these promising uses of bidirectional model transformation in software development, there are still very few serious practical applications in which bidirectional model transformation is actually used. There are three known problems.

First, there is uncertainty over fundamental semantics of bidirectional transformation. As strongly argued in [25], there lacks a clear definition of what bidirectional model transformation means. In practice, a model transformation may not be bijective, so its backward model transformation should not be defined merely as an inverse of forward model transformation. However, if there is no clear semantics of bidirectional transformation or backward transformation is not guaranteed to work without harm, no one would seriously use it in their systems.

Second, there is no powerful bidirectionalization method yet that can be used to automatically derive backward model transformations from forward model transformations so that both transformations form a consistent bidirectional model transformation. Even for simpler tree transformation, it is known [10, 21, 4] to be impractical in the sense that they ask users to write both forward and backward transformations (which may be very big) and to guarantee their bidirectionality. Although some attempts have been made [8, 27], only the part that has direct one-to-one correspondence between

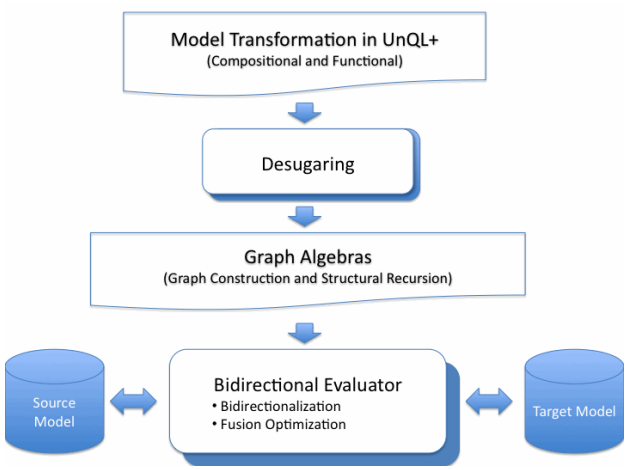

Figure 1. An Algebraic Framework for Bidirectional Model Transformation Framework

the source and target models can be transformed back and forth.

Third, and very important, there lacks a systematic way to develop bidirectional model transformation in large. As indicated in the conclusion in [9], most model transformation languages [9, 7] based on graph transformations have weak composition mechanism, which makes them hard to support systematic development of model transformations in the large [18]. The existing examples of bidirectional model transformations are somehow too simple to show both challenges and advantages of bidirectional model transformation.

In this paper, we propose a novel algebraic framework for bidirectional model transformation to solve these problems, by integrating two state-of-the-art techniques: bidirectional tree transformations [10, 21, 4, 16] in the programming language community, and the algebraic graph querying language UnQL [6] intensively studied in the database community. We make a significant extension from bidirectional *tree* transformation to bidirectional *graph* transformation, and give a powerful automatic bidirectionalization method to derive backward graph transformations from a forward transformation in UnQL$^+$ [14], an extension of UnQL for graph transformation.

Figure 1 depicts an architecture of our algebraic framework. A model transformation is described in UnQL$^+$, which is *functional* (rather than rule-based as in many existing tools) and *compositional* with high modularity for reuse and maintenance. The model transformation is then desugared to a core *graph algebra* which consists of a set of constructors for building graphs and a powerful structural recursion for manipulating graphs. This graph algebra can have clear bidirectional semantics and be efficiently evaluated in a bidirectional manner. Our main technical contributions can be summarized as follows.

- We made the first attempt of adapting an existing well-established graph querying language for model transformation, whose importance has not been recognized so far. We show that UnQL, a powerful graph querying language being suitable for systematic development of efficient graph queries, can be extended for systematic development of useful model transformations, as demonstrated in Section 6.

- We propose the first general method for bidirectionalizing *graph* transformation, although there have proposed several bidirectional *tree* transformation languages. Compared with trees, the challenge here is how to deal with cycles and traversals of graphs in bidirectional computation. The key to the success of our bidirectionalization is the simple but powerful internal algebra of UnQL$^+$, where structural recursion can be computed in a *bulk* way which is suitable for both forward and backward computation.

- We give an efficient implementation of bidirectional computation of UnQL$^+$. We carefully record necessary but least information during forward transformation for later efficient backward transformation, and apply automatic fusion transformation to eliminate unnecessary intermediate models used in model transformation composition. The core system has been implemented in Objective Caml with about 7,500 loc, and all the examples in this paper have been passed by the system. More application examples can be found at the system page[1].

The organization of this paper is as follows. After reviewing graph data model and structural recursion in UnQL in Section 2, we show that UnQL can be extended to UnQL$^+$ that are suitable for developing model transformations in a compositional way in Section 3. To implement our bidirectional framework, we show how to desugar UnQL$^+$ to a core graph algebra in Section 4, and how to implement a bidirectional evaluator for the core graph algebra with a clear bidirectional semantics and efficient implementation in Section 5. We show an application in Section 6, discuss related work in Section 7, and conclude the paper in Section 8.

2 Graph Data Model and Structural Recursion on Graphs

We start with a brief explanation of graph data model and structural recursion, which are two important concepts

[1] `http://research.nii.ac.jp/~hu/big/`

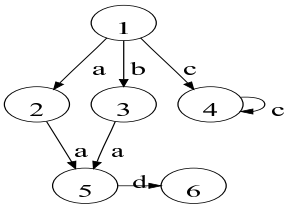

Figure 2. A Simple Graph

in the graph querying language UnQL [6]. They serve as the basis of our algebraic framework for model transformation.

2.1 Graph Data Model

External Graph Representation Graphs in UnQL are rooted and directed cyclic graphs with no order on outgoing edges. They are edged-labelled in the sense that all information is stored as labels on edges (the labels on nodes have no particular meaning). Figure 2 gives a small example of a directed cyclic graph with six nodes and seven edges. In text, it is represented by

$$\begin{array}{lcl} g & = & \{a : \{a : g_1\}, b : \{a : g_1\}, c : g_2\} \\ g_1 & = & \{d : \{\}\} \\ g_2 & = & \{c : g_2\} \end{array}$$

where the set $\{l_1 : e_1, \ldots, l_n : e_n\}$ denotes a graph which contains n edges with labels $l_1, \ldots, l_n$, each of which points to a graph again, and the empty set $\{\}$ denotes a graph with a single node. Two graphs can be merged using set union operation such as $g \cup g'$.

