Toward bidirectionalization of ATL with GRoundTram

Isao Sasano[1], Zhenjiang Hu[2], Soichiro Hidaka[2], Kazuhiro Inaba[2], Hiroyuki Kato[2], and Keisuke Nakano[3]

[1] Shibaura Institute of Technology, Japan,
sasano@sic.shibaura-it.ac.jp
[2] National Institute of Informatics, Japan,
{hu, hidaka, kinaba, kato}@nii.ac.jp
[3] The University of Electro-Communications, Japan,
ksk@cs.uec.ac.jp

Abstract. ATL is a language for describing model transformations currently in uni-direction. In our previous work we have shown that transformations of graph structures given in some form can be bidirectionalized and have implemented a system called GRoundTram system for bidirectional graph transformations. We say a transformation t is bidirectionalized when we obtain a backward transformation t' so that the pair (t, t') of transformations satisfies certain well-behavedness properties. Bidirectional model transformation is used to reflect the changes in the target model back to the source model, and vice versa. In this paper, as a first step toward realizing practical bidirectional model transformations, we present bidirectionalization of core part of the ATL by encoding it in the UnQL language, which is used as a transformation language in the GRoundTram system. We give the algorithm for the encoding, based on which we have implemented the system for bidirectionalizing the core ATL in OCaml language.

1 Introduction

ATL [15, 14, 1] is a widely used language for describing model transformation, and its environment is provided as an easy-to-use plug-in of the Eclipse framework. An ATL program consists of rules which specify how to transform components of a source model into components of a target model. A rule can describe computations like integer arithmetic or string manipulations and check various conditions in OCL expressions, and can change the structure between components when producing the target model. This rule-based mechanism enables us to declaratively describe a wide variety of model transformations.

Despite its practical and wide uses, ATL lacks the important *bidirectional* feature in that it can only describe unidirectional mapping from the source model to the target model. Bidirectionality, as being seen in many other model transformation languages such as QVT and TGG [17], plays an important role in model synchronization, consistency maintenance, and reverse engineering [9]. One attempt to bidirectionalize ATL was made in the level of byte code of the virtual machine of ATL system [20], but it imposes many restrictions on the ATL byte code, and this restrictions on the lower byte code is somehow difficult to be understood and controlled by the users who write ATL programs.

As an alternative, by contrast to the low level attempt, we shall take an *incremental* approach to bidirectionalizing ATL in a high level. Our idea is to show that a small core of ATL can be bidirectionalized, while making use of the fact that this core part can coexist well with other parts that cannot be bidirectionalized. This coexistence is possible because of modular execution of ATL programs; each rule specifies direct mapping from some elements in the input model to those in the output model. This core part could be extended and generalized in the future to deal with more of bidirectional computation in an ATL program. Now the problem is how to bidirectionalize ATL transformation rule, the basic unit of model transformation. Can we use the existing bidirectional languages to interpret ATL?

Bidirectional transformations, originated from the view update problem in the databases [4], have received much attention from the programming language community, and several *well-behaved* bidirectional transformation languages have been proposed [7, 16, 13, 18, 5, 19], where the round-trip properties like put-get or get-put, which characterizes the bidirectional transformations, are guaranteed to be satisfied. However, most of these well-behaved bidirectional transformation languages manipulate trees or strings, which are not suitable for bidirectionalizing ATL, because models are essentially graphs. Recently, in our previous work [11] it is shown that the UnQL language [8], a well-known graph query language, can be used as a well-behaved bidirectional graph transformation language. In addition, a bidirectional graph transformation system called GRoundTram (Graph Roundtrip Transformation) [2] has been developed, where we can write bidirectional graph transformations in the UnQL language.

In this paper, as a first step toward realizing practical and well-behaved bidirectional model transformations, we present bidirectionalization of core part of the ATL by encoding it in the UnQL language. We give the algorithm for the encoding, based on which we have implemented the system for bidirectionalizing the core ATL in OCaml language. With representing the source model in a graph data structure, we can bidirectionally apply the encoded transformation in GRoundTram system. Throughout the paper we use a simple example to illustrate our algorithm.

The organization of this paper is as follows. Section 2 shows the overview of ATL, GRoundTram system, and UnQL language. Section 3 shows the encoding and decoding process between models and UnQL graph structures. Section 4 presents the algorithm for encoding ATL rules. Section 5 concludes the paper.

