GRACE TECHNICAL REPORTS

Towards Compositional Approach to Model Transformation for Software Development

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

GRACE-TR-2008–01 August 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/

Towards Compositional Approach to Model Transformation for Software Development

Soichiro Hidaka† Zhenjiang Hu† Hiroyuki Kato† Keisuke Nakano‡

†National Institute of Informatics
{hidaka,hu,kato}@nii.ac.jp

‡The University of Electro-Communications
ksk@cs.uec.ac.jp

August 2008

Abstract

Model transformation plays an important role in Model-Driven Software Development that aims to introduce significant efficiencies and rigor to the theory and practice of software development. Although models may have different notation and representation, they are basically graphs, and model transformations are thus nothing but graph transformations. Despite a large amount of theoretical work and a lot of experience with research prototype on graph-based model transformations, it remains as an open issue how to compose model transformations. In this paper, we report our first attempt at a compositional framework for graph-based model transformations based on the graph query language UnQL. We show that the idea of UnQL that graph queries are fully captured by structural recursion can be adapted to structure graph transformations to attain efficient composition of model transformations. We have implemented a prototype of the framework and tested with several nontrivial examples. Our new framework supports systematic development of model transformation in the large, while guaranteeing that inefficiency due to this composition is automatically removed.

1 Introduction

Model-driven software development [11] is an emerging technology that aims to introduce significant efficiencies and rigor to the theory and practice of software development. MDSD advocates models as 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; each model addresses one concern, and the transformations between models provide a chain that enables the automated development of a system from its corresponding models.

Model transformation specification, implementation and execution are the critical parts in model transformation [8]. Although models may have different notation and representation, they are basically graphs, and model transformations are thus nothing but graph transformations. This has led to the so-called graph-based approach [10, 14] to model transformations based on heavily theoretical work in graph transformations [1, 7, 21]. The graph-based approach is powerful with a large amount of theoretical work and a lot of experience with research prototypes. However, it remains as a challenge to use it to develop model transformation in the large, which requires a composition mechanism with high modularity [8]. In a recent survey paper [10], it says:

> *Open issues for all graph transformation approaches are elaborated concepts to compose transformations ...*

Put it more concretely, the problems are:

- First, although the graph-based approach is declarative in the sense that a transformation is specified by a set of graph rewriting rules, there lacks a good support for composing model transformations so that a set of new graph rewriting rules can be efficiently derived from those of two transformations that are composed.

- Second, the graph-based approach is very complex, which stems from the non-determinism in scheduling and application strategy of transformation rules, which requires careful consideration of termination of the transformation process and the rule application ordering (including the property of confluence). In most systems based on graph transformations, a graph rewriting rule is not executable unless it is accompanied with a complex rewriting engine.

From the practical point of view, model composition would be necessary if one wants to chain and combine model transformations to produce new and more powerful transformations. To bridge large abstraction gaps between two models, it is often much easier to generate intermediate models rather than go straight to the target model. This would make model transformation more modular and maintainable.

In this paper, we report our first attempt at a compositional framework for graph-based model transformations, which cannot only support concise specification of model transformation, but also simplify and improve efficiency of model transformation implementation and execution. This work was greatly inspired by the compositional graph querying language UnQL [5], which has been intensively studied in the database community. The key idea of UnQL is that graph queries are fully captured by structural recursion that are suitable for efficient composition. We show that this idea can be adapted to structure graph transformations to gain efficient composition too. Our main contributions are two folds.

First, we propose a compositional framework for graph-based model transformations based on the graph querying language UnQL. We have made three important extensions over UnQL.

- We add the graph schemes to UnQL and give an efficient validation algorithm, so that the meta model, an important component in model transformation, can be described.

- We extend UnQL with three simple graph editing constructs so that model transformation can be directly and convenient described.
- We show that all model transformations in the extended UnQL, called UnQL$^+$, can be mapped to structural recursions that are suitable for efficient composition.

Second, we have implemented a prototype of the new framework and tested with several nontrivial examples. Our new framework cannot only support systematic development of model transformation in the large, but also guarantee that inefficiency due to this composition can be automatically removed.

- We demonstrate, with the nontrivial model transformation from classes to relational database management system, that a large model transformation can be systematically developed by gluing simpler model transformations.
- We show that by representing model transformations internally by structural recursions or their composition, we can automatically eliminate inefficiency due to the introduction of composition by fusion optimization. The experimental results show promising speedups by fusion optimization.

The organization of this paper is as follows. We start by considering a typical but nontrivial model transformation, called Class2RDBMS, which will be served as our running example, in Section 2. Then we show how UnQL and its extension can be useful for systematic development of model transformations in a compositional style in Section 3, and we explain the architecture of our compositional framework and its detailed implementation in Section 4. We discussed the related work in Section 5, and conclude the paper in Section 6.

2 An Example: Class2RDBMS

As a running example, we consider the model transformation, Class2RDBMS, a simplified version of the well known ”class to RDBMS” transformation. It was proposed as a common example to all the participants of the International Workshop on Model Transformations in Practice 2005 [3], whose purpose was to compare and contrast various kinds of approaches to model transformations. We shall explain the requirement of this model transformation in this section, and leave the details of how this model transformation can be described in our framework in Section 3.3.

2.1 Class Models

A class model consists of classes and directed associations. A class 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). Associations are used to associate two classes. Figure 1 shows a class model, which consists of three classes and two directed associations.

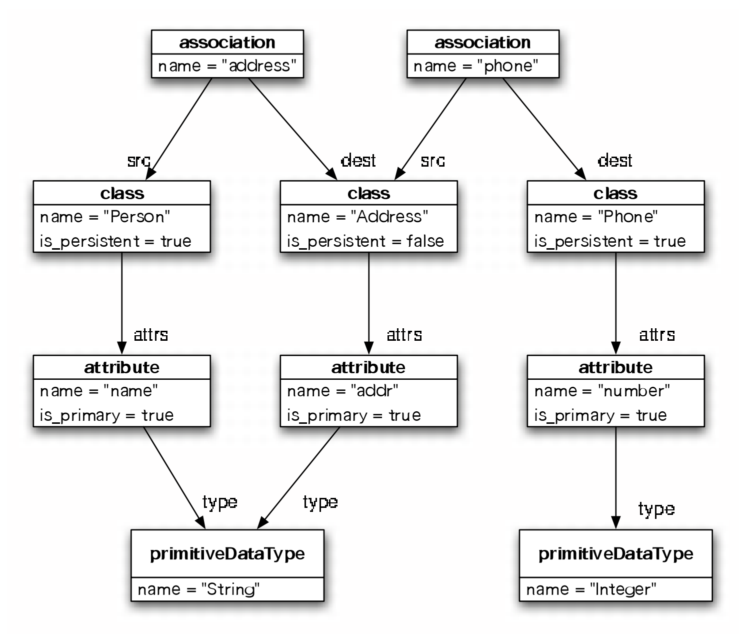

Figure 1: A Class Model

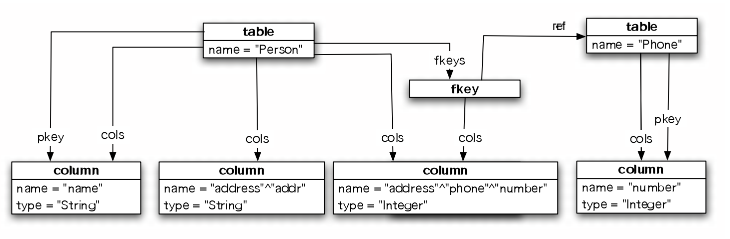

Figure 2: A RDBMS Model

2.2 RDBMS Models

An RDBMS model consists of one or more tables. A table consists of one or more columns. One or more of these columns will be included in the pkey slot of a table, denoting that the column forms part of the tables primary key slot. A table may also contain zero or more foreign keys. Each foreign key refers to the particular table it identifies, and denotes one or more columns in the table as being part of the foreign key. Figure 2 shows a RDBMS model that has two tables.

2.3 Transforming Class Models to RDBMS Models

Classes can be indicated as persistent or non-persistent. A persistent class is mapped to a table and all its attributes or associations to columns in this table. If the type of an attribute is primary, a primary key to from the table to the column is established. If the type of an attribute or association is another persistent class, a foreign key to the corresponding table is established.

If class hierarchies are transformed, only the topmost classes are mapped to tables. Additional attributes and associations of subclasses result in additional columns of the top-most classes.

Non-persistent classes are not mapped to tables, however, one of the main requirements for the transformation considered is to preserve all the information in the class diagram. That means attributes and associations of non-persistent classes are distributed over those tables stemming from persistent classes which access non-persistent classes.

This model transformation is not so trivial. We will show in Section 3.3 how to systematically develop it in our compositional framework.

3 Model Transformations in UnQL$^+$

In this section, we shall explain the language UnQL$^+$, a small extension of the graph query language UnQL [5], and show how it can used to describe model transformation in a compositional manner. First, we review the basic concepts on UnQL and its core UnCAL. Then, we extend UnQL to UnQL$^+$ with three editing operations. Finally, we demonstrate how to systematic develop model transformations with the example of Class2RDBMS.

3.1 UnQL: A Graph Querying Language

Our compositional framework for model transformations is based on UnQL [5], a language that was originally designed for querying unstructured data such as graphs. It has convenient select-where style surface syntax, which are translated into core graph algebra called UnCAL that consists of a small number of basic constructors and operators. Its expressive power is FO(TC) (first order with transitive closure), and complexity in answering UnQL query is PTIME. In this section, we briefly review the basic concepts of UnQL, which will serve as the basis of our framework.

3.1.1 Graph Representation

In UnQL, graphs are edged-labelled in the sense that all information is stored as labels on edges rather than on nodes (the labels on nodes have no particular meaning). It can be directly used to represent model. For example, Figure 3 shows a graph that represents the model in Figure 1. Formal definition of this graph representation will be given in Section 3.1.4.

3.1.2 UnQL

UnQL, like other query languages, has a convenient select-where structure for extracting information from a graph. We omit the formal definition of the language syntax, which can be found in Figure 13 in the Appendix. Rather we give some examples.

The following query `Q1` extracts all the primitive data types from database (denoted `db` in the query) in Figure 3.