As another example, consider to represent the class model diagram in Figure 3 by an edge-labelled graph. This example is from [3], which will also be used in Section 6. A class model consists of classes and directed associations between classes. A class is indicated as persistent or non-persistent. It consists of one or more attributes, at least one of which must be marked as constituting the classes' primary key. An attribute type is of a primitive data type (e.g. String, Integer). An association specifies an inheritance relation between two classes. The class model in Figure 3 consists of three classes and two directed associations, where each class has a primary attribute. This class model can be represented by the graph in Figure 4, where information is moved to edges.

Internal Graph Representation While the external graph representation is sufficient for users to consider when writing bidirectional model transformation, the internal graph representation is designed for internal implementation and semantics description of structural recursion on

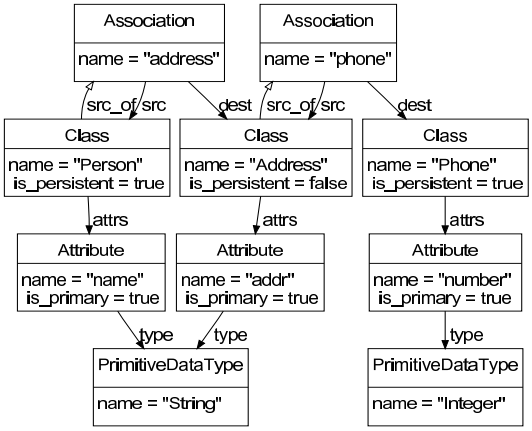

Figure 3. A Class Diagram

graphs. Different from the external representation, the internal representation introduce ϵ edge for representing shortcut of two nodes. For instance, we have

$$\{\epsilon : \{a : g_1\}, \epsilon : \{b : g_2\}\} = \{a : g_1, b : g_2\}.$$

As will be seen in Sections 2.2 and 5.1, the ϵ edge is important in defining both bulk and bidirectional semantics of structural recursion.

Another difference is that for internal composition of graphs, an internal graph may have some node marked with input or output marker, which is called input or output node, respectively. We use $\&x \in Marker$ to denote that $\&x$ is a marker. Input markers are used to select entry points of the graph, whereas output markers are used to connect with other input nodes later.

Graph Bisimulation *Graph bisimulation* defines value equalities between graph instances. Intuitively, when graph G_1 and G_2 are bisimilar, then every node x_1 in G_1 has a counterpart x_2 in G_2, and if there is an edge form x_1 to y_1, then there is a corresponding edge from x_2 to y_2. UnQL data model extends graph bisimulation by (1) requiring equalities between labels, (2) allowing insertion of one or more consecutive ϵ edges between normal edge and target node (y_1 or y_2 above), (3) requiring correspondence between input nodes in G_1 and G_2, (4) requiring correspondence between output markers of corresponding nodes (output markers may be associated with the node other than corresponding nodes, provided that the marker is associated with nodes that can be reached by traversing ϵ edges).

The notion of extended bisimulation is useful because it allows variation in representing semantically equivalent graphs. It has been shown that a graph transformation defined in UnQL preserves bisimilarity [6]. If two graphs G_1

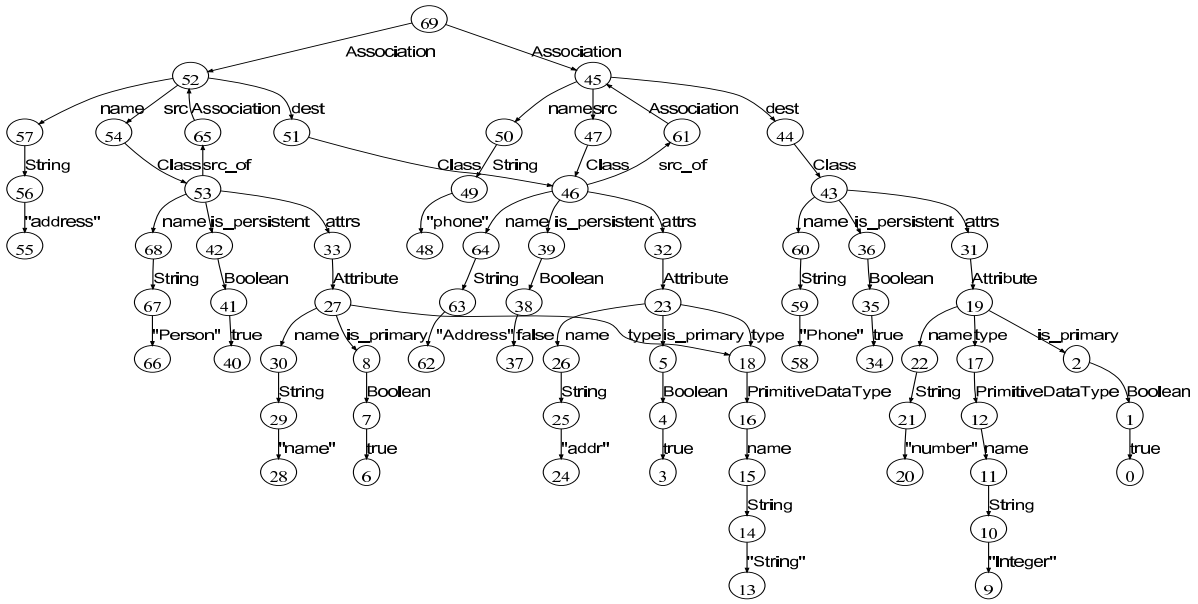

Figure 4. A Class Model Represented by an Edge-Labelled Graph

and G_2 are bisimilar, $f(G_1)$ and $f(G_2)$ are bisimilar for any transformation f in UnQL.

2.2 Structural recursion in UnQL

Structural recursion plays a significant role in our framework for description of graph transformation, bidirectionalization, and optimization. As will be shown in Section 4, any graph transformation in UnQL and its extension (Section 3) can be described in terms of structural recursion.

