Towards a Compositional Approach to Model Transformation for Software Development

Soichiro Hidaka
hidaka@nii.ac.jp

Zhenjiang Hu
hu@nii.ac.jp

Hiroyuki Kato
kato@nii.ac.jp

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Keisuke Nakano
ksk@cs.uec.ac.jp

The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585, Japan

1. INTRODUCTION

Model transformation plays an important role in model-driven software development [11] that aims to introduce significant efficiencies and rigor to the theory and practice of software development. Although models may have different notations and representations, 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 prototypes on graph-based model transformations, it remains 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 using the graph querying language UnQL. The main idea of UnQL is that graph queries are fully captured by structural recursion that is suitable for efficient composition. We show that this idea can also be applied to graph-based model transformations. We have implemented a prototype of the framework and tested it with several nontrivial examples. Our new framework supports systematic development of model transformation “in the large” with the advantage that it can automatically remove inefficiencies arising from their composition.

Categories and Subject Descriptors
D.2.9 [Software Engineering]: Management—software process models; D.3.2 [Programming Languages]: Language Classifications—applicative (functional) languages; E.1 [Data Structures]: Graphs and Networks

General Terms
Management, Design, Languages
We designed a graph transformation language, UnQL, model transformations using the UnQL graph querying language. Our main contributions are twofold. First, we propose a compositional framework for graph-based model transformations using the UnQL graph querying language. We designed a graph transformation language, UnQL+, which extends UnQL with three simple graph editing constructs to achieve efficiency and expressiveness. All model transformation described in UnQL+ can be mapped to structural recursions that are suitable for efficient composition.

Second, we implemented a prototype of the new framework and tested it with several nontrivial examples. Our new framework supports systematic development of model transformation in the large with the advantage that it can automatically remove inefficiencies arising from this composition. We demonstrate, using the nontrivial model transformation from classes to a relational database management system, that a large model transformation can be systematically developed by gluing simpler model transformations together. Since all model transformations are represented by structural recursions or their composition, our system automatically eliminates inefficiencies arising from composition by using fusion optimization. The experimental results show promising speedups. Additionally, our system can validate input and output models against given metamodels with an efficient algorithm.

The organization of this paper is as follows. In Section 2, we review the graph query language UnQL and its model of graphs. Section 3 describes how UnQL and its extension is useful for systematic development of model transformations in a compositional manner using a typical but nontrivial model transformation, Class2-RDBMS. In Section 4, we explain the architecture of our compositional framework and its implementation. We discuss related work in Section 5 and conclude the paper in Section 6.

2. GRAPH QUERYING LANGUAGE UNQL

In this section, we briefly review the graph querying language, UnQL [6]. Our compositional framework for model transformations is based on UnQL. This language has a convenient select-where style surface syntax, which is translated into a 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 its complexity in answering an UnQL query is PTIME. We present the basic concepts of UnQL starting with graph representation and bisimulation in UnQL.

2.1 Graph Representation

Graphs in UnQL are edge-labelled; that is, all information is stored as labels on edges rather than on nodes (the labels on nodes have no particular meaning). They are rooted and directed cyclic graphs whose orders between outgoing edges of a node are insignificant. Every node may be marked with an input or output marker, which is called an input or output node, respectively. Input markers are used to select entry points of the graph, whereas output markers are used to glue output nodes with input nodes of a graph.

Figure 1 shows an example of a graph that represents a class model, which will be shown in Figure 2. The numbers in circled nodes of the graph are just identifiers that are added for explanation. All information in the class model including object names, attribute names, and attribute values, appears as labels of edges of the graph. A shared object of the class model (e.g., the class object named “Address”) is represented by a shared node in the graph (e.g., node 34). Some edges may be labelled with ε, which works like an ε transition in automata theory in that it identifies its source with its destination. They are used in establishing connections between nodes.

Graph bisimulation defines value equalities between graph instances. Intuitively, when graphs 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 from x_1 to y_1, then there is a corresponding edge from x_2 to y_2. The UnQL data model extends graph bisimulation by (1) requiring equalities between labels, (2) allowing insertion of one or more consecutive ε edges between a normal edge and target node (y_1 or y_2 above), (3) requiring correspondence between input nodes in G_1 and G_2, and (4) requiring correspondence between output markers of corresponding nodes (output markers may be associated with a 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 is surprising that a graph transformation defined in UnQL preserves bisimilarity [6] even though evaluation orders and strategies generally introduce divergence in results. If two graphs G_1 and G_2 are bisimilar, f(G_1) and f(G_2) are bisimilar for any transformation f in UnQL.