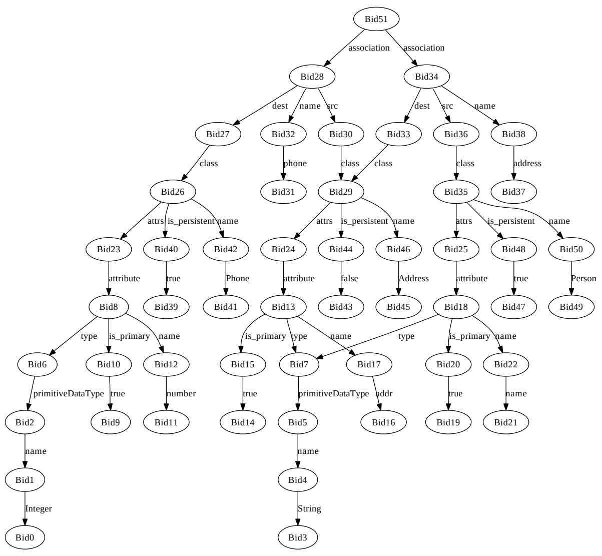

Figure 3: A Class Model Represented by an Edge-Labelled Graph

```
 (* Q1 *)
select T where
 {association:{dest:
   {class:{attrs:
    {attribute:{type:
      {primitiveDataType.name:T}}}}}}}
 in db
```

If we use the regular path pattern, all primitive data types that occur anywhere in the hierarchy can be easily obtained by the following query Q2:

```
 (* Q2 *)
select T where
  {_*.primitiveDataType.name:T} in db
```

More involved examples can be found in Section 3.3.

3.1.3 Structural Recursion in UnQL

Structural recursion plays a very important role in UnQL. Not only can it be used to described many useful queries, but also any queries in UnQL can be described in terms of structural recursion.

Structural recursive function f in UnQL is a simple mutually recursive computation scheme, satisfying the following two equations and guarantee that no return value of any function should be fed to another function.

$$\begin{array}{lcl} f\ \{\} & = & \{\} \\ f\ (t_1 \cup t_2) & = & f(t_1) \cup f(t_2) \end{array}$$

This simplicity allows manipulability of structural recursion which is a combinator that is similar to the higher-order function map in functional programming languages. Whereas map is applied recursively to tails, structural recursion is applied (vertically) to nodes, as well as (horizontally) to edges.

As a simple use of structural recursion, the following query Q3 replaces all labels `name` under `primitiveDataType` in Figure 3 with `typeName`. Due to the two equations above, definitions for horizontal recursion are always omitted.

```
 (* Q3 *)
select
 letrec
     sfun f1 ({primitiveDataType:T})
               = {primitiveDataType:g1(T)}
        | f1 ({L:T})    = {L:f1(T)}
 and sfun g1 ({name:T}) = {typeName:g1(T)}
        | g1 ({L:T})    = {L:g1(T)}
    in f1(db)
```

3.1.4 Data Model

UnQL data model is based on edge labeled, rooted, directed cyclic graphs, whose orders between outgoing edges of a node are insignificant. Nodes may be marked with input and output marker, both are denoted by $\&x \in \mathit{Marker}$, where Marker is an infinite set of symbols. A node marked with input and output marker is called input node and output node, respectively. Input markers are used to select entry point of the graph, whereas output markers are used to glue output nodes with input nodes of a graph.

Formally, a graph g is denoted by a quadruple (V, E, I, O), where V is a subset of (possibly infinite) set of nodes $\hat{V}$, a set of edges $E \subseteq V \times \mathit{Label}_\epsilon \times V$ where Label stands for infinite set of labels, and we denote $\mathit{Label} \cup \{\epsilon\}$ by Label_ϵ., a set of pairs of input marker and associated node $I \subseteq \mathcal{X} \times V$, and a set of pairs of output nodes and associated output marker $O \subseteq V \times \mathcal{Y}$. Each of these component set of the quadruple is denoted by $g.V$, $g.E$, $g.I$ and $g.O$, respectively. Since correspondence between input node and input marker in I is one-to-one, $I(\&x)$ denotes an input node labeled with $\&x$. On the other hand, more than one node can be marked with an identical output marker. The root marker, denoted by special input marker & represents default input node of a graph. $DB^{\mathcal{X}}_{\mathcal{Y}}$ represents a set of data graphs that has set of input markers $\mathcal{X}$ and output markers $\mathcal{Y}$. We use $DB_{\mathcal{Y}}$ as an abbreviation of $DB^{\{\&\}}_{\mathcal{Y}}$.

For example, a graph $g \in DB_{\{\&y\}}$ shown in Figure 4 is represented by

$$(\{1,2,3\}, \{(1,a,2),(2,c,2),(2,b,3)\}, \{(\&,1)\}, \{(3,\&y)\}).$$

Edges labeled with ϵ works like ϵ transition in automaton in that it identifies source and destination nodes. They are used in establishing connection between nodes. Detailed usages are given in Section 4.3.

3.1.5 UnCAL: A Graph Algebra

While UnQL is an interface language for users to write queries, UnCAL is its core language for internal implementation. UnCAL has a set of constructors

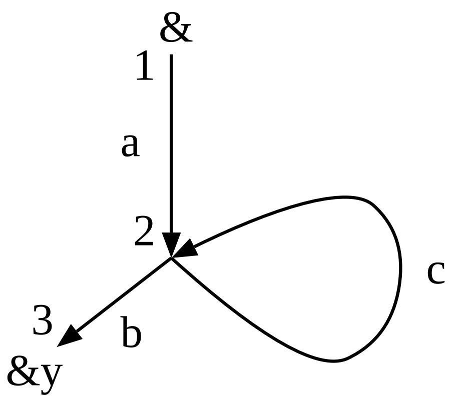

Figure 4: A simple graph example

and operators, by which arbitrary graphs can be represented. In addition to tree constructors, graph concatenation and cycle operator, together with input and output markers form cycles and confluences by gluing nodes marked with identical markers together.

Complete syntax and brief semantics of UnCAL expressions are depicted in Figure 14 in the Appendix.

Contrary to appearance of tree constructor {} and ∪, its semantics of unification is far from those of sets. In UnCAL, although value equality is explicitly defined, duplicate eliminations do not take place. A graph shown in Figure 4 is represented by the following UnCAL expression

```
&z1@cycle((&z1 := {a:&z2},
           &z2 := {c:&z2,b:&z3},
           &z3 := &y))
```

where `&z1, &2, &3` correspond to the three nodes respectively, and `a, b, c` correspond to the three edges.

3.1.6 Extended Graph Bisimulation and Bisimulation Genericness

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 in 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 $G_1.I(\&x)$ and $G_2.I(\&x)$, (4) requiring correspondence between output labels of corresponding nodes (output labels may be associated with the node other than corresponding nodes, provided that the label 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. Evaluation orders and strategies may introduce divergence in results. Instead of normalizing these graph to absorb these divergence, UnQL uses a relation called bisimulation genericness to establish equivalence class between them: if a function f is bisimulation generic, then for every component-wise bisimilar pair of n-tuples $(g_1, \ldots, g_n)$ and $(g'_1, \ldots, g'_n)$, $f(g_1, \ldots, g_n)$ and $f(g'_1, \ldots, g'_n)$ are bisimilar.

Semantics of UnCAL basic constructors and operators are carefully designed to satisfy bisimulation genericness, so that in tern UnCAL queries as a hole also are bisimulation generic. This allows safe application of various optimization including fusion and tupling.

3.2 UnQL$^+$

UnQL$^+$ is a small extension of UnQL to support convenient specification of graph transformations (model transformations). we extend UnQL with three editing constructs for transforming graphs. In Section 4.2, we will show that all these editing constructs can be mapped to structural recursions of UnCAL, the core language.

3.2.1 Deleting a Graph

The deletion construct, `delete ... where ...`, is used to describe deletion of part of the graph. Consider the class graph in Figure 3, and suppose we want to eliminate all the names of association. This can be described by

```
delete AssocName
where
 {association.name: AssocName} in db
```

where the subgraph matched with AssocName will be deleted from its original graph. In contrast, the following transformation keeps the association names as result.

```
select {result: AssocName}
where
 {association.name: AssocName} in db
```

So, we may consider the `delete` as the dual of the `select`.

3.2.2 Extending a Graph

The extension construct, `extend ... where ...`, is used to extend a graph with another one. For example, we may write the following transformation to add a `date` to each `association`.

```
extend AG with {date:"2008/8/4"}
where
 {association: AG} in db
```

3.2.3 Updating a Graph

The replacement construct, `replace ... where ...`, is used to replace a subgraph by a new graph. For example, the transformation of replacing the edge label `dest` by `tgt` can be specified as follows.

```
replace G by {tgt:G1}
where
 {association: G} in db,
 {dest: G1} in G
```

3.3 Class2RDBMS in UnQL$^+$

Now we demonstrate, with the example of Class2RDBMS, the usefulness of UnQL$^+$ in constructing complicated model transformation. The compositional style 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.

Recall the requirement of the transformation Class2RDBMS in Section 2, where we want to make tables (independent tables or tables pointed by a foreign key) from a class diagram, where each table should have a name, some a sequence of columns, some of which are pointed by primary or foreign keys. The whole transformation in UnQL$^+$ is given in Figure 6. Let us explain how it is developed.

It follows directly from the requirement of Class2RDBMS that the top level transformation can be described as follows.

```
select
   {table: {name: {Name: {}}} U
           MakePKeyCol U
           MakeGenCol U
           MakeFKeyCol,
    table: MakeFKeyTable}
where ...
```

Now to create columns of a table, we need to gather all information of classes that are directly or indirectly associated with the source persistent class. This suggests us to create an intermediate model `ChainDB`, in which indirectly associated classes are directly associated.

```
ChainDB in
(select
  ...
 where
  {association:
    {name: N1, src: C11, dest: C12}} in db,
  {association:
    {name: N2, src: C21, dest: C22}} in db,
    {name: {Name12: Any}} in C12,
    {name: {Name21: Any}} in C21,
    Name12 = Name21)
```

We look for two associations in which one's destination is the other's source, and then add a link edge to the indirected destination to the source. The detailed definition of `ChainDB` is given in Figure 6. Note that we do not need to worry about the relationship between the new and the old models, and this new model is just for intermediate use.

With `ChainDB`, it is easy to "query" the graph to extract information from each top (source) class that is persistent for creating a table later.

```
{group:
  {src:
    {name: {Name: Any},
     is_persistent: {Persistent: Any},
     attrs: As},
    chains: Chains}} in ChainDB,
Persistent = true,
```

Note that the symbols starting with a capital character are pattern variables used to save extracted information.

From the information obtained, we can create a primary key for the table by querying the data from the graphs and add two new edges `pkey` and `cols`.

```
MakePKeyCol in
  (select {pkey: Col, cols: Col}
   where
     {attribute: ...} in As,
     Primary = true,
     Col in {column: ...})
```

Note that `Col` is another shared intermediate model (graph), which appears twice in the `select` part.

We omit explanation of definitions for creating other columns and foreign keys, which are very similar.