2.2.1 Structural Recursion

Structural recursive function f in UnQL is a recursive computation scheme on graphs defined as follows, where $\odot$ is a given binary operator.

$$\begin{array}{lcl} f\left(\{\}\right) & = & \{\} \\ f\left(\{l:g\}\right) & = & l \odot f(g) \\ f\left(g_1 \cup g_2\right) & = & f(g_1) \cup f(g_2) \end{array}$$

Different choices of $\odot$ defines different functions, and for simplicity we abbreviate the above definition as follows.

$$\mathsf{sfun}\ f\left(\{l:g\}\right) = l \odot f(g)$$

Note that for a graph g which may contain cycles, the computation of $f(g)$ always terminates under the usual *recursive semantics*: remember all recursive calls and reuse their result to avoid entering infinite loops.

As a simple example, we may use the following structural recursion to replace all edges labelled a by d and delete the edges labeled c for the graph in Figure 2.

$$\begin{array}{lcl} \mathsf{sfun}\ a2d\_xc\left(\{l:g\}\right) & = & \mathsf{if}\ l = a\ \mathsf{then}\ \{d:f(g)\} \\ & & \mathsf{else\ if}\ l = c\ \mathsf{then}\ f(g) \\ & & \mathsf{else}\ \{l:f(g)\} \end{array}$$

A natural extension of the above structural recursion is to allow mutual recursion, because any mutually defined functions can be merged into one by the standard tupling transformation [15]. The following mutually defined structural recursive function $relabel$ can replace all labels `name` under $primitiveDataType$ with $typeName$ in Figure 4 .

$$\begin{array}{ll} \mathsf{sfun} & relabel\left(\{primitiveDataType:g\}\right) \\ & \quad = \{primitiveDataType : addT(g)\} \\ \mid & relabel\left(\{l:g\}\right) = \{l : relabel(g)\} \\ \mathsf{sfun} & addT\left(\{name:g\}\right) = \{typeName : addT(g)\} \\ \mid & addT\left(\{l:g\}\right) = \{l : addT(g)\} \end{array}$$

2.2.2 Bulk Semantics

One important feature of a structural recursive function is that it can be computed in a *bulk* manner (called *bulk semantics* [6]), which make it possible for our bidirectionalization (Section 5.1) and automatic optimization (Section 5.2).

Given a structural recursive function defined by

$$\mathsf{sfun}\ f\left(\{l:g\}\right) = l \odot f(g)$$

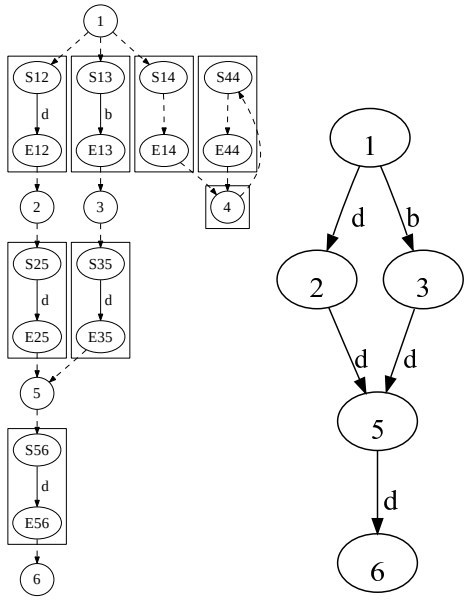

Figure 5. A Simple Edge-labelled Graph

the computation of f on a graph can be performed by the following three steps: (1) applying to each edge the following function

$$f_e(l) = l \odot \&z$$

where $\&z$ denotes a node with an output marker $\&z$ which will be connected with other nodes later, (2) marking the root node of the result graph with the input marker $\&z$, and (3) joining the results with the ϵ edge.

To be concrete, consider to apply the structural recursive function $a2d\_xc$ to the graph in Figure 2. Applying the function to each edge from i to j gives a subgraph containing a graph with an edge from Sij to Eij (where the dotted edge denotes an ϵ edge), then marking the root with an input marker, and finally joining the nodes with ϵ edges according to the original graph shape and input/out markers yields the graph in Figure 5 (left), which is equivalent to the graph in Figure 5 (right) if we remove all ϵ edges.

3 Model Querying and Transformation

UnQL [6] is a graph querying language based on structural recursion, and has an expressive power of FO(TC) (first order with transitive closure), with time complexity of PTIME for graph querying. It has a friendly surface language with a select-where structure, with which users can write graph queries without explicit use of structural recursion. In this section, we review the graph querying language UnQL [6], show our extension of UnQL to UnQL$^+$ for graph transformation (rather than just graph querying) [14], and demonstrate how it is used for writing model transformations.

3.1 Model Querying in UnQL

UnQL, like other query languages, has a convenient and powerful *select-where* structure for extracting information from a graph. We omit the formal definition of the language syntax, which can be found in [6]. For instance, the following query extracts all persistent classes from the class model in Figure 4, which is assumed to be bound by $\$classDB$.

$$\begin{array}{l}\mathsf{select}\ \$class\ \mathsf{where}\\ \quad \{Association.(src|dest).Class : \$class\}\ \mathsf{in}\ \$classDB,\\ \quad \{is\_presistent : \{Boolean : true\}\}\ \mathsf{in}\ \$class\end{array}$$

The symbols prefixed with $ denote variables. This query returns all bindings of variable $\$class$ satisfying the two conditions in the where clause. The first condition is to find bindings of $\$class$ by matching the *regular path pattern* $Association.(src|dest).Class$ with the graph bound by $\$classDB$, while the second condition is to ensure that the class is persistent.

3.2 Model Transformation in UnQL$^+$

In model transformation, we often want to replace a subgraph satisfying certain condition by another graph. It is onerous to describe these kinds of graph transformations in UnQL because some context structure is required to be copied and propagated. UnQL$^+$ is an extension of UnQL with a new *replace-where* construct suitable for specifying model transformation. We illustrate our idea with some examples, but omit the details which can be found in [14].