2.2 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 [6, 15]. We simply give some examples in this paper.

select-where construct. The following query Q1 extracts all primitive data types from the database (denoted $db in the query) in Figure 1.

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

In where-clause, we can describe pattern matching as a condition which has the form of pattern in variable, where pattern is tree-structured as $label : pattern'$ with label and pattern' which match an edge from its root and a graph following the edge, respectively. Unlike the original UnQL [6], variables are represented by $-prefixed symbols. We can use also regular path expressions for pattern matching at the left-hand side of the colon in the where clause.

Structural recursion in UnQL. Structural recursion plays a very important role in UnQL. Not only can it be used to describe 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, which satisfies the following two equations, f {} = {} and f (t₁ ∪ t₂) = f(t₁) ∪ f(t₂) for any graphs t₁ and t₂, where {} stands for a graph consisting of an empty node. Additionally, it guarantees that any return value of functions should not be fed to another function. 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 a map (on lists) is applied recursively to tail lists, structural recursion is applied (vertically) to nodes, as well as (horizontally) to edges.

As a simple use of structural recursion, the following query Q2 replaces all labels name under primitiveDataType in Figure 1 with type$@name. Due to the two equations above, definitions for horizontal recursion are always omitted.
Figure 1: Class model represented by an edge-labelled graph

```
(* Q2 *)
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)
```

The function \( f_1 \) takes the input graph \( \$db \) and matches all edges from its root with either \{primitiveDataType:$T\} or \{\$L:$T\}. According to the definition of \( f_1 \), it processes deeper subgraphs recursively. Since the subgraphs do not include the matched label, the recursion always terminates.

2.3 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 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 [15].

Contrary to the appearance of tree constructor \{\} and \( \cup \), its semantics of unification is different from those of sets. In UnCAL, although value equality is explicitly defined, duplicate eliminations do not take place. The graph shown in the right figure is represented by the following UnCAL expression:

```
&x2@cycle((&x2 := {a:&x21}, &x21 := {c:&x21,b:&x20}, &x20 := {}))
```

where \( &x20, &x21, \) and \( &x2 \) correspond to the three nodes, 0, 1, and 2, respectively, and a, b, and c correspond to the three edges.

3. MODEL TRANSFORMATION IN UnQL+

This section explains one of our most important contributions. We show how to extend the graph querying language UnQL [6] to UnQL+ with three useful editing operations, and demonstrate how it can be used for systematic development of model transformations with the example of the transformation from a class model to a relational database management system model.

3.1 UnQL+

UnQL is suitable for graph querying but not for graph transformation. For example, an UnQL query

```
select $Class where
{_*.class: $Class} in $db,
{is_persistent: true} in $Class
```

only extracts all of the persistent classes in Figure 1. In graph transformation, we often want to delete a subgraph, extend a subgraph with some new information, or replace a subgraph with a new one. It is onerous to describe these kinds of graph transformations in UnQL because we need to preserve the context by copying some structures in the input. UnQL+ is an extension of UnQL with three editing constructs that can support direct specification of these graph transformations (model transformations).

The deletion construct, \texttt{delete} ... where ..., is introduced to describe deletion of part of the graph. Consider the class graph in Figure 1, and suppose we want to eliminate all the names of association. This can be described by

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

Therefore, we may consider the \texttt{delete} as the dual of the \texttt{select}.

The extension construct, \texttt{extend} ... where ..., is introduced to extend a graph with another graph. For example, we write the following transformation to add date information to each association.

```
extend $AG with {date:"2008/8/4"}
where (association.name: $AssocName) in $AG
```

where the subgraph matched with \texttt{$AssocName} will be deleted from its original graph. In contrast, the following transformation extracts the association names as a result.

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

```
delete $AssocName
where (association.name: $AssocName) in $db
```

Therefore, we may consider the \texttt{delete} as the dual of the \texttt{select}.

The extension construct, \texttt{extend} ... where ..., is introduced to extend a graph with another graph. For example, we write the following transformation to add date information to each association.