As seen from this example, UnQL$^+$ enables us to productively develop model transformations in a compositional manner (we can glue results with unison operator `U` or sequentially apply simpler model transformations with some intermediate models.) This good result is not surprising, because usefulness of composition has been seen and widely known in development of program transformation.

The execution of this model transformation on the class graph in Figure 3 yields the graph in Figure 5, which is essentially the same as the table diagram in Figure 2.

4 Compositional Model Transformation System

Figure 7 shows an overview of our model transformation system. An input model represented by a graph is validated against a given input metamodel described in Kernel Metametamodel (KM3) [2]. The validated graph is transformed by a given query described in UnQL and an update described in UnQL$^+$ which will be introduced later. The transformation is performed by translating them into an UnCAL program, that is a structural recursion over an input graph, after integrating the UnQL$^+$ updating into an UnQL query. The output graph of the transformation is validated against a give output metamodel in KM3.

4.1 Graph Schema and Validation

Our system validates input and output graphs against given schemata of them. We employ Kernel MetaMetaModel (KM3) to describe schemata because it is widely used as a metametamodel[1] in actual software development and more

[1]A language for describing schemata is called a metametamodel while a schema is called a metamodel.

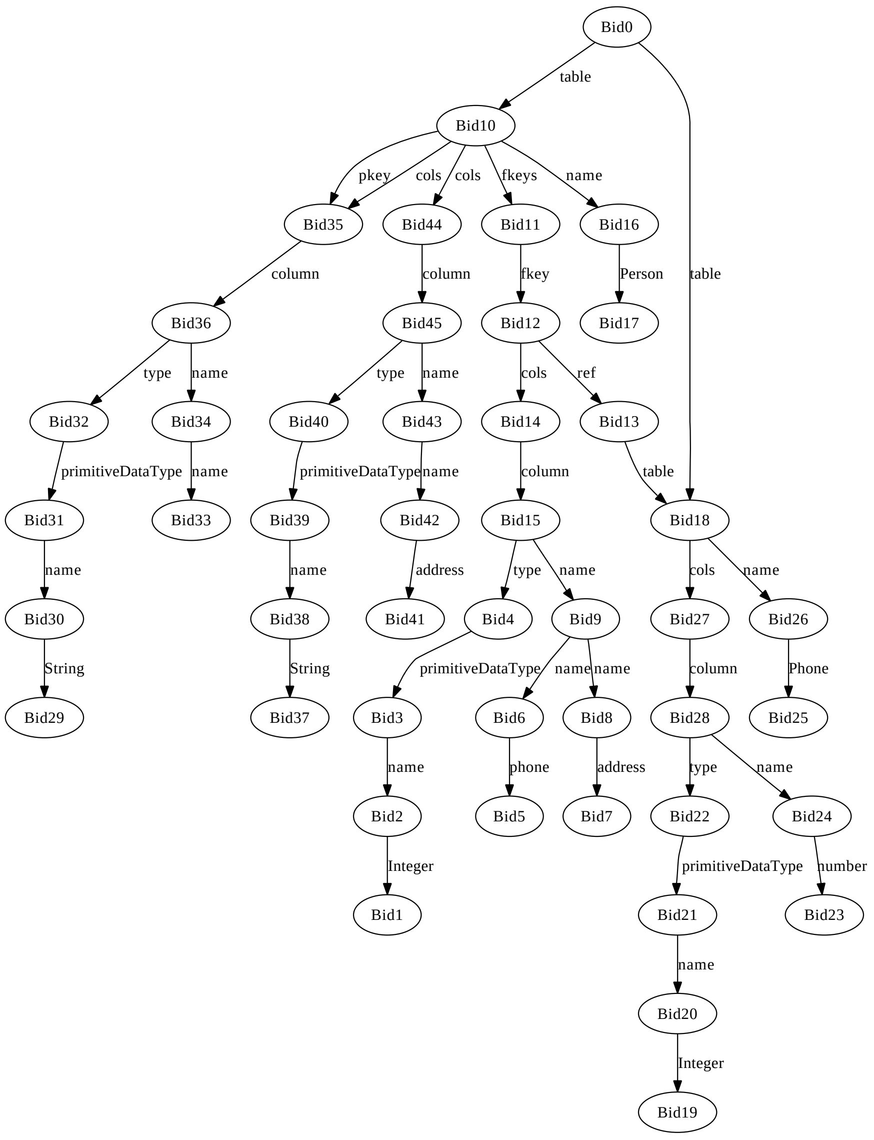

Figure 5: A Table Model Represented by an Edge-Labelled Graph

```
select
   {table: {name: {Name: {}}} U
           MakePKeyCol U
           MakeGenCol U
           MakeFKeyCol,
    table: MakeFKeyTable}
where
   ChainDB in
      (select {group: {src:C11,
                       chains:{dest:{cnames:{name: N1}, class:C12},
                               dest:{cnames:{name: N1, name:N2}, class:C22}}},
       group: {src:C21,
               chains:{dest:{cnames:{name:N2}, class:C22}}}}
       where {association: {name: N1, src: {class: C11}, dest: {class: C12}}} in db,
             {association: {name: N2, src: {class: C21}, dest: {class: C22}}} in db,
             {name: {Name12: Any}} in C12,
             {name: {Name21: Any}} in C21,
             Name12 = Name21),

   {group: {src: {name: {Name: Any},
                  is_persistent: {Persistent: Any},
                  attrs: As},
     chains: Chains}} in ChainDB,
   Persistent = true,

   MakePKeyCol in
      (select {pkey: Col, cols: Col}
       where {attribute: {name: AName, type: AType, is_primary: {Primary: Any}}} in As,
              Primary = true,
              Col in {column: {name: AName, type: AType}}),

   MakeGenCol in
      ((select {cols: {column: {name: AName, type: AType}}}
        where {attribute: {name: AName, type: AType, is_primary: {Primary: Any}}} in As,
              Primary = "false") U
       (select {cols: {column: {name: CNames, type: AType2}}}
        where  {dest:{cnames: CNames,
                   class:{is_persistent: {Persistent:Any},
                             attrs:As2}}} in Chains,
       Persistent = false,
               {attribute: {type: AType2}} in As2)),

   {dest:{cnames: CNames,
          class:{is_persistent: {Persistent:Any},
                 attrs:As2,
                 name: Name}}} in Chains,
   Persistent = true,
   {attribute: {type: AType2}} in As2,

   MakeFKeyTable in
      (select {name: Name, cols: Col}
       where {attribute: {name: AName, type: AType, is_primary: {Primary: Any}}} in As2,
              Col in {column: {name: AName, type: AType}}),

   MakeFKeyCol in {fkeys: {fkey: {cols: {name:CNames, type:AType2},
                                  ref: {table: MakeFKeyTable}}}}
```

Figure 6: Class2RDBMS in UnQL+

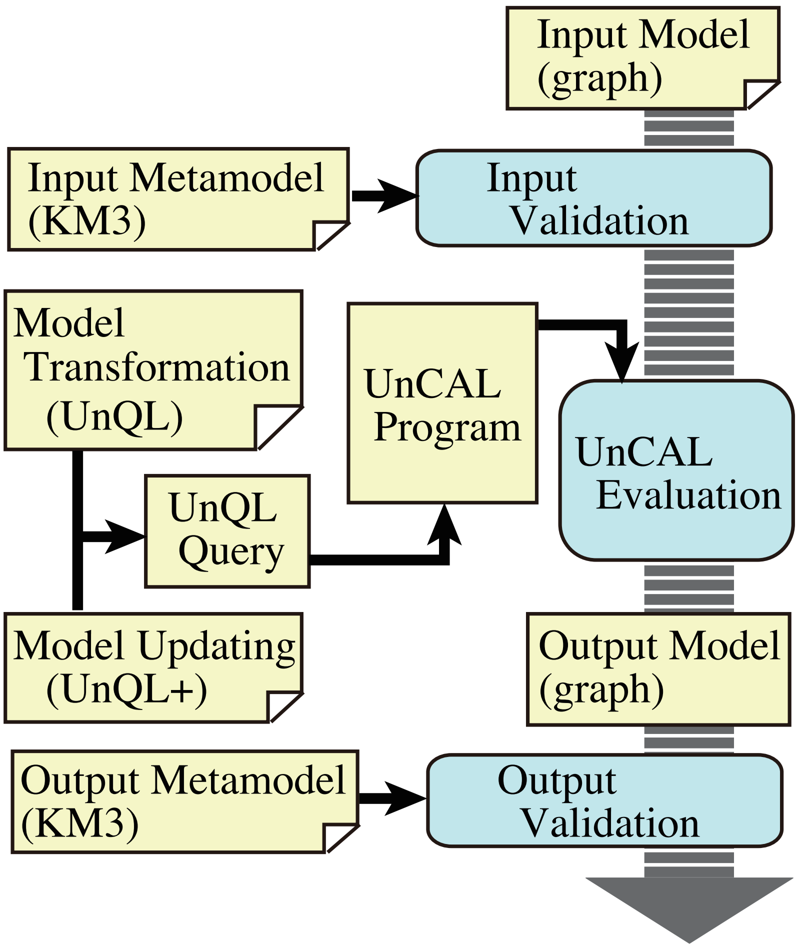

Figure 7: Overview of our System

formally defined than other metametamodels. We call *KM3 schema* for a metamodel written in KM3.

A KM3 schema has a structure similar to an XML schema base on regular tree grammar such as W3C XML schema [24] and RELAX NG [6] in the sense that a schema prescribes which kind of set of nodes must be referred to by a node by regular expression. See [2] for details on the specification of KM3.

Figure 8 shows an example of a KM3 schema for classes each of which is an input for the model transformation introduced in Section 2. The schema consists of four classes[2], `Association`, `Class`, `Attribute` and `PrimitiveDataType`. A class has some *features*, either *reference* or *attribute*. Every feature has a type, either class or data type. Since all of them inherit their super class `NamedElt`, they have an attribute `name` which is `String`. The `Association` class has two references `src` and `dest` which are `Class`. The `Class` class has an

[2]The term "class" here is used as an jargon of KM3. Do not confuse it with a "class" as an input of the model transformation in Section 2. The latter "class" appears as capitalized `Class` in a KM3 schema of Figure 8.

```
package Class {
  datatype String;
  datatype Boolean;
  abstract class NamedElt {
    attribute name : String;
  }
  class Association extends NamedElt {
    reference src : Class;
    reference dest : Class;
  }
  class Class extends NamedElt {
    attribute is_persistent : Boolean;
    reference attrs [*] : Attribute;
  }
  class Attribute extends NamedElt {
    attribute is_primary : Boolean;
    reference type : PrimitiveDataType;
  }
  class PrimitiveDataType
                extends NamedElt {
  }
}
```

Figure 8: KM3 Schema for Classes

attribute `is_persistent` which is `Boolean` and arbitrary number of references `attrs` which are `Attribute`. The `Attribute` class has an attribute `is_primary` which is `Boolean` and a reference `type` which is `PrimitiveDataType`. The `PrimitiveDataType` class has neither attribute nor reference besides an inherited attribute `name`.