Consider that we want to add a prefix of $class\_$ to each class name in the class model. We may write it with the replace-where structure by

$$\begin{array}{l}\mathsf{replace}\ \{\$name : \{\}\}\ \mathsf{by}\ \{(\text{"}class\_\text{"}\ \hat{}\ \$name) : \{\}\}\ \mathsf{where}\\ \quad \{\_*.Class.name.String : \{\$name : \{\}\}\}\ \mathsf{in}\ \$classDB\end{array}$$

where $\hat{}$ denotes string concatenation, and $\_*$ denotes arbitrary sequence of labels (in the path). The replace-where clause is similar to the select-where clause except that subgraph to be replaced is bound by either a graph variable $\$g$ or a pair of a label and graph variables $\{\$l : \$g\}$.

The replace-where construct can be used to define various model transformations such as extension of a subgraph with some new information and deletion of a subgraph satisfying a certain condition.

$$\begin{array}{l}\mathsf{extend}\ \$g\ \mathsf{with}\ \ \$g'\ \mathsf{where}\ \ldots\\ \quad \stackrel{\text{def}}{=}\ \mathsf{replace}\ \$g\ \mathsf{by}\ (\$g \cup \$g')\ \mathsf{where}\ \ldots\\ \mathsf{delete}\ \$g\ \mathsf{where}\ \ldots\\ \quad \stackrel{\text{def}}{=}\ \mathsf{replace}\ \$g\ \mathsf{by}\ \{\}\ \mathsf{where}\ \ldots\end{array}$$

3.3 Compositional Transformation

Unlike most rule-based model transformation languages where model transformation composition is not straightforwardly supported [9], UnQL$^+$ is functional and compositional; model transformations are functions, and smaller model transformations are composed to form a bigger one.

Consider that we want to extract all persistent classes from the class model $\$classDB$, and transform them to tables by replacing $attrs$ by $cols$ and $Attribute$ by $Column$. This can be described by composition of three transformations, each corresponding to one step above.

```
(* replace Attribute *)
replace{$l_A : $g} by{Column : $g} where
  $db in
   (* replace attrs *)
   (replace{$l_a : $A} by{cols : $A} where
      $class in
       (* select classes *)
       (select $class where
        {Association.(src|dest).Class : $class}
            in $classDB,
        {is_presistent : {Boolean : true}} in $class),
      {$l_a : $A} in $class, $l_a = attrs),
  {cols : {$l_A : $g}} in $db, $l_A = Attribute
```

A more involved example and discussion on systematic development of "big" model transformations with composition can be found in Section 6.

4 Desugaring UnQL$^+$ to Graph Algebra

While UnQL$^+$ is for users to write model transformations, UnCAL is its internal algebra for implementation, suitable for bidirectionalization and optimization.

4.1 Graph Constructors

There are nine data constructor which can be used to describe construction of arbitrary graphs.

- $\{\}$ (single node graph): it constructs a graph with a single node without an edge.
- $\{l : g\}$ (singleton graph): it constructs a graph with the root pointing to the root of the graph g through the edge l.
- $g_1 \cup g_2$ (union of graphs): it unions two graphs as defined in Section 2.
- $\&x := g$ (graph with input marker): it adds some input marker to the root of g.
- $\&y$ (output node): it constructs a graph with a single node marked with one output marker.
- $()$ (empty graph): it constructs an empty graph which has neither node nor edge.
- $g_1 \oplus g_2$ (disjoint union of graphs): it constructs a graph by putting two graphs one next to the other horizontally.
- g_1 @ g_2 (append of graphs): it connect the two graphs vertically by connecting the output nodes of g_1 with corresponding input nodes of g_2.
- $cycle(g)$ (cyclic graph): it connects the input nodes with the output nodes of g to form cycles.

4.2 UnCAL: A Graph Algebra

UnCAL, as defined in Figure 6, has a set of graph constructors and operators, by which arbitrary graphs can be represented and arbitrary graph transformation in UnQL$^+$ can be described.

The first nine expression structures correspond to nine graph data constructors. They are used to describe graph construction. For instance, the graph in Figure 2 can be represented by the following UnCAL expression.

$$\begin{array}{l}\&z_1 \text{ @ } cycle(\\ \quad (\&z_1 := \{a : \{a : \&z_5\}, b : \{a : \&z_5\}, c : \&z_4\}, \\ \quad \&z_5 := \{d : \{\}\}, \\ \quad \&z_4 := \{c : \&z_4\}))\end{array}$$

The most important operation for manipulating graphs in this algebra is structural recursion $rec(\lambda(l, g).e)$, corresponding to the following function h:

$$\mathsf{sfun}\ h\ (\{l : g\}) \quad = \quad e \text{ @ } h(g).$$

For any structural recursive function defined by

$$\mathsf{sfun}\ f\ (\{l : g\}) \quad = \quad l \odot f(g)$$

$f(v)$ can be represented in terms of rec by

$$\&z_1 \text{ @ } (rec(\lambda(l, g).(\&z_1 := l \odot \&z_1))(v)).$$

4.3 Mapping to UnCAL

UnQL$^+$ can be fully transformed to UnCAL. UnQL$^+$ is mapped to UnCAL in a way similar to the mapping of UnQL to UnCAL in [6] except for the newly introduced replace-where construct which can be encoded by structural recursion. The mapping from UnQL$^+$ to UnCAL consists of the following six steps: (1) simplifying where clauses;

$$
\begin{array}{rcl}
E & ::= & \{\} \\
& | & \{L : E\} \\
& | & E \cup E \\
& | & \&x := E \\
& | & \&y \\
& | & () \\
& | & E \oplus E \\
& | & E \ @ \ E \\
& | & cycle(E) \\
& | & Var \\
& | & \mathsf{if}\ B\ \mathsf{then}\ E\ \mathsf{else}\ E \\
& | & rec(\lambda(LabelVar, Var).E)(Var) \\
& | & \mathsf{let}\ Var = E\ \mathsf{in}\ E
\end{array}
$$

Figure 6. UnCAL: A Graph Algebra

(2) eliminating the replace-where construct; (3) transforming simple patterns to structural recursions; (4) transforming regular path patterns to mutual structural recursions; (5) tupling mutual structural recursions to single ones; and (6) mapping structural recursions to those in UnCAL. The details of this mapping can be found in [14].