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

where the subgraph matched with \texttt{$AssocName} will be deleted from its original graph. In contrast, the following transformation extracts the association names as a result.

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

Therefore, we may consider the \texttt{delete} as the dual of the \texttt{select}.

The extension construct, \texttt{extend} ... where ..., is introduced to extend a graph with another graph. For example, we write the following transformation to add date information to each association.

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

where the subgraph matched with \texttt{$AssocName} will be deleted from its original graph. In contrast, the following transformation extracts the association names as a result.

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

Therefore, we may consider the \texttt{delete} as the dual of the \texttt{select}.

The extension construct, \texttt{extend} ... where ..., is introduced to extend a graph with another graph. For example, we write the following transformation to add date information to each association.

```
extend $AG with {date:"2008/8/4"}
where (association: $AG) in $db
```
The replacement construct, replace ... where ..., is introduced to replace a subgraph with a new subgraph. For example, the transformation of replacing the edge label dest by tgt can be specified as follows.

\[
\text{replace } G \text{ by } \{\text{tgt: } G_1\}
\]
where \{\text{association: } G\} in $db$, \{\text{dest: } G_1\} in $G$

It is worth noting that as will be seen in Section 4.2, these new editing constructs can be mapped to structural recursions, and thus, all the advantages of UnQL, including the compositional property, are preserved.

3.2 Example: Class2RDBMS

As a nontrivial 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 approaches to model transformation. We explain two models, Class and RDBMS, and the requirement for a model transformation from Class to RDBMS, before showing how it can be systematically developed in UnQL+.

**Class Model.** A class model consists of classes and directed associations. 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). Associations are used to associate two classes. Figure 2 shows a class model, which consists of three classes and two directed associations. This class model is represented by the graph in Figure 1, where all information is stored on edges instead of nodes.

**RDBMS Model.** 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 table’s 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 3 shows an RDBMS model that has two tables.

**Specification of Class Models to RDBMS Models.** We recap the informal specification [3] of Class2RDBMS, the model transformation from class models to RDBMS models. A persistent class is mapped to a table and all its attributes or associations to columns in this table. If the type of attribute is primary, a primary key from the table to the column is established. If the type of 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 being 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 that access non-persistent classes. This model transformation is not trivial. We show below how to systematically develop it in our compositional framework.

Class2RDBMS in UnQL+. The compositional framework of UnQL+ allows us to develop bigger model transformations by gluing together 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. The entire transformation of Class2RDBMS in UnQL+ is given in [15]. Let us explain how it is systematically developed.

Recall the specification of Class2RDBMS, where we want to create tables (independent tables or tables pointed by foreign keys) from a class diagram, where each table should have a name and a sequence of columns, some of which are pointed by primary or foreign keys. This leads to the following top-level transformation.

\[
\text{select } \{\text{table: } \{\text{name: } \{\text{($Name: } \{\})\} \cup \text{($MakePKeyCol} \cup \text{($MakeGenCol} \cup \text{($MakeFKeyCol,}
\text{table: } \text{($MakeFKeyTable})\}}\}
\]
where ...

For the input class model $db$, as a preprocessing step, we replace all primitive data types in $db$ with their names and get $db'$ (because only the type names are used in the tables.)

\[
\text{$db'$ in}
\text{(replace $PrimDT$ by $Name$ where } \{\text{*_.type: } \text{$PrimDT}\} \text{ in $db$,}
\text{\{primitiveDataType.name: } \text{$Name\} \text{ in $PrimDT}}\)
\]

Now, to create columns of a table, we need to gather all the information on classes that are directly or indirectly associated with the source persistent class. This means we need to create an intermediate model $\text{ChainDB}$, in which indirectly associated classes are directly associated.

$\text{ChainDB in}$

\[
\text{(select}
\text{...}
\text{where } \text{\{association:}
\]

Figure 2: Class model

Figure 3: RDBMS model
The validated graph is transformed by a given program described in the input metamodel described in Kernel MetaMetaModel (KM3) [2]. An input model represented by a graph is validated against a given output metamodel in KM3. We also show how our model compositions work efficiently. The interest readers can find and execute some transformation examples in our demonstration Web pages at http://research.nii.ac.jp/~hidaka/big/.