2 Preliminaries

Here we show the overview of the ATL, UnQL, and the GRoundTram system.

2.1 ATL

In this paper we use the following subset of the ATL language to show the essential part of the bidirectionalization. The subset does not cover imperative features of the ATL. We also exclude the most of the OCL expressions to avoid cluttering the essential part of the bidirectionalization. Although the subset may not have the same description power as the full set, it is enough for the purpose of showing the idea of our approach to bidirectionalization of model transformations.

$$\begin{aligned}
ATL &= \textbf{module}\ id;\ \textbf{create}\ id : id;\ \textbf{from}\ id : id;\ rule^{+} \\
rule &= \textbf{rule}\ id\ \ \textbf{from}\ inPat\ \textbf{to}\ outPat^{+} \\
inPat &= id : oclType \\
outPat &= id : oclType\ binding^{*} \\
binding &= id \leftarrow oclExp \\
oclExp &= id \\
&\mid\ id.id \\
&\mid\ string \\
&\mid\ oclExp + oclExp
\end{aligned}$$

ATL consists of rules, each of which specifies a transformation that is applied to some components in the source model. A rule is described by the $rule$ construct in the above syntax and the $inPat$ construct id: $oclType$ in each rule specifies to which component the rule is applied. For the details of ATL, refer to the documents in the ATL web page [1].

Here we illustrate the intuitive meaning of the ATL language by using an example in Fig. 1. It consists of two rules, `Class2Table` and `Attribute2Column`. The example is made by simplifying the class2RDBMS example provided as a non-trivial benchmark application for testing the power of model transformation languages in the announcement of the workshop MTiP 2005 [6].

In ATL we need to specify the metamodels for source models and target models. Let the metamodel of source models be the one in Fig. 2 and the metamodel of target models be the one in Fig. 3. In the ATL environment the metamodels are described by the ECore diagram or KM3 (kernel meta meta model) [3]. The metamodels in Fig. 2 and 3 are given in KM3.

Let us use the model in Fig. 4 as an example of the source model. This model specifies that a Person class has two attributes (fields), name and address. This is transformed into the target model in Fig. 5. In the next section we give the core idea of bidirectionalizing ATL by using the example given above.

2.2 UnQL

Let us briefly review the graph querying language, UnQL [8]. The language resembles the SQL for relational databases in its **select-where** syntax, but is designed for manipulating graphs. In particular, it has a construct called structural recursion to traverse over the given input graphs. We omit the formal definition of the language, which can be found in [8]. We here informally present the basic concepts of UnQL starting with its graph data model.

Graph Data Model Graphs in UnQL are rooted and directed cyclic graphs with no order between outgoing edges. They are edge-labeled in the sense that all information is stored as labels on edges and the labels on nodes serve as a unique identifier and have no particular meaning. The edge-labels can be either integers (e.g., 123, or 42),

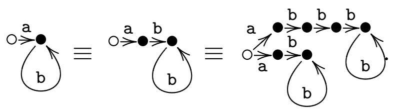
```
rule Class2Table {
    from
        s : ClassDiagram!Class
    to
        t : Relational!Table (
            name <- s.name,
            col <- s.attr
        )
}

rule Attribute2Column {
    from
        s : ClassDiagram!Attribute
    to
        t : Relational!Column (
            name <- s.name
        )
}
```

Fig. 1. A model transformation in ATL

strings (like `"hello"`) for representing data-values, or bare-symbols (`name`, or `attr`) for representing structures of graphs.

Two graphs in UnQL are considered to be equal if they are *bisimilar*. Intuitive understanding of bisimulation is that unfolding of cycles and duplication of equivalent subgraphs are not distinguished, and unreachable part from the root is ignored. Here is an examples of graphs that are bisimilar:



Every construct in UnQL respects bisimulation, i.e., if two bisimilar graphs are fed as inputs to a query, the results are always bisimilar again. This notion of equivalence plays an important role for query optimization [8] or bidirectionalization [11]. When the user does want to distinguish two bisimilar graphs as a different object, the user can add special *tag* edges labeled with unique identifiers to them, which breaks the bisimilarity and are dealt with separately.