We validate a graph by matching each edge in them with a name of a class or a feature in a given schema. A validation of a graph proceeds as follows.

1. All class inheritances are eliminated from a given schema by expanding features of classes with their super classes. The elimination is recursively done since a super class may inherit another super class.
2. We associate a vertex following from the root of the graph with a class whose name is the same as the label of the edge between the vertex and the root. For example, vertices Bid28 and Bid34 in Figure 3 are associated with a class `Association`.
3. We match a set of labels on edges from the vertex with a set of names of features of the class. Every destination vertex of the edge should have edges which is labelled by the name of a class or a data type of the feature and the number of which is specified by a multiplicity in the KM3 schema such as `[*]` in Figure 8. If the label of the edge is the name of the class, the destination vertex of the edge is associated with the class and is checked the feature again. If the label of the edge is the name of the data type, the destination vertex of the edge should have an edge whose label has the same type and whose destination vertex has no edge. This step is repeatedly performed until all vertices in the graph are visited.

This procedure always terminates because the number of vertices are finite.

Currently our system does not check the type of UnQL/UnCAL transformation, which is our future work. If it is attained, we do not have to validate either input or output graphs.

4.2 Mapping to the Core Language

UnQL$^+$ provides a friendly interface language for users to describe model transformations. For efficient implementation, UnQL$^+$ can be transformed to the core language UnCAL, where structural recursion plays an important role in supporting efficient composition of model transformations. The mapping from UnQL$^+$ to UnCAL consists of the following six steps: (1) simplifying where clauses; (2) removing the editing constructs; (3) transforming simple patterns to structural recursions; (4) transforming regular patterns to mutual structural recursions; (5) tupling mutual structural recursions to single ones; and (6) mapping structural recursions to those in UnCAL. In the following, we explain these steps one by one.

4.2.1 Simplifying Where Clauses

In the second step, (Step2) in Figure 9 transforms an UnQL query into one with only single patterns by applying (Rule1) to (Rule5) according to the patterns of *BindCond* in the where expression. Theses rules in Figure 9 are applied recursively based on the inductive definition of UnQL. Inference rules for a judgement of the form $t \xrightarrow{apstp2} t'$ are shown in Figure 15 in Appendix.

(Rule1) and (Rule3) are from the original paper. (Rule2) is to represent a compositional expression. (Rule4) and (Rule5) are to lift a constant leaf node up to an edge because in UnCAL data model all information is stored as labels on edges and, both *newv* and *anyv* are fresh variable names.

For example, on `Q1` described in Section 3 these rules produce:

```
(* Q1' *)
select T
where {association:V1} in db,
      {dest:V2} in V1,
      {class:V3} in V2,
      {attrs:V4} in V3,
      {attribute:V5} in V4,
      {type:V6} in V5,
      {primitiveDataType.name:T} in V6
```

In this query, the pattern

```
{association:{dest:{class:{attrs:{attribute:
       {type:{primitiveDataType.name:T}}}}}}}
```

in the *BindCond* in `Q1` is split into single patterns using fresh variable names `V1...V6`.

4.2.2 Eliminating Editing Constructs

We show informally how to map the three newly introduced editing constructs (Section 3.2) to structural recursions.

$$\frac{v :: \mathit{Var} \quad \{pe_i : pat_i\}\ \mathsf{in}\ v :: \mathit{BindCond} \xrightarrow{pat} bs_i :: \mathit{BindCond\,list} \quad (i = 1, \ldots, n)}{\{pe_1 : pat_1, \ldots, pe_n : pat_n\}\ \mathsf{in}\ v :: \mathit{BindCond} \xrightarrow{pat} bs_1 +\!\!+ \cdots +\!\!+ bs_n :: \mathit{BindCond\,list}} \tag{\textsc{Rule1}}$$

$$\frac{\begin{array}{c} \mathit{newv} :: \mathit{Var} \quad t :: \mathit{Template} \xrightarrow{apstp2} t' :: \mathit{Template} \\ \{pe_i : pat_i\}\ \mathsf{in}\ \mathit{newv} :: \mathit{BindCond} \xrightarrow{pat} bs_i :: \mathit{BindCond\,list} \quad (i = 1, \ldots, n) \end{array}}{\{pe_1 : pat_1, \ldots, pe_n : pat_n\}\ \mathsf{in}\ t :: \mathit{BindCond} \xrightarrow{pat} \mathit{newv}\ \mathsf{in}\ t', (bs_1 +\!\!+ \cdots +\!\!+ bs_n) :: \mathit{BindCond\,list}} \tag{\textsc{Rule2}}$$

$$\frac{\begin{array}{c} pat :: \{PE_1 : Pat_1, \ldots, PE_n : Pat_n\} \qquad t :: \mathit{Template} \xrightarrow{apstp2} t' :: \mathit{Template} \qquad \mathit{newv} :: \mathit{Var} \\ pat\ \mathsf{in}\ \mathit{newv} :: \mathit{BindCond} \xrightarrow{pat} bs :: \mathit{BindCond\,list} \end{array}}{\{pe : pat\}\ \mathsf{in}\ t :: \mathit{BindCond} \xrightarrow{pat} \{pe : \mathit{newv}\}\ \mathsf{in}\ t', bs :: \mathit{BindCond\,list}} \tag{\textsc{Rule3}}$$

$$\frac{c :: \mathit{Const} \qquad t :: \mathit{Template} \xrightarrow{apstp2} t' :: \mathit{Template}}{\{pe : c\}\ \mathsf{in}\ t :: \mathit{BindCond} \xrightarrow{pat} \{pe : \mathit{newv}\}\ \mathsf{in}\ t', \{c : \mathit{anyv}\}\ \mathsf{in}\ \mathit{newv} :: \mathit{BindCond\,list}} \tag{\textsc{Rule4}}$$

$$\frac{c :: \mathit{Const} \qquad t :: \mathit{Template} \xrightarrow{apstp2} t' :: \mathit{Template}}{c\ \mathsf{in}\ t :: \mathit{BindCond} \xrightarrow{pat} [\{c : \{\}\}\ \mathsf{in}\ t'] :: \mathit{BindCond\,list}} \tag{\textsc{Rule5}}$$

$$\frac{t :: \mathit{Template} \xrightarrow{apstp2} t' :: \mathit{Template} \qquad bc_1 :: \mathit{BC} \xrightarrow{bc} bs_1 :: \mathit{BC\,list} \qquad \cdots \qquad bc_n :: \mathit{BC} \xrightarrow{bc} bs_n :: \mathit{BC\,list}}{\mathsf{select}\ t\ \mathsf{where}\ bc_1, \ldots, bc_n :: \mathit{Query} \xrightarrow{step2} \mathsf{select}\ t'\ \mathsf{where}\ bs_1 +\!\!+ \cdots +\!\!+ bs_n :: \mathit{Query}} \tag{\textsc{Step2}}$$

$$\frac{bc :: \mathit{BindCond} \xrightarrow{pat} bs :: \mathit{BindCond\,list}}{bc <: \mathit{BC} \xrightarrow{bc} bs <: \mathit{BC\,list}} \qquad \frac{bc :: \mathit{BoolCond}}{bc <: \mathit{BC} \xrightarrow{bc} [bc] <: \mathit{BC\,list}}$$

Figure 9: Second step: Conjunctive patterns are split into single pattern in where expression

First, deletion or extension of a subgraph can be expressed by the `replace` construct based on the following two rules.

```
delete G where ... => replace G by {} where ...
extend G with G1 where ... => replace G by G U G1 where  ...
```

Second, the `replace` construct can be eliminated using the `select` construct and structural recursions. After simplification of the where clause, the where clause becomes a sequence of boolean conditions *bc* of relation expressions *r* such as `A=5`, simple pattern-in boolean expressions *pi* such as `{P:G1} in G2`, or simple binding expressions *bd* such as `G in template`. So the form to be transformed is

$$\texttt{replace } G_1 \texttt{ by } G_2 \texttt{ where } bc_1, \ldots, bc_{k-1}, \{pe : G_1\}, bc_{k+1}, \ldots, bc_n$$

where $bc_1, \ldots, bc_{k-1}$ are either relation expressions or pattern-in boolean expressions. Our transformation rules are as follows.

```
replace G1 by G2 where {l:{}} in D, r1,...,rm, rest
=>
let sfun h1{L:G} =
       if L=l and isEmpty(G) and r1 and ... and rm then
          replace G1 by G2 where rest
       else
          {L:G}
in h1(D}

replace G1 by G2 where {L:{}} in D, r1,...,rm, rest
=>
let sfun h1{L:G} =
       if isEmpty(G) and r1 and ... and rm then
          replace G1 by G2 where rest
       else
          {L:G}
in h1(D}

replace G1 by G2 where {l:G3} in D, r1,...,rm, rest
=>  { condition: G1 /= G3 }
let sfun h1{L:G3} =
       if L=l and r1 and ... and rm then
          {L:(replace G1 by G2 where rest)}
       else
          {L:G3}
in h1(D}

replace G1 by G2 where {L:G3} in D, r1,...,rm, rest
=>  { condition: G1 /= G3 }
let sfun h1{L:G3} =
       if r1 and ... and rm then
          {L:(replace G1 by G2 where rest)}
       else
          {L:G3}
in h1(D}

replace G1 by G2 where {l:G3} in D, r1,...,rm, rest
```

```
=>  { condition: G1 = G3 }
let sfun h1{L:G3} =
       if L=l and r1 and ... and rm then
          letval G1' = select {l:{}} where rest in
          letval G2' = select G2 where rest in
          if isEmpty(G1') then {L:G3} else {L:G2'}
       else
          {L:G3}
in h1(D}

replace G1 by G2 where {L:G3} in D, r1,...,rm, rest
=>  { condition: G1 = G3 }
let sfun h1{L:G3} =
       if r1 and ... and rm then
          letval G1' = select {l:{}} where rest in
          letval G2' = select G2 where rest in
          if isEmpty(G1') then {L:G3} else {L:G2'}
       else
          {L:G3}
in h1(D}

replace {L:G1} by G2 where {L:G3} in D, r1,...,rm, rest
=>  { condition: G1 = G3 }
let sfun h1{L:G3} =
       if r1 and ... and rm then
          letval G1' = select {l:{}} where rest in
          letval G2' = select G2 where rest in
          if isEmpty(G1') then {L:G3} else {G2'}
       else
          {L:G3}
in h1(D}
```