4. Metamodel and Validation

Our system validates both input and output models represented by graphs against given metamodels of them. We employ Kernel MetaMetaModel (KM3) to describe metamodels because it has been used as a metametamodel in actual software development and is more formally defined than other metametamodels. A KM3 metamodel prescribes which sets of nodes must be referred to by a node by a regular expression. See [2] for details on the specification of KM3.

Figure 5 shows an example of a KM3 metamodel for classes, each of which is an input for the model transformation introduced in Section 3.2. The metamodel consists of four classes, 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. For example, the Association class has two references src and dest which are Class.

We validate a graph by associating each edge in them with the name of a class or a feature in a given KM3 metamodel. The validation of a graph starts with its root node. Every edge from the root node is associated with a class name. For example, an edge to node 33 in Figure 1 is associated with a class Association. Next we associate edges that follow it with feature names of the class. For example, an edge to node 32 is associated with the reference dest. We visit all edges repeatedly in this way and record matching intermediate use.

We look for two associations in which one’s destination is the other’s source, and then add a link edge from the source to the indirect destination. 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 $\$\text{ChainDB}$, it is easy to "query" the graph to extract information from each top (source) class that is persistent for creating a table later.

$\{\text{name: } $\$\text{N1, } \text{src: } $\$\text{C11, } \text{dest: } $\$\text{C12}\}$ in $\$\text{db}'$.
$\{\text{association:}
\text{name: } $\$\text{N2, src: } $\$\text{C21, dest: } $\$\text{C22}\}$ in $\$\text{db}'$.
$\{\text{name: } $\$\text{N12, src: } $\$\text{Any}\}$ in $\$\text{C12}$,
$\{\text{name: } $\$\text{N21, src: } $\$\text{Any}\}$ in $\$\text{C21}$,
$\$\text{Name12} = $\$\text{Name21}\}$

We validate a graph by associating each edge in them with the name of a class or a feature in a given KM3 metamodel. The validation of a graph starts with its root node. Every edge from the root node is associated with a class name. For example, an edge to node 33 in Figure 1 is associated with a class Association. Next we associate edges that follow it with feature names of the class. For example, an edge to node 32 is associated with the reference dest. We visit all edges repeatedly in this way and record matching information. This process is completed in linear time with regard to the size of the graph.

4.2 Mapping to the Core Language

UnQL+ provides a friendly interface 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. UnQL+ is mapped to UnCAL in a similar way to

\begin{verbatim}
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 [1-*] : Attribute;
  }
  class Attribute extends NamedElt {
    attribute is_p persistent : Boolean;
    reference type : PrimitiveDataType;
  }
  class PrimitiveDataType extends NamedElt {
  }
}
\end{verbatim}
that found in [6] except for the editing constructs. We show how to eliminate editing constructs in UnQL+ to enable mapping from UnQL+ to UnCAL.

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

\[
\text{delete } \mathcal{G} \text{ where } \ldots \\
\Rightarrow \text{replace } \mathcal{G} \text{ by } \emptyset \text{ where } \ldots
\]

\[
\text{extend } \mathcal{G} \text{ with } \mathcal{G}_1 \text{ where } \ldots \\
\Rightarrow \text{replace } \mathcal{G} \text{ by } \mathcal{G} \cup \mathcal{G}_1 \text{ where } \ldots
\]

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 such as \(\mathcal{A}=5\), simple pattern-in boolean expressions \(pi\) such as \(\{\mathcal{P}:G\}\) in \(\text{template}\) with a pattern \(\mathcal{P}\) and variable \(\mathcal{G}\), or simple binding expressions \(bd\) such as \(\mathcal{G}\) in \(\text{template}\) with a variable \(\mathcal{G}\). Thus, the general form of an expression using replace is

\[
\text{replace } \mathcal{G}_1 \text{ by } \mathcal{G}_2 \text{ where } bc_1, \ldots, bc_{k-1}, \{\mathcal{P}:G_1\} \text{ in } bd_1, bd_{k+1}, \ldots, bd_n
\]

where \(bc_1, \ldots, bc_{k-1}\) are either relation expressions or pattern-in boolean expressions. Among others, two important rules are as follows to reduce the number of boolean conditions in the where clause. In the following rules, like [6], we use \(rest\) for a syntactic meta-variable which stands for the remaining clauses in the where component. Note that the first element in \(rest\) is restricted to \(pi\) or \(bd\) unless \(rest\) is empty. The first rule is to deal with the case where the first pattern-in boolean expression (in the where clause) does not match the graph to be replaced, i.e., \(\mathcal{G}_1 \neq \mathcal{G}_3\).