Query Syntax The syntax of UnQL query is summarized in Figure 6. The $\mathbf{select}\ T$ $\mathbf{where}\ B, \ldots, B$ form is the entry point of the query. It selects the subgraphs satisfying the $\mathbf{where}\, B$ part and bind them to variables, and construct a result according to the template expression T. In T, the expression $\{L_1 : T_1, \ldots, L_n : T_n\}$ creates a new node having n outgoing edges labeled L_i and pointing to another node T_i. The union

```
package Class {
    abstract class NamedElt {
        attribute name : String;
    }

    class Class extends NamedElt {
        reference attr[*] : Attribute oppositeOf owner;
    }

    class Attribute extends NamedElt {
        reference type : Class;
        reference owner : Class oppositeOf attr;
    }
}

package PrimitiveTypes {
    datatype Boolean;
    datatype Integer;
    datatype String;
}
```

Fig. 2. The meta model in KM3 for the source models

$G_1 \cup G_2$ constructs a graph with a root sharing the roots of G_1 and G_2. For example, $\{L_1 : g_1\} \cup \{L_2 : g_2\}$ equals to $\{L_1 : g_1, L_2 : g_2\}$. In the template, by using the keyword $\mathbf{sfun}$, the programmer can also define a powerful *structural recursion*, which will be explained later. In the binding condition B part, comparison of label values and regular-expression based traversal on paths on graphs can be used.

Structural Recursion A function f on graphs is called a structural recursion if it is defined by the following equations

$$\begin{aligned} f(\{\}) &= \{\} \\ f(\{\$l : \$g\}) &= e \\ f(\$g_1 \cup \$g_2) &= f(\$g_1) \cup f(\$g_2), \end{aligned}$$

where the expression e may contain references to variables $\$l$ and $\$g$, and recursive calls of the form $f(\$g)$, but no application of f to other graphs than $\$g$. Since the first and the third equations are common in all structural recursions, we omit them in UnQL. For the second line, since it is customary to dispatch the graph operation by labels, pattern-matching can be used instead of using long if-then-else sequence. For instance,

```
package Relational {
    abstract class Named {
        attribute name : String;
    }

    class Table extends Named {
        reference col[*] : Column oppositeOf owner;
    }

    class Column extends Named {
        reference owner : Table oppositeOf col;
    }
}

package PrimitiveTypes {
    datatype Boolean;
    datatype Integer;
    datatype String;
}
```

Fig. 3. The meta model in KM3 for the target models

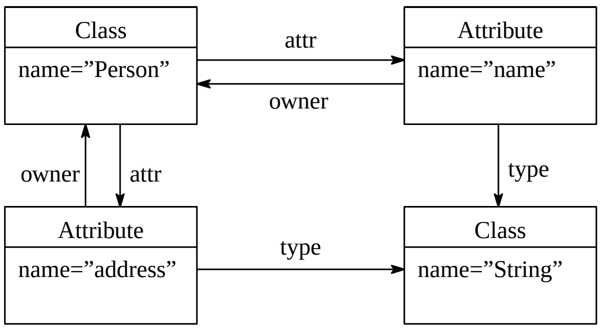


Fig. 4. A source model

we can write

$$\begin{array}{lll}\textbf{sfun}\ f\ (\{\texttt{class} : \$g\}) & = e_1 \\ \quad|\quad f\ (\{\texttt{interface} : \$g\}) & = e_2 \\ \quad|\quad f\ (\{\texttt{int} : \$g\}) & = e_3 \\ \quad\vdots & \end{array}$$