5 Bidirectional Evaluator

This section explains one of our important contributions, the first bidirectional interpretation of the graph algebra of UnCAL. The prototype system has been implemented in Objective Caml and the GUI interface for users to construct and modify graph models through the DOT and DOTTY system[2], graph layout products developed by AT&T. In this section, we describe our bidirectional semantics for UnCAL, and highlight how the system is implemented efficiently by automatic fusion.

5.1 Bidirectionalizing UnCAL

UnCAL can be bidirectionalized in the sense that a formal and sound bidirectional semantics can be given to UnCAL, such that the forward computation performs the same as the usual UnCAL evaluator does, and the backward computation is guaranteed to satisfy the bidirectional properties [10] with respect to the forward computation.

5.1.1 Bidirectional Properties

Given an expression e and an environment ρ denoting a mapping from variables to values (a label or a graph), we define two computations: a *forward computation*

$$\rho \ \overset{e}{\rightharpoonup} \ g$$

is to evaluate the expression e to a graph g under the environment ρ, while a *backward computation*

$$\rho' \ \overset{e}{\leftharpoonup}_{\rho} \ g'$$

is to compute a new environment ρ' from the old ρ and a revised graph g' over g obtained from forward computation. The forward and backward computations with respect to an expression e should satisfy the following two properties [10].

- The *GetPut* Property: no change on the graph should give no change on the environment.

$$\frac{\rho \ \overset{e}{\rightharpoonup} \ g}{\rho \ \overset{e}{\leftharpoonup}_{\rho} \ g}$$

- The *PutGet* Property: the backward computation computes a new environment ρ' from g' in such a way that applying the forward computation under ρ' again should give the same graph g'.

$$\frac{\rho' \ \overset{e}{\leftharpoonup}_{\rho} \ g'}{\rho' \ \overset{e}{\rightharpoonup} \ g'}$$

5.1.2 Bidirectional Interpretation of UnCAL

We give a set of rules to describe both forward and backward interpretation of expressions of UnCAL, which satisfy the bidirectional properties.

The rules for forward computation is summarized in Figure 7. It is quite standard. We choose some to explain. The rule FWD_SG says that a single graph expression $\{le : e\}$ is evaluated by first computing the label expression le and the graph expression e under the environment ρ and then combining their results. The input marker rule (FWD_I) says that the expression $\&x := e$ is evaluated by first computing e and get a disjoint union of n graphs $g_1, \ldots, g_n$ where g_i has the input marker of $\&y_i$, then the root node of this disjoint union is given a new input marker $\&y$ and the original input marker of $\&y_i$ is renamed to a new marker $\&x.\&y_i$. Here the infix operator . is a Skolem function on markers which is associative, idempotent with respect to default input (root) marker & ($\&.\&x = \&x.\& = \&x$), and *invertible*. The most interesting rule is FWD_REC for evaluating structural recursion. As explained in Section 2.2, it computes on each edge of the source graph in parallel and then combines the results.

The rules for backward computation is to reflect the change on the output back to the input by changing the binding information of variables in the environment. The rules summarized in Figure 8, which guarantee the bidirectional properties.

[2] http://hoagland.org/Dot.html

$$\rho \xrightarrow{\{\}} \{\} \quad \text{(FWD\_SN)} \qquad \frac{\rho \xrightarrow{le} l \quad \rho \xrightarrow{e} g}{\rho \xrightarrow{\{le:e\}} \{l : g\}} \quad \text{(FWD\_SG)} \qquad \frac{\rho \xrightarrow{e_1} g_1 \quad \rho \xrightarrow{e_2} g_2}{\rho \xrightarrow{e_1 \cup e_2} g_1 \cup g_2} \quad \text{(FWD\_UNION)}$$

$$\frac{\rho \xrightarrow{e} (\&y_1 := g_1, \ldots, \&y_n := g_n)}{\rho \xrightarrow{\&x := e} \&y := (\&x.\&y_1 := g_1, \ldots, \&x.\&y_n := g_n)} \quad \text{(FWD\_I)} \qquad \rho \xrightarrow{\&y} \&y \quad \text{(FWD\_ON)} \qquad \rho \xrightarrow{()} () \quad \text{(FWD\_EMP)}$$

$$\frac{\rho \xrightarrow{e_1} (g_{11}, \ldots, g_{1m}) \quad \rho \xrightarrow{e_2} (g_{21}, \ldots, g_{2n})}{\rho \xrightarrow{e_1 \oplus e_2} (g_{11}, \ldots, g_{1m}, g_{21}, \ldots, g_{2n})} \quad \text{(FWD\_DUNION)} \qquad \frac{\rho \xrightarrow{e_1} (g_{11}, \ldots, g_{1m}) \quad \rho \xrightarrow{e_2} (g_{21}, \ldots, g_{2n})}{\rho \xrightarrow{e_1 @ e_2} (g_{31}, \ldots, g_{3m})} \quad \text{(FWD\_APPEND)}$$

$$\frac{\rho \xrightarrow{e} g}{\rho \xrightarrow{cycle(e)} cycle(g)} \quad \text{(FWD\_CYCLE)} \qquad \rho \xrightarrow{v} \rho(v) \quad \text{(FWD\_VAR)} \qquad \frac{\rho \xrightarrow{b} true \quad \rho \xrightarrow{e_1} g}{\rho \xrightarrow{\text{if } b \text{ then } e_1 \text{ else } e_2} g} \quad \text{(FWD\_IF)}$$

$$\frac{\begin{array}{c}\text{for each } \{l_i : g_i\} \text{ in } \rho(v) : \ \rho[l \leftarrow l_i, g \leftarrow g_i] \xrightarrow{e} g_i \\ mergedG = merge(g_1, \ldots, g_n)\end{array}}{\rho \xrightarrow{rec(\lambda(l,g).e)(v)} mergedG} \quad \text{(FWD\_REC)} \qquad \frac{\rho \xrightarrow{e_1} g_1 \quad \rho[v \leftarrow g_1] \xrightarrow{e_2} g_2}{\rho \xrightarrow{\text{let } v = e_1 \text{ in } e_2} g_2} \quad \text{(FWD\_LET)}$$

Figure 7. Forward Computation Rules

Rules BWD_SN, BWD_ON and BWD_EMP indicate that a graph produced by the single node expression, the output marker expression, or the empty expression should have no effect on the environment because their forward computation does not depend on the environment. Rules BWD_UNION, BWD_DUNION, BWD_APPEND, BWD_REC first use the operator $\Rightarrow$ to decompose the revised graph in such as way that each can be used for backward computation with one of the subexpressions, and then unify the updated environments by $\uplus_\rho$ such as $\rho'_1 \uplus_\rho \rho'_2$. We make such decomposition possible by associating graph nodes with information of where they come from during the forward computation. Because of this decomposition, these rules do not allow insertion of new edges. An output graph can be freely updated with insertion and deletion if it is computed from a variable in its forward computation, as seen in Rule BWD_VAR.