As an example, consider the following expression.

```
replace Name by Name'
where
     {association:Assoc} in db,
     {name:Name} in Assoc,
     {string:Na} in Name,
     {N:{}} in Na,
     N = "phone",
     Name' in {string:{"assoc"^N":{}}}
```

It can be desugared to the following.

```
let sfun h1{L:Assoc} =
       if L=association then
          let sfun h2 {L:Name} =
             if L=name then
                letval G1' = (select {name:{}}
                              where
                              {string:Na} in Name,
                              {N:{}} in Na,
                              N = "phone",
                              Name' in {string:{"assoc"^N":{}}}) in
                     letval G2' = (select G2
                              where
                              {string:Na} in Name,
                              {N:{}} in Na,
                              N = "phone",
                              Name' in {string:{"assoc"^N":{}}}) in
                     if isEmpty{G1') then {L:Name} else {L:G2'}
                  else {L:Name}
                in h2(Assoc)
             else {L:Assoc}
      in h1(db)
```

4.2.3 From Patterns to Structural Recursions

In the third step, we apply (Step3) in Figure 10. (Step3) transforms an UnQL, generated by the second step, into structural recursion with letval and filter expressions. In Figure 10, (RuleA) and (RuleD) are from the original paper. (RuleB) is used to represent binding values to variables by letval expressions newly introduced by us. (RuleC) is to represent Boolean conditions in where expressions by newly introduced filter expressions. Note that like the original paper, *rest* used in (RuleA),(RuleB) and (RuleC) is a syntactic meta-variable which stands for the remaining clauses in the where component. Theses rules in Figure 10 are also applied recursively based on the inductive definition of UnQL. Inference rules for a judgement of the form $t \xrightarrow{apstp3} t'$ are shown in Figure 16 in Appendix.

When applied to Q1', the result is:

```
let sfun h1({association:V1}) =
 let sfun h2({dest:V2}) =
  let sfun h3({class:V3}) =
   let sfun h4({attrs:V4}) =
    let sfun h5({attribute:V5}) =
     let sfun h6({type:V6}) =
      let sfun h7({primitiveDataType.name:T})
              = T
      in h7(V6)
     in h6(V5)
    in h5(V4)
   in h4(V3)
  in h3(V2)
 in h2(V1)
in h1(db)
```

4.2.4 Regular Path Patterns

In the fourth step, when the path patterns are regular path patterns, these regular path patterns can be translated into functions defined by structural recursion. Any regular path pattern can be translated into structural recursion, by expressing first the regular expression as an NFA(Non-deterministic Finite

$$\frac{\textsf{select}\, e \,\textsf{where}\, rest :: Query \xrightarrow{sr} t :: Template \qquad e' :: Template \xrightarrow{apstp3} t' :: Template}{\textsf{select}\, e \,\textsf{where}\, (\{pe : pat\} \,\textsf{in}\, e', rest) :: Query \xrightarrow{sr} \textsf{let sfun}\, h(\{pe : pat\}) = t \,\textsf{in}\, h(t') :: Template} \quad (\textsc{RuleA})$$

$$\frac{v :: Var \qquad \textsf{select}\, e \,\textsf{where}\, rest :: Query \xrightarrow{sr} t :: Template \qquad e' :: Template \xrightarrow{apstp3} t' :: Template}{\textsf{select}\, e \,\textsf{where}(v \,\textsf{in}\, e', rest) :: Query \xrightarrow{sr} \textsf{letval}\, v := t' \,\textsf{in}\, t :: Template} \quad (\textsc{RuleB})$$

$$\frac{v :: Var \qquad \textsf{select}\, e \,\textsf{where}\, rest :: Query \xrightarrow{sr} t :: Template}{\textsf{select}\, e \,\textsf{where}(bc, rest) :: Query \xrightarrow{sr} \textsf{filter}(bc, t) :: Template} \quad (\textsc{RuleC})$$

$$\frac{}{\textsf{select}\, e \,\textsf{where}() :: Query \xrightarrow{sr} e :: Template} \quad (\textsc{RuleD})$$

$$\frac{e :: Template \xrightarrow{apstp3} t :: Template \qquad \textsf{select}\, t \,\textsf{where}\, bs :: Query \xrightarrow{sr} t' :: Template}{\textsf{select}\, e \,\textsf{where}\, bs :: Query \xrightarrow{step3} t' :: Template} \quad (\textsc{Step3})$$

Figure 10: Third step: Patterns are transformed into structural recursion.

Automaton) and associating a function with each state. This transformation is achieved by the standard way in transformation from regular expressions to NFA without epsilon. The generated functions are mutually recursive. Note that unlike the original paper, function application associated with terminal states is unioned with not the argument of the function but an identity function application with the argument. This transformation enables us to apply the optimization technique, called fusion, described in Section 4.4.

For example, consider the regular expression `_*.primitiveDataType.name` in `Q2`, an equivalent non-deterministic automaton[3] has five states and the following transitions :

$$s_1 \xrightarrow{Any} s_4, s_1 \xrightarrow{Any} s_5, s_1 \xrightarrow{primitiveDataType} s_3,$$
$$s_3 \xrightarrow{name} s_2, s_4 \xrightarrow{primitiveDataType} s_3,$$
$$s_5 \xrightarrow{Any} s_4, s_5 \xrightarrow{Any} s_5, s_4 \xrightarrow{primitiveDataType} s_3$$

The initial state is s_1 and the terminal state is s_2. So, `Q2` is equivalent to the following mutual structural recursion, `Q2'`.

```
(* Q2' *)
letval T :=
  let
    sfun h1({primmitiveDataType:T'}) = h3(T')
       | h1({L:T'}) = h4(T') U h5(T')
    sfun h2({L:T'}) = {}
    sfun h3({name:T'}) = h2(T') U id(T')
       | h3({L:T'}) = {}
    sfun h4({primitiveDataType:T'})=h3(T')
       | h4({L:T'}) = {}
    sfun h5({primitiveDataType:T'})=h3(T')
```

[3]There are some equivalent automata, of course.

```
      | h5({L:T'}) = h4(T') U h5(T')
    sfun id({L:T'}) = {L:id(T')}
  in h1(db)
in T
```

In this query, each function corresponds to a state, and has one pattern for each symbol occurring on some transition from that sate. Since s_2 is the terminal state, `h2(T')`occurs in the right hand side with unioned with `id(T')`, where `id()`is an identity function.

4.2.5 Tupling Mutual Structural Recursions

In the fifth step, tupling is applied to mutually recursive functions. Tupling [15] is a standard way to transform mutual recursive functions into a single one by defining a new function that returns a tuple of results each corresponding to a function that is mutually defined with others. The tupling transformation of mutual structural recursions has been given in [5], so we omit the details but just give an example.

By applying the tupling to the mutually recursion `Q2'`, we can have the following single recursive function `h1h2h3h4h5id()`. Note that this is an illustrative expression, so the following expression has a mixture of UnQL and UnCAL syntax.

```
letval T :=
  let
    sfun h1h2h3h4h5id(name:T') =
          (&z1 := &z4 U &z5,
             &z2 := {},
               &z3 := &z2 U &z6,
                 &z4 := {},
                   &z5 := &z4 U &z5,
                     &z6 := {name:&z6})
       | h1h2h3h4h5id(primitiveDataType:T') =
          (&z1 := &z3,
             &z2 := {},
               &z3 := {},
                 &z4 := &z3,
                   &z5 := &z3,
                     &z6 := {primitiveDataType:&z6})
       | h1h2h3h4h5id(L:T') =
          (&z1 := &z4 U &z5,
             &z2 := {},
               &z3 := {},
                 &z4 := {},
                   &z5 := &z4 U &z5,
                     &z6 := {L:&z6})
  in &z1@h1h2h3h4h5id(db)
in T
```

4.2.6 Mapping to UnCAL

Finally, structural recursion translated using the above steps can be mapped to the UnCAL expressions. This mapping has been given in [5].

4.3 Interpretation of the Core Language

4.3.1 Interpreting Structural Recursion

UnQL paper [5] provides two evaluation strategies that are proved to be equivalent: *bulk* semantics and *recursive* semantics. The latter is intuitive in that applications of body expression (e_1 in $rec(e_1)(e_2)$) take place in a top-down fashion. Revisiting of nodes caused by cycles can be correctly handled by memoization. The former deals possible cycles by applying e_1 once for every edge in input graph and connect together using Skolem functions on markers and nodes. We describe here two peculiar aspects of our implementation in bulk semantics. Implementation of recursive semantics was fairly straightforward.

Skolem Functions In bulk semantics, graph is represented by independent set comprehensions on nodes, edges and markers. "Rendezvous" between them is required through Skolem functions: you have to glue nodes whose Skolem function values are identical. We implemented the function using a set of OCaml's constructor of algebraic data structure (variants) which directly reflects recursive nature of Skolem function (result of application of the function to node ID is again a node ID), so that there is a one-to-one mapping between Skolem function and constructor. It also allowed us almost straight-forward implementation of set comprehensions using standard Set library and fold operations on the Set instances.

Caching Values of the Body Expression Since we use identities of nodes encoded with instances of the algebraic data structures, and create fresh identities for each evaluation of tree constructors, each evaluation on e_1 produces "different" instances for identical inputs. Therefore, values of e_1 are obtained collectively beforehand and stored into tables. Looking up the cache value of the value of e_1 instead of evaluating in the set comprehensions allows correct rendezvous between nodes and edges, as well as elimination of recomputation.

4.3.2 Epsilon Edge Elimination

Bulk semantics generously introduces epsilon edges. They are eliminated when necessary using straightforward algorithm.

Static Type Estimation Regardless of evaluation strategy, *rec* requires static estimation of input and output markers of e_1. In UnCAL, this information is called type. Inference rules of the estimation, which may be non-trivial, is depicted in Figure 11.

Dynamic Semantics of UnCAL Figure 17 in Appendix shows concrete dynamic semantics of major UnCAL expressions except *rec*, which are already explained.