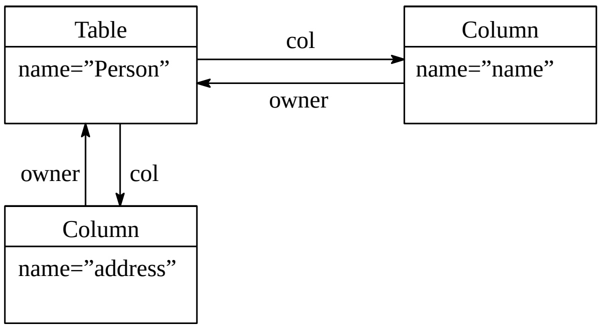


Fig. 5. The target model obtained by applying the rules to the source model

$$
\begin{array}{lll}
\text{(query)} & Q & ::= \ \textbf{select}\ T\ \textbf{where}\ B, \ldots, B \\
\text{(template)} & T & ::= \ Q \ \mid\ \{L : T, \ldots, L : T\} \ \mid\ T \cup T \ \mid\ \$G \ \mid\ f(\$G) \\
 & & \quad \mid\ \textbf{if}\ BC\ \textbf{then}\ T\ \textbf{else}\ T \\
 & & \quad \mid\ \textbf{let sfun}\ f\ \{Lp : Gp\} \ = T \\
 & & \qquad\qquad \mid\ f\ \{Lp : Gp\} \ = T \\
 & & \qquad\qquad\ \ldots \\
 & & \qquad\quad \textbf{sfun}\ f'\ \{Lp : Gp\} = T \\
 & & \qquad\qquad \mid\ f'\ \{Lp : Gp\} = T \\
 & & \qquad\qquad\ \ldots \\
 & & \qquad\ \ldots \\
 & & \quad \textbf{in}\ T \\
\text{(binding)} & B & ::= \ Gp\ \textbf{in}\ \$G \ \mid\ BC \\
\text{(condition)} & BC & ::= \ \mathsf{not}\, BC \ \mid\ BC\, \mathsf{and}\, BC \ \mid\ BC\, \mathsf{or}\, BC \\
 & & \quad \mid\ \mathsf{isEmpty}(\$G) \ \mid\ L = L \ \mid\ L \neq L \ \mid\ L < L \ \mid\ L \leq L \\
\text{(label)} & L & ::= \ \$l \ \mid\ \mathsf{a} \\
\text{(label pattern)} & Lp & ::= \ \$l \ \mid\ Rp \\
\text{(graph pattern)} & Gp & ::= \ \$G \ \mid\ \{Lp : Gp, \ldots, Lp : Gp\} \\
\text{(regular path pattern)} & Rp & ::= \ \mathsf{a} \ \mid\ \_ \ \mid\ Rp.Rp \ \mid\ (Rp|Rp) \ \mid\ Rp? \ \mid\ Rp*
\end{array}
$$

Fig. 6. Syntax of UnQL

instead of writing

$$
\begin{array}{ll}
\textbf{sfun}\ f\ (\{\$l : \$g\}) = & \textbf{if}\ \$l = \texttt{class}\ \textbf{then}\ e_1 \\
 & \textbf{else if}\ \$l = \texttt{interface}\ \textbf{then}\ e_2 \\
 & \textbf{else if}\ \$l = \texttt{int}\ \textbf{then}\ e_3 \\
 & \textbf{else}\ \ldots
\end{array}
$$

The following example shows a simple usage of structural recursion.

$$
\begin{array}{lll}
\textbf{sfun} & a2d\_xc(\{\mathsf{a} : \$g\}) & = \{\mathsf{d} : a2d\_xc(\$g)\} \\
\mid & a2d\_xc(\{\mathsf{c} : \$g\}) & = a2d\_xc(\$g) \\
\mid & a2d\_xc(\{\$l : \$g\}) & = \{\$l : a2d\_xc(\$g)\}
\end{array}
$$

It replaces all edges in the graph labeled a by d, contracts the edges labeled c, and keeps the other edges unchanged.

Despite its simplicity, structural recursion (and hence UnQL) is powerful enough to describe interesting nontrivial model transformations [12].

In this paper, for simplicity, we often write

$$\textbf{sfun}\ f\ (\{\mathrm{a} : \{\mathrm{b} : \$v\}\}) = \ldots$$

to denote

$$\textbf{sfun}\ f'\ (\{\mathrm{a} : \$g\}) = \textbf{let}\ \textbf{sfun}\ h\ (\{\mathrm{b} : \$v\}) = \ldots \textbf{in}\ h(\$g).$$

Bidirectional Semantics Usually, a query is run in one direction. That is, given an input environment (a mapping from variables to graphs) ρ, a query Q is evaluated and generated the result graph which we denote $\mathcal{F}[\![Q]\!]^{\rho}$. Now, let $G = \mathcal{F}[\![Q]\!]^{\rho}$ and consider the user has edited the result graph into G'. For example, he can add a new subgraph, or modify some label, or delete several edges, and so on. In our previous work [11], we have given a *backward semantics* that properly reflects back the editing to the original inputs. More formally speaking, given the modified result graph G' and the original input environments ρ, the modified environment $\rho' = \mathcal{B}[\![Q]\!]^{\rho}_{G'}$ can be computed.