One interesting rule is Rule BWD_LET for dealing with composition. It has two stages: performing backward computation for e_2 using the forward computation result of e_1 before performing backward computation for e_1. This rule shows the difference between bidirectional computation and inverse computation, where inverse computation does not need to use input.

As an example of backward computation where modification propagates to source, consider the following simple UnCAL expression in which the forward computation just prepend *result* edge to $\$classDB$

$$\text{let } \$v = \$classDB \text{ in } \{result : \$v\}$$

and suppose the user modifies the "Phone" edge to "Telephone" on the result. Let this result be g'. During backward computation, BWD_LET is invoked, which in tern invokes backward computation of $\{result : \$v\}$ (first stage above) using environment $\rho = [\$classDB \leftarrow h, \$v \leftarrow h]$. This invokes BWD_SG which extract h' from g' by removing topmost *result* edge and BWD_VAR reflects this modification back to the binding of $\$v$. Going back to BWD_LET, using the modified binding of $\$v$ as the new value of the expression $\$classDB$, BWD_VAR is invoked again (second stage above), which results in updated binding of $\$classDB$, the source graph. Lastly, $\uplus_\rho$ unifies the results of the two stages. In computing $[\$classDB \leftarrow h'] \uplus_{[\$classDB \leftarrow h]} [\$classDB \leftarrow h]$, $\uplus$ knows that left hand side indicates modification by comparing it with original value of $\$classDB$ in its subscript environment. We can thus obtain the updated graph by extracting this new binding from this resultant environment.

5.2 Optimization by Fusion

In our framework, consecutive model transformations are translated into composition of structural recursions in UnCAL. The composition may introduce unnecessary intermediate graphs, but this efficiency can be removed by applying fusion transformation [6]. The following two rules are main fusion transformation rules for cascading *rec*'s. The first rule is applied when $e_2(l, t)$ does not depend on t, while the second rule is applied otherwise.

$$\begin{array}{l} rec(e_2) \circ rec(e_1) = rec(rec(e_2)) \circ e_1 \\ rec(e_2) \circ rec(e_1) \\ \qquad = rec(\lambda(l, g).\, rec(e_2)(e_1(l, g) \mathbin{@} rec(e_1)(g))) \end{array}$$

Here $\circ$ denotes function composition, i.e., $(f \circ g)(x) = f(g(x))$. To make the above rules applicable, we apply the

$$\rho \overset{\{\}}{\leftarrow}_\rho \{\} \text{ (BWD\_SN)} \qquad \frac{\rho_l \overset{le}{\leftarrow}_\rho l' \quad \rho_e \overset{e}{\leftarrow}_\rho g'}{\rho_l \uplus_\rho \rho_e \overset{\{le:e\}}{\leftarrow}_\rho \{l' : g'\}} \text{ (BWD\_SG)} \qquad \frac{g' \Rightarrow g'_1 \cup g'_2 \quad \rho_1 \overset{e_1}{\leftarrow}_\rho g'_1 \quad \rho_2 \overset{e_2}{\leftarrow}_\rho g'_2}{\rho_1 \uplus_\rho \rho_2 \overset{e_1 \cup e_2}{\leftarrow}_\rho g'} \text{ (BWD\_UNION)}$$

$$\frac{\rho' \overset{e}{\leftarrow}_\rho (\&y_1 := g'_1, \ldots, \&y_n := g'_n)}{\rho' \overset{\&x:=e}{\leftarrow}_\rho \&y := (\&x.\&y_1 := g'_1, \ldots, \&x.\&y_n := g'_n)} \text{ (BWD\_I)} \qquad \rho \overset{\&y}{\leftarrow}_\rho \&y \text{ (BWD\_ON)} \qquad \rho \overset{()}{\leftarrow}_\rho () \text{ (BWD\_EMP)}$$

$$\frac{\rho'_1 \overset{e_1}{\leftarrow}_\rho (g'_{11}, \ldots, g'_{1m}) \quad \rho'_2 \overset{e_2}{\leftarrow}_\rho (g'_{21}, \ldots, g'_{2n})}{\rho'_1 \uplus_\rho \rho'_2 \overset{e_1 \oplus e_2}{\leftarrow}_\rho (g'_{11}, \ldots, g'_{1m}, g'_{21}, \ldots, g'_{2n})} \text{ (BWD\_DUNION)}$$

$$\frac{(g'_{31}, \ldots, g'_{3m}) \Rightarrow (g'_{11}, \ldots, g'_{1m}) \mathbin{@} (g'_{21}, \ldots, g'_{2n}) \quad \rho'_1 \overset{e_1}{\leftarrow}_\rho (g'_{11}, \ldots, g'_{1m}) \quad \rho'_2 \overset{e_2}{\leftarrow}_\rho (g'_{21}, \ldots, g'_{2n})}{\rho'_1 \uplus_\rho \rho'_2 \overset{e_1 @ e_2}{\leftarrow}_\rho (g'_{31}, \ldots, g'_{3m})} \text{ (BWD\_APPEND)}$$

$$\frac{\rho' \overset{e}{\leftarrow}_\rho g'}{\rho' \overset{cycle(e)}{\leftarrow}_\rho cycle(g')} \text{ (BWD\_CYCLE)} \qquad \rho[v \leftarrow g'] \overset{v}{\leftarrow}_\rho g' \text{ (BWD\_VAR)}$$