4.4 Model Compositions

We identify two forms of model composition. First one is a pair of consecutive transformations T_1 and T_2, where the output model of T_1 is the input model of

$$\frac{d \in DB^{\mathcal{X}}_{\mathcal{Y}} \;\; \mathcal{Y} \subseteq \mathcal{Y}'}{d \in DB^{\mathcal{X}}_{\mathcal{Y}'}} \quad \{\} \in DB_{\mathcal{Y}} \quad \frac{d \in DB_{\mathcal{Y}}}{\{l : d\} \in DB_{\mathcal{Y}}} \quad \frac{d_1, d_2 \in DB_{\mathcal{Y}}}{d_1 \cup d_2 \in DB_{\mathcal{Y}}} \quad \frac{d_1, d_2 \in DB^{\mathcal{X}}_{\mathcal{Y}}}{d_1 \cup d_2 \in DB^{\mathcal{X}}_{\mathcal{Y}}}$$

$$() \in DB^{\emptyset}_{\mathcal{Y}} \quad \frac{d \in DB_{\mathcal{Y}}}{\&x := d \in DB^{\{\&x\}}_{\mathcal{Y}}} \quad \frac{d \in DB^{\mathcal{Z}}_{\mathcal{Y}}}{\&x \cdot d \in DB^{\{\&x\}\cdot\mathcal{Z}}_{\mathcal{Y}}} \quad \frac{y \in \mathcal{Y}}{\&y \in DB_{\mathcal{Y}}} \quad \frac{d_1 \in DB^{\mathcal{X}_1}_{\mathcal{Y}} \;\; d_2 \in DB^{\mathcal{X}_2}_{\mathcal{Y}} \;\; \mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset}{d_1 \oplus d_2 \in DB^{\mathcal{X}_1 \cup \mathcal{X}_2}_{\mathcal{Y}}}$$

$$\frac{d_1 \in DB^{\mathcal{X}}_{\mathcal{Y}} \;\; d_2 \in DB^{\mathcal{Y}}_{\mathcal{Z}}}{d_1 \mathbin{@} d_2 \in DB^{\mathcal{X}}_{\mathcal{Z}}} \quad \frac{d_1 \in DB^{\mathcal{X}}_{\mathcal{X}\cup\mathcal{Y}}}{\mathit{cycle}(d) \in DB^{\mathcal{X}}_{\mathcal{Y}}}$$

$$\frac{e : \mathit{Label} \times DB_{\mathcal{Y}} \to \; DB^{\mathcal{Z}}_{\mathcal{Z}} \;\; d \in DB^{\mathcal{X}}_{\mathcal{Y}}}{\mathit{rec}(e)(d) \in DB^{\mathcal{X}\cdot\mathcal{Z}}_{\mathcal{Y}\cdot\mathcal{Z}}} \quad \frac{e_t \in DB^{\mathcal{X}}_{\mathcal{Y}} \;\; e_f \in DB^{\mathcal{X}}_{\mathcal{Y}}}{\textsf{if}\; b \;\textsf{then}\; e_t \;\textsf{else}\; e_f \in DB^{\mathcal{X}}_{\mathcal{Y}}}$$

Figure 11: UnCAL Typing Rules

T_2: $\mathcal{M}' = (T_2 \circ T_1)(\mathcal{M}) = T_2(T_1(\mathcal{M}))$. Second one is a pair of transformations T_1 and T_2, that share identical input model: $(\mathcal{M}_1, \mathcal{M}_2) = (T_1 \vartriangle T_2)(\mathcal{M}) \overset{\text{def}}{=} (T_1(\mathcal{M}), T_2(\mathcal{M})))$. In the first composition, intermediate result can be eliminated by fusion technique, while in the second composition, duplicate traversal of the input model can be unified by tupling technique. Since tupling at this level is not implemented, only fusion is described in this section.

4.4.1 Fusing Model Composition

In our framework, consecutive model transformations are translated into compositional UnQL queries. Since these queries are translated into composition of structural recursions in UnCAL, fusion transformation for UnCAL that is proposed in [5] is directly applicable. For very simple case, consider the following scenario[4]: first apply `Q3` to model in Figure 3, and then retrieve all `name`s by the following query `Q4`.

```
(* Q4 *)
select
 letrec sfun f2 ({name:T}) = {name:g2(T)}
          |  f2 ({L:T})    = f2(T)
 and   sfun  g2 ({L:T})    = {L:g2(T)}
 in f2(db))
```

Note that since these transformations are for illustrative purpose, KM3 validation to intermediate and final result may fail.

Compositional query would look like the following query `Q5`, by which our desugaring module produces an UnCAL query `Q6`. Our UnCAL rewriter translate it into `Q7`, which is equivalent to `Q8`, which is further simplified by hand. Two *rec*s in `Q6` is fused into one *rec* in `Q7`.

Table 1 shows execution times of `Q5` for each evaluation strategy of *rec*. The experiment was conducted on a 1.5GHz quad Xeon SMP machine running Linux kernel 2.4.20. About 3 to 5 fold speed-up had been confirmed.

```
 (* Q5 *)
```

[4]This scenario is derived from that of [5].

evaluation strategy of *rec*	before fusion	after fusion	speedup ratio
bulk	1.31	0.25	5.32
recursive	2.08	0.73	2.86

Table 1: Execution time [sec] of Q5

```
select
 letrec sfun f2 ({name:T}) = {name:g2(T)}
          |  f2 ({L:T})    = f2(T)
  and   sfun g2 ({L:T})    = {L:g2(T)}
  in
  letrec sfun f1 ({primitiveDataType:T})
              = {primitiveDataType:g1(T)}
           |  f1 ({L:T})   = {L:f1(T)}
  and   sfun  g1 ({name:T})= {typeName:g1(T)}
          |   g1 ({L:T})   = {L:g1(T)}
  in f2(f1(db))
```

```
 (* Q6 *)
&z1@rec(\ (L,T).
  if L = "name"
  then (&z1 := {"name": &z2},
        &z2 := {"name": &z2})
  else (&z1 := &z1, &z2 := {L: &z2}))
  (&z1@rec(\ (L,T).
    if L = "name"
    then (&z1 := {"name": &z1},
          &z2 := {"typeName": &z2})
    else if L = "primitiveDataType"
      then (&z1 := {"primitiveDataType": &z2},
            &z2 := {"primitiveDataType": &z2})
      else (&z1 := {L: &z1}, &z2 := {L: &z2}))
      (db))
```

```
 (* Q7 *)
&z1@(&z2 := &z1&z2, &z1 := &z1&z1)@
 rec(\ (Sa1,T).
  if Sa1="name"
  then (&z1 := (&z1 := {"name": &z2},
                &z2 := {"name": &z2})
     @ (&z2 := &z1&z2, &z1 := &z1&z1),
        &z2 := (&z1 := &z1,
                &z2 := {"typeName": &z2})
     @ (&z2 := &z2&z2, &z1 := &z2&z1))
  else if Sa1 = "primitiveDataType"
    then (&z1 := (&z1 := &z1,
                  &z2 := {"primitiveDataType": &z2})
       @ (&z2 := &z2&z2, &z1 := &z2&z1),
          &z2 := (&z1 := &z1,
                  &z2 := {"primitiveDataType": &z2})
       @ (&z2 := &z2&z2, &z1 := &z2&z1))
    else (&z1 := llet L = Sa1 in
     if L = "name"
     then (&z1 := {"name": &z2},
           &z2 := {"name": &z2})
     else (&z1 := &z1, &z2 := {L: &z2})
           @ (&z2 := &z1&z2, &z1 := &z1&z1),
              &z2 := llet L = Sa1 in
       if L = "name"
       then (&z1 := {"name": &z2},
             &z2 := {"name": &z2})
       else (&z1 := &z1, &z2 := {L: &z2})
          @ (&z2 := &z2&z2, &z1 := &z2&z1)))(db)
```

```
 (* Q8 *)
&z1@(&z2 := &z1&z2, &z1 := &z1&z1)@
 rec(\ (Sa1,T).
  if Sa1="name"
  then (&z1&z1 := {"name": &z1&z2},
        &z1&z2 := {"name": &z1&z2},
        &z2&z1 := &z2&z1,
        &z2&z2 := {"typeName": &z2&z2})
  else if Sa1 = "primitiveDataType"
    then (&z1&z1 := &z2&z1,
          &z1&z2 := {"primitiveDataType": &z2&z2},
          &z2&z1 :=  &z2&z1,
          &z2&z2 := {"primitiveDataType": &z2&z2}
    else (&z1 := llet L = Sa1 in
            if L = "name"
             then (&z1 := {"name": &z1&z2},
                   &z2 := {"name": &z1&z2})
             else (&z1 := &z1&z1,
                   &z2 := {L: &z1&z2}),
          &z2 := llet L = Sa1 in
            if L = "name"
            then (&z1 := {"name": &z2&z2},
                  &z2 := {"name": &z2&z2})
            else (&z1 := &z2&z1,
                  &z2 := {L: &z2&z2})
           ))(db)
```

The following two rules are main transformation rules for cascading *rec*'s. The first rule is applied when $e_2(l,t)$ does not depend on t. Second rule is applied otherwise.

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

Following rules are also used to make recursive application of the above fusion rules to subexpressions possible.

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

$$\frac{\mathcal{Z} = \{\&z_1, \ldots, \&z_p\} \quad e \in DB_{\mathcal{Z}}^{\mathcal{Z}}}{\mathit{rec}(e)(\&y) = (\&z_1 := \&y \cdot \&z_1, \ldots, \&z_p := \&y \cdot \&z_p)}$$

$$\&x := (\&z := e) \downarrow \&x.\&z := e \quad \&x := (e_1 \oplus e_2) \downarrow (\&x := e_1) \oplus (\&x := e_2)$$

$$e \cup \{\} \downarrow e \quad \{\} \cup e \downarrow e$$

$$e \oplus () \downarrow e \quad () \oplus e \downarrow e$$

$$() \mathbin{@} e \downarrow ()$$

$$\mathit{cycle}(()) \downarrow () \quad \mathit{cycle}(\{\}) \downarrow \{\} \quad \frac{e \in DB_{\mathcal{Y}}^{\mathcal{X}} \quad \mathcal{X} \cap \mathcal{Y} = \phi}{\mathit{cycle}(e) \downarrow e}$$

Figure 12: Auxiliary rewriting rules

$$\begin{array}{rcl} \mathit{rec}(e)() & = & () \\ \mathit{rec}(e)(d_1 \oplus d_2) & = & \mathit{rec}(e)(d_1) \oplus \mathit{rec}(e)(d_2) \end{array}$$

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

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

Figure 12 shows additional rules to further simplify the body of *rec*. There may be other rules applicable. Exploring these rules s a part of our future works.

5 Related Work

Our work is very much related to research on model transformation based on graph transformation in the software engineering community, as well as on research on graph querying in the database community.

In the software engineering community, graph transformation has been widely used for expressing model transformations [10, 19, 16].