\[
\text{replace } \mathcal{G}_1 \text{ by } \mathcal{G}_2 \text{ where } (1:G_3) \text{ in } bd_1, r_1, \ldots, r_m, \text{ rest } \\
\Rightarrow \text{let fun } h_1((\mathcal{L}:G_3)) = \\
\begin{cases} 
\text{if } \mathcal{L}=l_1 \text{ and } r_1 \text{ and } \ldots \text{ and } r_m \text{ then } \\
(1:G_1 \text{ by } G_2 \text{ where rest}) & \text{else } (1:G_3) 
\end{cases}
\]

in \(h_1(\mathcal{G}_D)\)

The second rule, on the other hand, is to deal with the case where the first boolean condition matches the graph to be replaced.

\[
\text{replace } \mathcal{G}_1 \text{ by } \mathcal{G}_2 \text{ where } (1:G_1) \text{ in } bd_1, r_1, \ldots, r_m, \text{ rest } \\
\Rightarrow \text{let fun } h_1((\mathcal{L}:G_1)) = \\
\begin{cases} 
\text{if } \mathcal{L}=l_1 \text{ and } r_1 \text{ and } \ldots \text{ and } r_m \text{ then } \\
\text{letval } G_1 = (1:G_1) \text{ where rest in } \\
\text{letval } G_2' = \text{select } G_2 \text{ where rest in } \\
\text{if isEmpty}(G_1') \text{ then } (1:G_1) \text{ else } (1:G_2') & \text{else } (1:G_1) 
\end{cases}
\]

in \(h_1(\mathcal{G}_D)\)

To see how these rules work, consider the following expression with replace, which is to replace the association name "phone" with "assoc_phone".

\[
\text{replace } \text{Name} \text{ by } \text{Name'} \text{ where } \{\text{association}:=\text{Assoc}\} \text{ in } \text{db}, \\
\{\text{name}:=\text{Name}\} \text{ in } \text{Assoc}, \\
\{\text{string}:=\text{Name}\} \text{ in } \text{Name}, \\
\{\text{N}:=\text{Name}\} \text{ in } \text{Na}, \\
\text{N} = "phone", \\
\text{Name'} \text{ in } \{\text{string}:=\text{"assoc_phone"} \text{N}\}
\]

With the two rules, we can successfully desugar it to the following, where replace is removed.

\[
\text{let fun } h_1((\mathcal{L}:\text{Assoc})) = \\
\text{if } \mathcal{L}=\text{association} \text{ then } \\
\text{let fun } h_2((\mathcal{L}:\text{Name})) = \\
\]

4.3 Interpretation of the Core Language

Buneman et al.'s UnQL paper [6] 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 \(rec(e_1)\) take place in a top-down fashion. Revisiting of nodes caused by cycles can be correctly handled by memoization. The former deals with possible cycles by applying \(e_1\) once for every edge in an input graph and connecting them together using Skolem functions on markers and nodes. Our system implemented in Objective Caml uses an algebraic data type for the Skolem function and a tree-structured set library for the first-order formula on nodes and edges. It is fairly straightforward and efficient in both recursive semantics and bulk semantics.

4.4 Model Composition

We identify two forms of model composition. The 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 \(T_2\): \(M' = (T_2 \circ T_1)(M)\). The second one is a pair of transformations, \(T_1\) and \(T_2\), that share an identical input model: \((M_1,M_2) = (T_1 \triangle T_2)(M)\), where \(\triangle\) is defined as \((T_1(M), T_2(M))\). In the first composition, an intermediate result can be eliminated by the fusion technique, while in the second composition, a duplicate traversal of the input model can be unified by the tupling technique. Our system provides automatic fusion for the first composition.