$$\frac{\rho' \overset{e_1}{\leftarrow}_\rho g' \quad \rho' \overset{b}{\rightharpoonup} true}{\rho' \overset{\text{if } b \text{ then} e_1 \text{ else} e_2}{\leftarrow}_\rho g'} \text{ (BWD\_IF\_TRUE)} \qquad \frac{\rho' \overset{e_2}{\leftarrow}_\rho g' \quad \rho' \overset{b}{\rightharpoonup} false}{\rho' \overset{\text{if } b \text{ then } e_1 \text{ else } e_2}{\leftarrow}_\rho g'} \text{ (BWD\_IF\_FALSE)}$$

$$\frac{\begin{array}{c} g' \Rightarrow g'_1, \ldots, g'_n \qquad \rho(v) \Rightarrow g_1, \ldots, g_n \\ \rho'_i \overset{e}{\leftarrow}_{\rho[\{l:g\} \leftarrow g_i]} g'_i \\ \rho' = mergeEnv_\rho(\rho'_1, \ldots, \rho'_n) \\ g'' = merge(\{\rho_1(l) : \rho_1(g)\}, \ldots, \{\{\rho_n(l) : \rho_n(g)\}) \\ \rho'' = \rho'[v \leftarrow g''] \end{array}}{\rho'' \overset{rec(\lambda(l,g).e)(v)}{\leftarrow}_\rho g'} \text{ (BWD\_REC)}$$

$$\frac{\rho \overset{e_1}{\rightharpoonup} g_1 \quad \rho'_1 \overset{e_2}{\leftarrow}_{\rho[v \leftarrow g_1]} g' \quad \rho' \overset{e_1}{\leftarrow}_\rho \rho'_1(v)}{\rho' \uplus_\rho (\rho'_1 \setminus v) \overset{\text{let } v=e_1 \text{ in } e_2}{\leftarrow}_\rho g'} \text{ (BWD\_LET)}$$

Figure 8. Backward Computation Rules

following rules to to move *rec*s close to each other.

$$\begin{array}{lcl} rec(e)(\{\}) & = & \{\} \\ rec(e)(\{l : d\}) & = & e(l, d) \mathbin{@} rec(e)(d) \\ rec(e)(d_1 \cup d_2) & = & rec(e)(d_1) \cup rec(e)(d_2) \\ rec(e)(\&x := d) & = & \&x \cdot (rec(e)(d)) \\ rec(e)() & = & () \\ rec(e)(d_1 \oplus d_2) & = & rec(e)(d_1) \oplus rec(e)(d_2) \end{array}$$

$$\frac{g \text{ does not occur free in } e}{rec(\lambda(l, g).e)(d_1 \mathbin{@} d_2) = rec(e)(d_1) \mathbin{@} rec(e)(d_2)}$$

$$\frac{g \text{ does not occur free in } e}{rec(\lambda(l, g).e)(cycle(d)) = cycle(rec(e)(d))}$$

6 An Application: Class2RDBMS

This section shows how we write an UnQL$^+$ program for a practical model transformation, *Class2RDBMS*. It is a simplified version of the well known "Class to RDBMS" which was proposed at [3] to compare and contrast various kinds of approaches to model transformations. We show how the model transformation can be systematically developed in UnQL$^+$ after explaining the requirement of a model transformation Class2RDBMS.

6.1 Specification of Class2RDBMS

Class2RDBMS is a model transformation from Class models, which has been explained in Section 2, to RDBMS

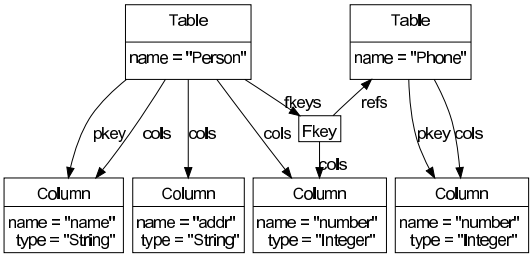

Figure 9. A RDBMS Model

models. For instance, it transforms the Class model in Figure 3 into an RDBMS model in Figure 9. The corresponding RDBMS model should satisfy the following requirement. Class2RDBMS maps each persistent class in a Class model to a table in a RDBMS model. All attributes of the class or its subclasses are mapped to columns in the corresponding table. If a primary attribute belongs to the class, a `pkey` pointing from the table to the corresponding column is established. If an attribute belongs to its subclass which is persistent, a foreign key to the corresponding table is established. Non-persistent classes are not mapped to tables, however. One of the main requirements for Class2RDBMS is to preserve all the information in the class diagram. That means all attributes of non-persistent subclasses are distributed over those tables stemming from persistent classes which access non-persistent classes. Note that we represent RDBMS models by edge-labelled graphs in UnQL$^+$ as

well as Class models. This model transformation is not trivial. We show below how to systematically develop it in our framework.

6.2 Class2RDBMS in UnQL$^+$

The framework of UnQL$^+$ allows us to develop bigger model transformations by gluing smaller transformations via intermediate models, without worrying about inefficiency due to the intermediate models. This is because unnecessary intermediate models will be removed automatically by our system. Figure 10 shows the whole transformation of Class2RDBMS in UnQL$^+$. Let us explain how it is systematically developed.

We construct an UnQL$^+$ program for Class2RDBMS by splitting its specification into two steps. On the first step, every persistent class is mapped to a table which is connected with its columns according to attributes of the class and its subclasses. All subclasses are collected by regular path patterns as shown in Section 3.1. If necessary, references `pkey` and `fkeys` are added by an `extend` construct in UnQL$^+$, provided that references `refs` of `Fkey` do not point to the referring table because the table may not have been constructed yet. They point to the name of the referring table instead. On the second step, each name pointed by `refs` is replaced by the corresponding table by using a `replace` construct.

Our framework allows us to modify on the target model RDBMS. The modification is reflected to the source model Class. For example, we can modify string values (e.g., `"Person"` and `"addr"`) in the RDBMS model in Figure 9.

7 Related Work

Besides the related work in the introduction, we highlight some others that are related to graph-based model transformation, graph querying, and linguistic approach to bidirectional programming.

Our work is much related to research on model transformation based on graph transformation [9, 22, 17]. AGG [26] is a well-known rule-based visual tool that supports typed (attributed) graph transformations including type inheritance and multiplicities. Triple Graph Grammars (TGG) [19, 12] is an extension of Pratt's pair grammar approach [23], which aims at the declarative specification of model to model integration rules. Different from these approach which are rule-based, our approach is function-based, using graph algebras for graph construction, model transformation composition for systematic development, and model transformation manipulation for automatic optimization.