By ”properly reflecting back”, we mean the following two properties to hold.

$$\begin{array}{lll} \mathcal{F}[\![Q]\!]^{\rho} = G \quad \text{implies} \quad \mathcal{B}[\![Q]\!]^{\rho}_{G} = \rho & & (\textsc{GetPut}) \\ \mathcal{B}[\![Q]\!]^{\rho}_{G'} = \rho' \quad \text{implies} \quad \mathcal{B}[\![Q]\!]^{\rho}_{\mathcal{F}[\![Q]\!]^{\rho'}} = \rho' & & (\textsc{WPutGet}) \end{array}$$

The (GetPut) property says that if no change is made on the output G, then there should occur no change on the input environment. The (WPutGet) property is an unrestricted version of (PutGet) property appeared in [10], which requires $G' \in \mathrm{Range}(\mathcal{F}[\![Q]\!])$ and $\mathcal{B}[\![Q]\!]^{\rho}_{G'} = \rho'$ to imply $\mathcal{F}[\![Q]\!]^{\rho'} = G'$. The (PutGet) property states that if a result graph is modified to G' which is in the range of the forward evaluation, then this modification can be reflected to the source such that a forward evaluation will produce the same result G'. In contrast to it, the (WPutGet) property allows the modified result and the result obtained by backward evaluation followed by forward evaluation to differ, but require both to have the same effect on the original source if backward evaluation is applied.

2.3 GRoundTram system

In our previous work we have developed a system called GRoundTram system, which enables us to describe bidirectional transformations on graph data structures in the UnQL language. Note that we describe both the transformations and the graph data structures in UnQL language. We show the overview of the system in Fig. 7. Our system implements the bidirectional evaluator between source graphs and target graphs.

Figure 8 shows a screenshot of our system. The user loads a source graph (displayed in the left pane) and a transformation written in UnQL. User can optionally specify the source metamodel and target metamodel in KM3. Once they are loaded, forward

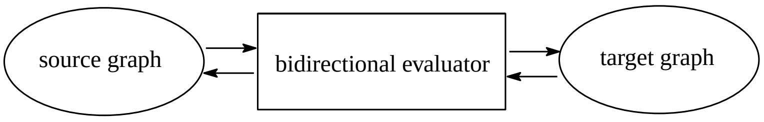


Fig. 7. GRoundTram system

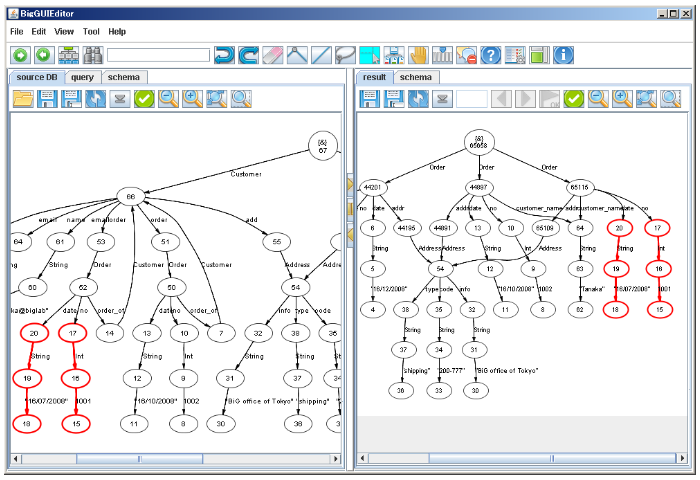


Fig. 8. Screenshot of GRoundTram System

transformation can be conducted by pushing "forward" button (right arrow icon). The target graph appears on the right pane. User can graphically edit the target graph and apply backward transformation by pushing "backward" button (left arrow icon). Source graph can be edited as well, of course. Metamodel conformance of the source and the target can be checked any time by pushing check icon on both panes. The transformation itself can also be *statically* checked: given source/target metamodel and transformation, the system checks whether the target graph *always* conforms to given target metamodel. If not, a counterexample graph is displayed.

Figure 8 also demonstrates the traceability between source and target (red part). If the user selects subgraphs on either pane, then corresponding subgraphs on other pane are also highlighted. This helps the user to predict modification on which part in the target will affect which part on the source, and vice versa.

3 Encoding and decoding between models and graph structures

In order to use the GRoundTram system we need to encode the models in UnQL language and decode the results back to models. Instead of giving algorithms for them, here we illustrate the encoding process using the model in Fig. 4.