In our framework, consecutive model transformations are translated into a composition of structural recursions in UnCAL. Hence, we can directly apply fusion transformation for the UnCAL proposed in [6]. As a very simple case, consider the following scenario (borrowed from [6]): first apply \(Q2\) (in Section 2) to the model in Figure 1, and then retrieve all names by the following query \(Q3\).

\[
Q3 \ast \\
\begin{aligned}
&\text{select} \\
&\text{letrec } sf(\text{Name}:=\text{\$T}) = \text{Name}\text{\$T} \\
&\text{f} = \text{sf}(\text{Name}:=\text{\$T}) \\
&\text{g} = \text{sf}(\text{Name}:=\text{\$T}) \\
&\text{in } \text{f} \text{ in } \text{g} \\
\end{aligned}
\]

This compositional query would look like the following query \(Q4\), by which our desugaring module produces an UnCAL query \(Q5\). Our UnCAL rewriter translates it into \(Q6\) (with simple rewriting by hand for readability), where two \text{let} in \(Q5\) are fused into one.
select
letrec sfun f2 ((name:$T)) = (name:g2($T))
| f2 ((SL:$T)) = f2($T)
and sfun g2 ((SL:$T)) = (SL:g2($T))
in
letrec sfun f1 ((primitiveDataType:$T))
= (primitiveDataType:g1($T))
| f1 ((SL:$T)) = (SL:f1($T))
and sfun gl ((name:$T)) = (typeName:gl($T))
g1 ((SL:$T)) = (SL:gl($T))
in f2(f1($db))

(* Q5 *)
$$\{z1\} \subseteq \{z2\} \implies \{z1\} \cup \{z2\} \subseteq \{z1\} \cup \{z2\}$$

(* Q6 *)
$$\{z1\} \cup \{z2\} = \{z1\} \cup \{z2\} \iff \{z1\} = \{z2\}$$

The table below shows the efficiency gained from the above fusion. The experiment was conducted on a 1.5 GHz quad Xeon SMP machine running Linux kernel 2.4.20. An approximate three- to five-fold speed-up was confirmed.

<table>
<thead>
<tr>
<th>evaluation strategy of rec</th>
<th>before fusion</th>
<th>after fusion</th>
<th>speed-up ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursive</td>
<td>1.31 sec</td>
<td>0.25 sec</td>
<td>5.32</td>
</tr>
<tr>
<td>bulk</td>
<td>2.08 sec</td>
<td>0.73 sec</td>
<td>2.86</td>
</tr>
</tbody>
</table>

Our system employs the optimization rules presented in [6] to reduce the size of arguments of rec. In addition, we introduce the following simplification rules for other constructs to optimize graph transformation.

$$\forall x. \forall z : e \implies \forall x. \forall z : e$$
$$\forall x : (e_1 \oplus e_2) \implies (\forall x : e_1) \oplus (\forall x : e_2)$$

where we follow the notation of graph constructs in [6]. The operator \( \oplus \) constructs disjoint union of two graphs while \( \ominus \) denotes an empty graph, thus, the expressions like \(\{x_1 := g_1, x_2 := g_2, \ldots, x_n := g_n\}\), which we have already seen in this paper are the syntactic shorthands for \(\{x_1 := g_1\} \ominus (\{x_2 := g_2\} \ominus \ldots \ominus (\{x_n := g_n\})\). The last rule requires that the sets of input and output markers be inferred for a given UnCAL expression. Our system is capable of doing this estimation at compile time in a way similar to that in [5]. There may be other rules applicable. Exploring these rules will be part of our future work.

5. RELATED WORK

Our work is very closely related to research on model transformation based on graph transformation in the software engineering community, as well as to 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 that supports an algebraic approach to graph transformation. AGG supports typed (attributed) graph transformations including type inheritance and multiplicities. Rule application can contain a non-deterministic choice of rules that 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 aims 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 is 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 the main conceptual difference. Since their recursive control structures can be very complicated, it remains unclear how to efficiently compose them. Our approach puts reasonable restrictions on the recursive structure so that it is not only powerful enough to specify various model transformations but also suitable for efficient composition.

On the other hand, in the database community, a lot of work has been done on language design and implementation for efficient graph querying [13, 22, 6]. Different from querying trees, issues on representation and equivalence of graphs are subtle and important to define the correctness of graph querying (as well as graph transformation), and the use of bisimulation and structural recursion in [6] 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 design and implement a compositional framework for model transformations based on UnQL. Although UnQL is well known in the 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 that 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 updates on the target model can be reflected in the source model. This would connect the interesting idea of bidirectional computation in both the programming language and software engineering communities.

7. 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 Objective Caml.

Our 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, the Grant-in-Aid for Scientific Research, who kindly provided us with the SML source codes of an system in Objective Caml.

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

8. REFERENCES