Our work was greatly influenced by interesting work on efficient graph querying [24, 6]. Unlike trees, graphs have subtle issues on their representation and their equivalence. The use of bisimulation and structural recursion in [6] opens a new way to build a framework for both declarative and efficient graph querying with high modularity and composability. This motivated us to extend the framework from graph querying to graph transformation and apply it to model transformation.

This work has been inspired by recent work on linguistic approach to bidirectionalization of tree transformation [10, 21, 4, 16] for tree data synchronization. One important feature of these systems is a clear bidirectional semantics, which, however, does not exist in most existing bidirectional model transformation systems [22, 25]. Although some attempts have been made [27, 2], it remains as a challenge to provide a general bidirectional framework for graphs, and our this work is a big step to this direction.

8 Conclusions

In this paper, we propose a novel algebraic framework to support systematic development of bidirectional model transformation. Different from many existing frameworks that are rule-based, our framework is functional and algebraic, which is based on a graph algebra and structural recursion. Our new framework supports systematic development of model transformations in a compositional manner, has a clear semantics for bidirectional model transformation, and can be efficiently implemented.

This work is our first step towards *bidirectional model programming*, a linguistic framework to support systematic development of model transformation programs. In the future, we wish to look more into relation between the rule-based approach and the algebraic and functional approach, and see how to integrate them to have a more powerful framework for bidirectional model transformation.

Acknowledgements

We would like to thank Mary Fernandez from AT&T Labs Research, who kindly provided us with the SML source codes of an UnQL system, which helped us a lot in implementing our extended system in OCaml.

The research was supported in part by the Grand-Challenging Project on "Linguistic Foundation for Bidirectional Model Transformation" from National Institute of Informatics, the National Natural Science Foundation of China under Grant No. 60528006, Grant-in-Aid for Scientific Research (C) No.20500043, and Encouragement of Young Scientists (B) of the Grant-in-Aid for Scientific Research No. 20700035.

```
select $tables where
  $tables in
    (select $tables where
       {Class:$class} in (select $assoc where {Association.(src|dest):$assoc} in $db),
       {is_persistent.Boolean:true} in $class,
       $dests in (select {Class:$dest} where {(src_of.Association.dest.Class)+:$dest} in $class),
       $related in ({Class:$class} U $dests),
       $cols in (select {cols:{Column:{name:$n,type:$t}}} where {Class.attrs.Attribute:{name:$n,type:$t}} in $related),
       $tables in (select {Table:{name:$cname} U $cols} where {name:$cname} in $class),
       $tables in (extend $table with $pkeys U $fkeys where
                     {Table:$table} in $tables,
                     {cols:$cols} in $table,
                     {Column.name.String:{$cname:{}}} in $cols,
                     $pkeys in (select {pkey:$cols} where
                                  {attrs.Attribute: {is_primary.Boolean:true, name.String:{$pname:{}}}} in $class,
                                  $cname = $pname),
                     $fkeys in (select {fkeys:{Fkey:{cols:$cols, ref:$ref}}} where
                                  {Class:{is_persistent.Boolean:true,
                                          attrs.Attribute.name.String:{$aname:{}}, name:$ref}} in $dests,
                                  $cname = $aname))),
  $tables in (replace $ref by {Table:$table} where
                {Table.fkeys.Fkey.ref:$ref, Table:$table} in $tables,
                {String:{$rname:{}}} in $ref,
                {name.String:{$tname:{}}} in $table,
                $tname = $rname)
```

Figure 10. Class2RDBMS in UnQL$^+$

References

[1] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS 2006: Proceedings of the 9th nternational Conference on Model Driven Engineering Languages and Systems*, pages 692–706. Springer-Verlag, 2006.

[2] M. Antkiewicz and K. Czarnecki. Design space of heterogeneous synchronization. In *GTTSE '07: Proceedings of the 2nd Summer School on Generative and Transformational Techniques in Software Engineering*, 2007.

[3] J. Bezivin, B. Rumpe, and T. L. Schürr A. Model transformation in practice workshop announcement. In *MTiP 2005, International Workshop on Model Transformations in Practice*. Springer-Verlag, 2005.

[4] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In G. C. Necula and P. Wadler, editors, *POPL*, pages 407–419. ACM, 2008.

[5] P. Braun and F. Marschall. BOTL : The bidirectional object oriented transformation language. Technical Report TUM-I0307, Technische Universität München, 2003.

[6] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases*, 9(1):76–110, 2000.

[7] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[8] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *Fundamental Approaches to Software Engineering*, pages 72–86. 2007.

[9] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice*. Springer-Verlag, 2005.

[10] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05 : ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 233–246, 2005.

[11] M. Garcia. Bidirectional synchronization of multiple views of software models. In *Proceedings of DSML-2008*, volume 324 of *CEUR-WS*, pages 7–19, 2008.

[12] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Models '06: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 543–557. Springer Verlag, October 2006.

[13] J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, 1998.

[14] S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards compostional approach to model transformations for software development. Technical Report GRACE-TR08-01, GRACE Center, National Institute of Informatics, Aug. 2008.

[15] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

[16] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.

[17] F. Jouault and I. Kurtev. Transforming models with ATL. In *Proceedings of Satellite Events at the MoDELS 2005 Conference*, pages 128–138. LNCS 3814, Springer, 2006.

[18] F. Klar, A. Königs, and A. Schürr. Model transformation in the large. In I. Crnkovic and A. Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 285–294. ACM, 2007.

[19] A. Konigs and A. Schurr. Tool integration with triple graph grammars - a survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113–150, February 2006.

[20] R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, Nov. 2004.

[21] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and a. Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 47–58. ACM Press, Oct. 2007.

[22] OMG. MOF QVT final adopted specification. `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

[23] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *J. Comput. Syst. Sci.*, 5(6):560–595, 1971.

[24] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE*, pages 572–581, 1999.

[25] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proc. 10th MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.

[26] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE*, volume 3062 of *LNCS*, pages 446–453. Springer, 2003.

[27] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 164–173. ACM Press, Nov. 2007.