Corresponding to the encoding of the rules in Section 4, we encode the models by using the constant pattern ClassName. For example, the component of the Class with name field "Person" is encoded into the following UnQL graph structure.

$$
\begin{aligned}
g_1 = \{\text{ClassName} : \{\text{Class} : \{&\text{name} : \{''\text{Person}'' : \{\}\},\\
&\text{attr} : g_2,\\
&\text{attr} : g_3\}\}\}
\end{aligned}
$$

The graphs g_2 and g_3 are obtained encoding of the components with Attribute class as follows.

$$
\begin{aligned}
g_2 = \{\text{ClassName} : \{\text{Attribute} : \{&\text{name} : \{''\text{name}'' : \{\}\},\\
&\text{owner} : g_1,\\
&\text{type} : g_4\}\}\}\\
g_3 = \{\text{ClassName} : \{\text{Attribute} : \{&\text{name} : \{''\text{address}'' : \{\}\},\\
&\text{owner} : g_1,\\
&\text{type} : g_4\}\}\}
\end{aligned}
$$

The graph g_4, encoded as follows, is for the remaining component.

$$
g_4 = \{\text{ClassName} : \{\text{Class} : \{\text{name} : \{''\text{String}'' : \{\}\}\}\}\}
$$

The above representation is informal one for giving intuitive understanding. Formally, when encoding models with cycles as in Fig. 4, we use the cycle construct and markers in UnCAL language [8], which we omit for simplifying the presentation.

One thing we should note is we have to be able to get the original model representations from the graph structures. The overall figure of our approach is summarized in Fig. 9. The decoding process, which we omit, is performed naturally in the reverse way of the encoding process.

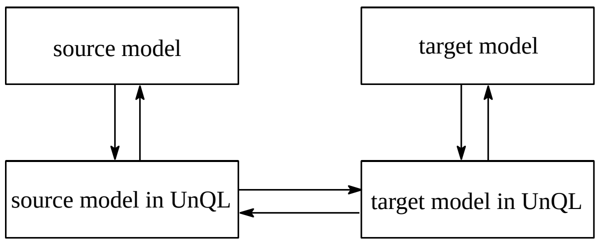


Fig. 9. Overview of our system

4 Encoding ATL rules in UnQL

In this section we present the algorithm to encode a given ATL program into an UnQL expression. An ATL program consists of rules, as we have shown in Section 2.1. Our strategy for encoding is to transform each ATL rule into a function in the **sfun** construct in UnQL, by using the identifiers in the ATL rules when making an UnQL function.

We design the algorithm along the structure of the ATL language in the following. The top level transformation function is $atl2unql$.

$$\begin{aligned}&\mathit{atl2unql}\ (\mathbf{rule}\ r\ \mathbf{from}\ \mathit{inPat}\ \mathbf{to}\ \mathit{outPatSeq}) =\\ &\quad \mathbf{sfun}\ r\ (\mathit{inPat2arg}\ \mathit{inPat}) = \mathit{outPatSeq2unql}\ \mathit{outPatSeq}\end{aligned}$$

This function takes a rule in ATL and produces a function in UnQL language. This function is applied to rules in the ATL program, producing one function for each rule. We use the name of the rule as the name of the function. The $inPat$ is the pattern specifying to which components the rule is applied. We transform this part by applying $inPat2arg$, which produces a pattern that appears as the argument of the UnQL function. As we will mention in Section 3, each component in the model is encoded in UnQL graph structure so that the function can find the encoded components by pattern matching. For this purpose we encode each component using the constant pattern ClassName. So we define $inPat2arg$ to produce the pattern including the constant ClassName. The pattern $s : A$ in ATL is just encoded to the reversed pattern $\{A : \$s\}$ since we encode the model in the reverse order. The symbol \$ is just used for clarifying the variable pattern in the UnQL language.

$$\mathit{inPat2arg}\ (s : A) = \{\text{ClassName} : \{A : \$s\}\}$$

The transformation function $outPatSeq2unql$ is applied to $outPatSeq$ in the rule. The output pattern $outPat$ in ATL specifies each of the produced components. A rule may produce one or more components in the target model from a component in the source model, although the example in Fig. 1 produces just one component. The variables $t_1, t_2, \ldots$, each of which is bound to some component in the target model, may be used in the output patterns $outPatSeq$. We encode the rule into a mutually recursive functions in UnQL, where we use the name of the variables $t_1, t_2, \ldots$ in the encoded UnQL expression. Since in UnQL the value of the result of application of the function should be a graph, we just select one variable t_1 from the variables $t_1, t_2, \ldots$.