AGG [23, 9] is a well-known rule-based visual tool which supports an algebraic approach to graph transformation. AGG supports typed (attributed) graph transformations including type inheritance and multiplicities. Rule application can contain non-deterministic choice of rules which may be controlled by rule layers. Different from our approach, AGG does not have a clear separation between the source and target graphs, it is not straightforward to compose/write multi-staged transformations in AGG.

Triple Graph Grammars (TGG) [17, 12] were proposed as an extension of Pratt's pair grammar approach [20], which aim at the declarative specification of model to model integration rules. TGGs consist of a schema and a set of graph rewriting rules, and they explicitly maintain the correspondence of two graphs by means of correspondence links. These correspondence links play the role of traceability links that map elements of one graph to elements of the other graph and vice versa. With TGG, one has to explicitly describe correspondence between the source and target models, which would be difficult if the transformation is complex and the intermediate models are required during the transformation.

Neither AGG nor TGG has strict control over application of elementary algebraic graph transformation rules. To increase usability and efficiency of graph transformation, a variety of control concepts for rule and match selection have been considered in many graph transformation approaches such as VIATRA [4] and VMTS [18], where graph transformations are controlled with recursive graph patterns. Unlike AGG and TGG, graph transformation rules are guaranteed to be executable, which is a main conceptual difference. Since their recursive control structures can be very complicated, it remains unclear how to efficiently compose them. Our approach puts reasonable restriction on the recursive structure so that it cannot only be powerful enough to specify various model transformations but also suitable for efficient composition.

On the other hand, in the database community, there have been a lot of work on language design and implementation for efficient graph querying [13, 22, 5]. Different from querying trees, issues on representation and equivalence of graphs are subtle and important to define correctness of graph querying (as well as graph transformation), and the use of bisimulation and structural recursion in [5] leads to a very nice framework for both declarative and efficient graph querying with high modularity and composability. This has motivated us to see if we can extend the framework from graph querying to graph transformation.

6 Conclusion

In this paper, we have reported our first attempt to designing and implementing a compositional framework for model transformations based on UnQL. Although UnQL is well known in database community for its unique solution to the composition problem, no one, as far as we are aware, has recognized its usefulness in software development. We have shown that it is indeed useful and the main theory and technique can be applied to solve the composition problem in model transformations.

We are currently working on extending this framework further to add "bidirectionality" to the compositional model transformation so that updating the target model can be reflected in the source model. This would combine the interesting idea on bidirectional computation in both programming language and software engineering communities.

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.

References

[1] Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. In *704*, page 55. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 30 1995.

[2] ATLAS group. KM3:Kernel MetaMetaModel manual. http://www.eclipse.org/gmt/atl/doc/.

[3] J. Bezivin, B. Rumpe, and Tratt L Schürr A. Model transformation in practice workshop announcement. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005).* Springer-Verlag, 2005. http://sosym.dcs.kcl.ac.uk/events/mtip/.

[4] Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors. *Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003, Proceedings*, volume 2589 of *Lecture Notes in Computer Science.* Springer, 2003.

[5] Peter Buneman, Mary F. Fernandez, and Dan 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.

[6] James Clark and Makoto Murata. RELAX NG specification. http://www.oasis-open.org/committees/relax-ng/spec.html, 2001.

[7] Andrea Corradini and Fabio Gadducci. A 2-categorical presentation of term graph rewriting. In *Category Theory and Computer Science*, pages 87–105, 1997.

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

[9] Hartmut Ehrig, Karsten Ehrig, Gabriele Taentzer, Juan de Lara, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Transformation Techniques in Software Engineering*, volume 05161 of *Dagstuhl Seminar Proceedings.* Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[10] 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).* Springer-Verlag, 2005.

[11] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley & Sons, 2003.

[12] Holger Giese and Robert Wagner. Incremental model synchronization with triple graph grammars. In Oscar Nierstrasz, John Whittle, David Harel, and Gianna Reggio, editors, *Models '06: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 543–557. Springer Verlag, October 2006.

[13] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs, 2002.

[14] Lars Grunske, Leif Geiger, and Michael Lawley. A graphical specification of model transformations with triple graph grammars, 2005.

[15] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko 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] Frederic Jouault and Ivan Kurtev. Transforming models with ATL. In *Proceedings of Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2006.

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

[18] László Lengyel, Tihamér Levendovszky, Gerely Mezei, and Hassan Charaf. Model transformation with a visual control flow language. *International Journal of Computer Science (IJCS)*, 1(1):45–53, 2006.

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

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

[21] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[22] Lei Sheng, Z. Meral Ozsoyoglu, and Gultekin Ozsoyoglu. A graph query language and its query processing. In *ICDE*, pages 572–581, 1999.

[23] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[24] W3C XML Schema. http://www.w3c.org/XML/Schema/.

$$
\begin{array}{rcl}
\mathit{Query} & ::= & \mathsf{select}\ \mathit{Template}\ \mathsf{where}\ BC_1, \ldots, BC_n \\
\mathit{Template} & ::= & \{\mathit{TE}_1 : \mathit{Template}_1, \ldots, \mathit{TE}_n : \mathit{Template}_n\} \\
 & \mid & \mathit{Var} \\
 & \mid & (\mathit{Query}) \\
 & \mid & \mathit{Template}_1 \cup \mathit{Template}_2 \\
 & \mid & \mathit{FName}(\mathit{Template}) \\
 & \mid & \mathsf{let\,sfun}\ \mathit{FName}_1(\{\mathit{PE}_{1_1} : \mathit{Pat}_{1_1}\}) = \mathit{Template}_{1_1} \\
 & & \qquad \mid\ \mathit{FName}_1(\{\mathit{PE}_{1_2} : \mathit{Pat}_{1_2}\}) = \mathit{Template}_{1_2} \\
 & & \qquad\qquad , \ldots , \\
 & & \qquad \mid\ \mathit{FName}_1(\{\mathit{PE}_{1_{m_1}} : \mathit{Pat}_{1_{m_1}}\}) = \mathit{Template}_{1_{m_1}} \\
 & & \quad \mathsf{sfun}\ \mathit{FName}_n(\{\mathit{PE}_{n_{m_n}} : \mathit{Pat}_{n_{m_n}}\}) = \mathit{Template}_{n_{m_n}} \\
 & & \mathsf{in}\ \mathit{Template} \\
 & \mid & (\mathit{Template}_1, \ldots, \mathit{Template}_n) \\
 & \mid & \mathsf{filter}(\mathit{BoolCond}, \mathit{Template}) \\
 & \mid & \mathsf{letval}\ \mathit{Var} := \mathit{Template}_1\ \mathsf{in}\ \mathit{Template}_2 \\
\mathit{TE} & ::= & \mathit{ValExp} \\
\mathit{ValExp} & ::= & \mathit{Var} \\
 & \mid & \mathit{Const} \\
 & \mid & \mathsf{not}\ \mathit{ValExp} \\
 & \mid & \mathsf{isEmpty}\ \mathit{ValExp} \\
 & \mid & \mathit{ValExp}_1\ \mathsf{and}\ \mathit{ValExp}_2 \\
 & \mid & \mathit{ValExp}_1\ \mathsf{or}\ \mathit{ValExp}_2 \\
 & \mid & \mathit{ValExp}_1 = \mathit{ValExp}_2 \\
 & \mid & \mathit{ValExp}_1 < \mathit{ValExp}_2 \\
 & \mid & \mathit{ValExp}_1 > \mathit{ValExp}_2 \\
BC & ::= & \mathit{BindCond} \\
 & \mid & \mathit{BoolCond} \\
\mathit{BindCond} & ::= & \mathit{Pat}\ \mathsf{in}\ \mathit{Template} \\
\mathit{BoolCond} & ::= & \mathit{ValExp} \\
\mathit{Pat} & ::= & \{\mathit{PE}_1 : \mathit{Pat}_1, \ldots, \mathit{PE}_n : \mathit{Pat}_n\} \\
 & \mid & \mathit{Var} \\
 & \mid & \mathit{Const} \\
\mathit{PE} & ::= & \mathit{Var} \\
 & \mid & \mathit{Const} \\
 & \mid & \mathit{RPP} \\
\mathit{RPP} & ::= & \mathit{Label} \\
 & \mid & \_ \\
 & \mid & (\mathit{RPP} \,.\, \mathit{RPP}) \\
 & \mid & (\mathit{RPP} \mid \mathit{RPP}) \\
 & \mid & \mathit{RPP}? \\
 & \mid & \mathit{RPP} * \\
\mathit{Const} & ::= & \mathit{Bool} \\
 & \mid & \mathit{String} \\
 & \mid & \mathit{Int}
\end{array}
$$

Figure 13: Abstract Syntax of UnQL

$$
\begin{array}{rcll}
E & ::= & \{\} & \text{(* empty tree *)} \\
 & | & \{L : E\} & \text{(* singleton tree *)} \\
 & | & \{l_1 : d_1, \ldots, l_n : d_n\} & \text{(* syn. sugar of } \{l_1 : d_1\} \cup \ldots \cup \{l_n : d_n\} \text{ *)} \\
 & | & E \cup E & \text{(* union of two trees *)} \\
 & | & \&x := E & \text{(* label the root node with input marker } x \text{ *)} \\
 & | & \&y & \text{(* data graph with output marker } y \text{ *)} \\
 & | & () & \text{(* empty data graph *)} \\
 & | & E \oplus E & \text{(* disjoint union *)} \\
 & | & (d_1, \ldots, d_n) & \text{(* synt. sugar of } d_1 \oplus \ldots \oplus d_n \text{ *)} \\
 & | & E \mathbin{@} E & \text{(* append of two data graphs *)} \\
 & | & \mathit{cycle}(E) & \text{(* data graph with cycles *)} \\
 & | & \mathit{Var} & \text{(* variable reference *)} \\
 & | & \mathsf{if}\ B\ \mathsf{then}\ E\ \mathsf{else}\ E & \text{(* conditional *)} \\
 & | & \mathit{rec}(\lambda(\mathit{LabelVar}, \mathit{Var}).E)(E) & \text{(* structural recursion *)} \\
 & | & \mathsf{let}\ \mathit{Var} = E\ \mathsf{in}\ E & \text{(* variable binding *)} \\
 & | & \mathsf{llet}\ \mathit{LabelVar} = L\ \mathsf{in}\ E & \text{(* label variable binding *)} \\
L & ::= & \mathit{LabelVar} & \text{(* label variable reference *)} \\
 & | & a & \text{(* label (} a \in \mathit{Label} \text{) *)} \\
 & | & L + L \mid L - L \mid L * L \mid L \mathbin{/} L & \text{(* arithmetic operation *)} \\
 & | & L\,\hat{}\,L & \text{(* concatenation *)} \\
B & ::= & \mathit{isempty}(E) & \text{(* true if value of E is empty *)} \\
 & | & L = L \mid L < L \mid L > L & \text{(* comparison *)} \\
 & | & \mathsf{true} \mid \mathsf{false} & \text{(* boolean literal *)} \\
 & | & \mathsf{not}\ L \mid L\ \mathsf{and}\ L \mid L\ \mathsf{or}\ L & \text{(* logical expression *)}
\end{array}
$$