$$\begin{aligned}&\mathit{outPatSeq2unql}\ (t1\ :\ ty1\ (\mathit{binds}_1),\ t2\ :\ ty2\ (\mathit{binds}_2),\ \ldots) =\\ &\quad \mathbf{letrec}\ t1 = \mathit{outPat2unql}\ (ty1\ (\mathit{binds}_1))\\ &\qquad\quad\ \ t2 = \mathit{outPat2unql}\ (ty2\ (\mathit{binds}_2))\\ &\qquad\quad\ \ \ldots\\ &\quad \mathbf{in}\ t1\end{aligned}$$

The function $outPat2unql$ is applied to each output patterns. An output pattern consists of an identifier and a tuple of bindings. The identifier determines the class of the output pattern, so we attach the constant pattern ClassName.

$$\begin{aligned}&\mathit{outPat2unql}\ (B\ (\mathit{bind}1,\ \mathit{bind}2,\ \ldots)) =\\ &\quad \{\text{ClassName}\ :\ \{B\ :\ (\mathit{bind2unql}\ \mathit{bind}_1) \cup (\mathit{bind2unql}\ \mathit{bind}_2) \cup \ldots\}\}\\ &\mathit{bind2unql}(m \leftarrow \mathit{oclExp}) =\\ &\quad \mathbf{select}\ \{m\ :\ \$g\}\\ &\quad \mathbf{where}\ \mathit{oclExp2unqlBinds}\ \$g\ \mathit{oclExp}\end{aligned}$$

The right-hand side of each binding is a subset of OCL expressions. The bindings are transformed and put in the where clause of the select-where construct in UnQL. We produce the bindings by applying the function *oclExp2unqlBinds*, defined as follows.

$$\begin{aligned}
&\mathit{oclExp2unqlBinds}\ p\ v = p\ \mathbf{in}\ \$v\\
&\mathit{oclExp2unqlBinds}\ p\ (vs\ .\ v) = \mathit{oclExp2unqlBinds}\ \$g\ vs\ (\$g:\ \text{fresh})\\
&\qquad\qquad\qquad\qquad\qquad\qquad \{v\ :\ p\}\ \mathbf{in}\ \$g\\
&\mathit{oclExp2unqlBinds}\ p\ \mathit{string} = p\ \mathbf{in}\ \{\mathit{string}\ :\ \{\}\}\\
&\mathit{oclExp2unqlBinds}\ p\ (e1 + e2) = \mathit{oclExp2unqlBinds}\ \{\$l1\ :\ \{\}\}\ e1\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{oclExp2unqlBinds}\ \{\$l2\ :\ \{\}\}\ e2\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad p\ \mathbf{in}\ \{\$l1\ \text{++}\ \$l2\ :\ \{\}\}
\end{aligned}$$

One thing to note here is the sequence of ids separated by dot is encoded by sequence of bindings, where fresh variables are introduced for each binding. Another thing to note is that string concatenation is encoded in the string concatenation in UnQL, which is represented by ++ here.

By applying the algorithm above to the ATL example in Fig. 1, we obtain the UnQL functions in Fig. 10. Note that `dummy` is used as a dummy label for making mutually recursive functions. Note also that the actual UnQL does not allow patterns of general form in the argument part but here we used them for simplifying the presentation as we mentioned in Section 2.2. After obtained these functions, we apply these functions recursively to the encoded source model. We define the following functions to do this application.

$$\begin{aligned}
&\mathbf{sfun}\ \mathit{mapClass2Table}\ (\{\text{ClassName} : \$g\}) = f_1\ \$g\\
&\quad |\ \mathit{mapClass2Table}\ \{\$l : \$g\} = \{\$l : \mathit{mapClass2Table}\ \$g\}\\
&\mathbf{sfun}\ f_1\ (\{\text{Class} : \$g\}) =\\
&\qquad \mathit{Class2Table}\ (\{\text{ClassName} : \{\text{Class} : \mathit{mapClass2Table}\ \$g\}\})\\
&\quad |\ f_1\ \{\$l : \$g\} = \{\$l : \mathit{mapClass2Table}\ \$g\}\\
&\\
&\mathbf{sfun}\ \mathit{mapAttribute2Column}\ (\text{ClassName} : \$g\}) = f_2\ \$g\\
&\quad |\ \mathit{mapAttribute2Column}\ \{\$l : \$g\} = \{\$l : \mathit{mapAttribute2Column}\ \$g\}\\
&\mathbf{sfun}\ f_2\ (\{\text{Attribute} : \$g\}) =\\
&\qquad \mathit{Attribute2Column}\ (\{\text{ClassName} : \{\text{Attribute} : \mathit{mapAttribute2Column}\ \$g\}\})\\
&\quad |\ f_2\ \{\$l : \$g\} = \{\$l : \mathit{mapAttribute2Column}\ \$g\}
\end{aligned}$$

Then we apply these functions to the encoded source model as follows.

$$\mathit{mapAttribute2Column}\ (\mathit{mapClass2Table}\ \$db)$$

Here $\$db$ represents the encoded source model. We omit the algorithm for generating these functions since it is fairly straightforward.

We have implemented the algorithm in a functional language called OCaml. The complexity of the algorithm is linear in the size of the input model and the rule descriptions. For the examples given in this paper, the program in OCaml works well.

```
    sfun Class2Table ({ClassName:{Class:$s}}) =
       letrec
       sfun t ({_:{}}) =
           {ClassName:
               {Table:
                   select {name:$a}
                   where $b in $s,
                         {name:$a} in $b
                   U
                   select {col:$c}
                   where $d in $s,
                         {attr:$c} in $d
               }
           }
        in t({dummy:{}})
    sfun Attribute2Column ({ClassName:{Attribute:$s}}) =
        letrec
        sfun t ({_:{}}) =
            {ClassName:
                {Column:
                    select {name:$a}
                    where $b in $s,
                          {name:$a} in $b
                }
            }
       in t({dummy:{}})
```

Fig. 10. UnQL functions obtained by encoding the example ATL

5 Conclusions

In this paper we presented an approach to bidirectionalizing a subset of ATL. Although small, the core part of the ATL is shown to be bidirectionalized. The prototype implementation in OCaml is available at `http://www.biglab.org/src/icmt11/index.html`. The program works by putting it in the src directory in the source code of the GRoundTram system. This work is a first step toward realizing a practical bidirectional model transformations. We believe this approach is promising and in the future we will further develop it on the settings with less restrictions on the ATL transformations.

Acknowledgments

We would like to thank Massimo Tisi and Frederic Jouault for valuable discussions and for providing us simple examples of ATL rules. The research was supported in part by the Grand-Challenging Project on "Linguistic Foundation for Bidirectional Model

Transformation” from the National Institute of Informatics, Grant-in-Aid for Scientific Research (B) No. 22300012, 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. The ATL web site. `http://www.eclipse.org/m2m/atl/`.
2. The BiG project web site. `http://www.biglab.org/`.
3. ATLAS group. KM3: Kernel MetaMetaModel manual. `http://www.eclipse.org/gmt/atl/doc/`.
4. François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.
5. Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. In *ACM SIGPLAN International Conference on Functional Programming*, pages 193–204. ACM, 2010.
6. Jean Bezivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model transformation in practice workshop announcement. In *Satellite Events at the MoDELS 2005 Conference*, pages 120–127. Springer-Verlag, 2005.
7. Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In Stijn Vansummeren, editor, *PODS*, pages 338–347. ACM, 2006.
8. 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.
9. Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *International Conference on Model Transformation (ICMT 2009)*, pages 260–283. LNCS 5563, Springer, 2009.
10. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL ’05: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 233–246, 2005.
11. Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 205–216. ACM, 2010.
12. Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Towards a compositional approach to model transformation for software development. In *SAC’09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 468–475, New York, NY, USA, 2009. ACM.
13. Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.
14. Frederic Jouault, Freddy Allilaire, Jean Bezivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.
15. Frederic Jouault and Ivan Kurtev. Transforming models with ATL. In *Proceedings of Satellite Events at the MoDELS 2005 Conference*, pages 128–138. LNCS 3844, Springer, 2006.
16. Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 47–58. ACM Press, October 2007.

17. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proc. 10th MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.
18. Janis Voigtländer. Bidirectionalization for free! (pearl). In *POPL '09: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 165–176, New York, NY, USA, 2009. ACM.
19. Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Combining syntactic and semantic bidirectionalization. In *ACM SIGPLAN International Conference on Functional Programming*, pages 181–192. ACM, 2010.
20. Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 164–173. ACM Press, November 2007.