Figure 14: Abstract Syntax and brief semantics of UnCAL

$$\frac{t_1\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_1\text{:: }\mathit{Template} \quad \ldots \quad t_n\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_n\text{:: }\mathit{Template}}{\{te_1 : t_1, \ldots, te_n : t_n\}\text{:: }\mathit{Template} \xrightarrow{apstp2} \{te_1 : t'_1, \ldots, te_n : t'_n\}\text{:: }\mathit{Template}}$$

$$\frac{v\text{:: }\mathit{Var}}{v <: \mathit{Template} \xrightarrow{apstp2} v <: \mathit{Template}} \qquad \frac{q\text{:: }\mathit{Query} \xrightarrow{step2} q'\text{:: }\mathit{Query}}{q <: \mathit{Template} \xrightarrow{apstp2} q' <: \mathit{Template}}$$

$$\frac{t_1\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_1\text{:: }\mathit{Template} \quad t_2\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_2\text{:: }\mathit{Template}}{t_1 \cup t_2\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_1 \cup t'_2\text{:: }\mathit{Template}}$$

$$\frac{t\text{:: }\mathit{Template} \xrightarrow{apstp2} t'\text{:: }\mathit{Template}}{f(t)\text{:: }\mathit{Template} \xrightarrow{apstp2} f(t')\text{:: }\mathit{Template}}$$

$$\frac{t_1\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_1\text{:: }\mathit{Template} \quad \ldots \quad t_n\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_n\text{:: }\mathit{Template} \qquad t\text{:: }\mathit{Template} \xrightarrow{apstp2} t'\text{:: }\mathit{Template}}{\begin{array}{c}\mathsf{let\,sfun}\, f_1(\{pe_1 : pat_1\}) = t_1, \ldots, \mathsf{sfun}\, f_n(\{pe_n : pat_n\}) = t_n \,\mathsf{in}\, t\text{:: }\mathit{Template} \xrightarrow{apstp2} \\ \mathsf{let\,sfun}\, f_1(\{pe_1 : pat_1\}) = t'_1, \ldots, \mathsf{sfun}\, f_n(\{pe_n : pat_n\}) = t'_n \,\mathsf{in}\, t'\text{:: }\mathit{Template}\end{array}}$$

$$\frac{t_1\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_1\text{:: }\mathit{Template} \quad t_n\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_n\text{:: }\mathit{Template}}{(t_1, \ldots, t_n)\text{:: }\mathit{Template} \xrightarrow{apstp2} (t'_1, \ldots, t'_n)\text{:: }\mathit{Template}}$$

$$\frac{t\text{:: }\mathit{Template} \xrightarrow{apstp2} t'\text{:: }\mathit{Template}}{\mathsf{filter}(bc, t)\text{:: }\mathit{Template} \xrightarrow{apstp2} \mathsf{filter}(bc, t')\text{:: }\mathit{Template}}$$

$$\frac{t_1\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_1\text{:: }\mathit{Template} \quad t_2\text{:: }\mathit{Template} \xrightarrow{apstp2} t'_2\text{:: }\mathit{Template}}{\mathsf{letval}\, v := t_1 \,\mathsf{in}\, t_2\text{:: }\mathit{Template} \xrightarrow{apstp2} \mathsf{letval}\, v := t'_1 \,\mathsf{in}\, t'_2\text{:: }\mathit{Template}}$$

Figure 15: Inference rules for a judgement of the form $t \xrightarrow{apstp2} t'$ in the second step of desugaring.

$$\frac{v\text{:: } Var}{v <: Template \xrightarrow{apstp3} v <: Template} \qquad \frac{q\text{:: } Query \xrightarrow{step3} q'\text{:: } Query}{q <: Template \xrightarrow{apstp3} q' <: Template}$$

$$\frac{t_1\text{:: } Template \xrightarrow{apstp3} t_1'\text{:: } Template \qquad t_2\text{:: } Template \xrightarrow{apstp3} t_2'\text{:: } Template}{t_1 \cup t_2\text{:: } Template \xrightarrow{apstp3} t_1' \cup t_2'\text{:: } Template}$$

$$\frac{t\text{:: } Template \xrightarrow{apstp3} t'\text{:: } Template}{f(t)\text{:: } Template \xrightarrow{apstp3} f(t')\text{:: } Template}$$

$$\frac{t_1\text{:: } Template \xrightarrow{apstp3} t_1'\text{:: } Template \quad \ldots \quad t_n\text{:: } Template \xrightarrow{apstp3} t_n'\text{:: } Template \qquad t\text{:: } Template \xrightarrow{apstp3} t'\text{:: } Template}{\begin{array}{c}\mathsf{let\,sfun}\; f_1(\{pe_1 : pat_1\}) = t_1, \ldots, \mathsf{sfun}\; f_n(\{pe_n : pat_n\}) = t_n \;\mathsf{in}\; t\text{:: } Template \xrightarrow{apstp3} \\ \mathsf{let\,sfun}\; f_1(\{pe_1 : pat_1\}) = t_1', \ldots, \mathsf{sfun}\; f_n(\{pe_n : pat_n\}) = t_n' \;\mathsf{in}\; t'\text{:: } Template\end{array}}$$

$$\frac{t_1\text{:: } Template \xrightarrow{apstp3} t_1'\text{:: } Template \qquad t_n\text{:: } Template \xrightarrow{apstp3} t_n'\text{:: } Template}{(t_1, \ldots, t_n)\text{:: } Template \xrightarrow{apstp3} (t_1', \ldots, t_n')\text{:: } Template}$$

$$\frac{t\text{:: } Template \xrightarrow{apstp3} t'\text{:: } Template}{\mathsf{filter}(bc, t)\text{:: } Template \xrightarrow{apstp3} \mathsf{filter}(bc, t')\text{:: } Template}$$

$$\frac{t_1\text{:: } Template \xrightarrow{apstp3} t_1'\text{:: } Template \qquad t_2\text{:: } Template \xrightarrow{apstp3} t_2'\text{:: } Template}{\mathsf{letval}\; v := t_1 \;\mathsf{in}\; t_2\text{:: } Template \xrightarrow{apstp3} \mathsf{letval}\; v := t_1' \;\mathsf{in}\; t_2'\text{:: } Template}$$

Figure 16: Inference rules for a judgement of the form $t \xrightarrow{apstp3} t'$ in the third step of desugaring

$$i(g) \overset{\text{def}}{=} \{\&x | (\&x, v) \in g.I\} \quad o(g) \overset{\text{def}}{=} \{\&y | (v, \&y) \in g.I\}$$

$$\frac{v \in \hat{V}}{\{\} \Rightarrow (\{v\}, \phi, \{(\&, v)\}, \phi)}$$

$$\frac{e \Rightarrow g \;\; v \notin g.V \;\; g.I(\&) = r \;\; p \to l}{\{p : e\} \Rightarrow (g.V \cup \{v\}, g.E \cup \{(v, l, r)\}, g.I \setminus (\&, r) \cup \{(\&, v)\}, g.O)}$$

$$\frac{v \in \hat{V}}{\&x \Rightarrow (\{v\}, \phi, \{(\&, v)\}, \{(v, \&x)\})}$$

$$\frac{e \Rightarrow g}{\&x := e \Rightarrow (g.V, g.E, \{(\&x.\&m, v) | (\&m, v) \in g.I\}, g.O)}$$

$$\frac{\begin{array}{c} e_1 \Rightarrow g_1 \;\; e_2 \Rightarrow g_2 \;\; \mathcal{X}_1 = i(g_1) \;\; \mathcal{X}_2 = i(g_2) \;\; \mathcal{X}_1 = \mathcal{X}_2 \\ E_1 = \{(v, \epsilon, g_1.I(\&x)) | \&x \in \mathcal{X}_1, v \in \hat{V}, v \notin g_1.V, v \notin g_2.V\} \\ E_2 = \{(v, \epsilon, g_2.I(\&x)) | \&x \in \mathcal{X}_1, (v, a, u) \in E_1, g_1.I(\&x) = v\} \\ V = \{u | (u, e, v) \in E_1\} \\ I = \{(\&x, u) | (u, a, v) \in E_1, \&x \in \mathcal{X}_1, g_1.I(\&x) = v\} \end{array}}{e_1 \cup e_2 \Rightarrow (g_1.V \cup v_2.V \cup V, g_1.E \cup g_2.E \cup E_1 \cup E_2, I, g_1.O \cup g_2.O)}$$

$$() \Rightarrow (\phi, \phi, \phi, \phi) \quad \frac{e_1 \Rightarrow g_1 \;\; e_2 \Rightarrow g_2 \;\; \mathcal{X}_1 = i(g_1) \;\; \mathcal{X}_2 = i(g_2) \;\; \mathcal{X}_1 \cap \mathcal{X}_2 = \phi}{e_1 \oplus e_2 \Rightarrow (g_1.V \cup g_2.V, g_1.E \cup g_2.E, g_1.I \cup g_2.I, g_1.O \cup g_2.O)}$$

$$\frac{\begin{array}{c} e_1 \Rightarrow g_1 \;\; e_2 \Rightarrow g_2 \\ E = \{(u, \epsilon, v) | (u, \&y) \in g_1.O, (\&x, v) \in g_2.I, \&y = \&x\} \end{array}}{e_1 \mathbin{@} e_2 \Rightarrow (g_1.V \cup g_2.V \cup V, g_1.E \cup g_2.E \cup E, g_1.I, g_2.O)}$$

$$\frac{\begin{array}{c} e \Rightarrow g \;\; \mathcal{X} = i(g) \;\; \mathcal{Y} = o(g) \;\; \mathcal{Z} = \mathcal{X} \cap \mathcal{Y} \\ I = \{(\&x, u) | (\&x, u) \in g.I, \&x \in \mathcal{Z}\} \\ O = \{(v, \&y) | (v, \&y) \in g.O, \&y \in \mathcal{Z}\} \\ E = \{(u, \epsilon, v) | (u, \&y) \in O, (\&x, v) \in I, \&y = \&x\} \end{array}}{\mathit{cycle}(e) \Rightarrow (g.V, g.E \cup E, g.I \setminus I, g.O \setminus O)}$$

Figure 17: UnCAL Dynamic Semantics