Towards A Trace-based Approach to Increasing the Comprehensibility and Predictability of Bidirectional Graph Transformations

Soichiro Hidaka, Martin Billes, Quang Minh Tran

Bidirectional graph transformation is expected to play an important role in model-driven software engineering where changes in the artifacts are reflected not only from upstream to downstream, but also into the other direction, in the compositions of transformations that refine the artifacts from high level specifications down to concrete ones closer to implementation.

However, how the changes are propagated to which part, and what kind of updates are successfully propagated, is difficult to predict if not impossible, especially when the transformation becomes more complex.

In this paper, we propose, in the compositional framework of bidirectional graph transformation, a well-defined tracing mechanism between source, transformation and target. In our GUI, when the user selects some part of these artifacts, the corresponding parts in the other artifacts are highlighted. It helps the user predict to which part of the source the modification is reflected, or which part of the code the user should modify in case the part comes from the transformation. Moreover, edges of the target artifacts are rendered in a classified manner so that any edits in the constant class are always rejected while in the rest of correlated classes, simultaneous edits of multiple edges in a class amount to inconsistent edits so users can know without executing backward transformation which parts can be safely edited. Our graph transformation framework has been made comprehensible and predictable by our proposals.

1 Introduction

Models being used for Model-Driven Software Engineering can become large, so large in fact that they become difficult to comprehend for the developers. To allow for better comprehension, one can use a view of the model to highlight certain interesting aspects of it. A view of a model does not have to serve simplifcation alone: it can provide a different perspective on the model information, like the classical transformation from classes to relational databases. With views arises the problem of view updating. When the view of a model is modified, the source model should have those changes reflected, if possible. This view-update problem can be solved with bidirectional transformations [5]. The view-update problem is widely studied in the database community [1] [6] and more recently the programming language community [7] [2].

In software engineering, bidirectional transformation is expected to play an important role, especially in the generative approach taken in model-driven software development in which models are manipulated by transformations to generate models and other artifacts, where changes in the downstream of the chain of transformations are expected to be reflected back to upstream, so that, for example, defect found and fixed downstream can be propagated to the upstream to avoid reproducing the defect.

Such bidirectional transformations consist of two parts (see Fig. 1): forward transformation ($\mathcal{F}$) and backward transformation ($\mathcal{B}$) [14]. A source model is transformed to a view model via a forward trans-

---

グラフ双方向変換のトレースに基く分かり易さと予測可能性の向上にむけて

日高 宗一郎, 国立情報学研究所, National Institute of Informatics.

マーティン ビレス, アウグスブルク大学, Augsburg University.

クアン ミン トラン, ダイムラー IT イノベーションセンター/ベルリン工科大学, Daimler Center for IT Innovations, Technical University of Berlin.

formation. The forward transformation may discard some of the information found in the source and create some new information on its own. It also has the possibility to rearrange the information found in the source model, sometimes duplicating parts of the source model. For this paper, we assume source and view models to be represented in the form of edge-labeled graphs, G_S and G_V, as we consider model transformation as our primary application and models are represented by graphs [10] [11]. The supporting bidirectional graph transformation [8] is based on bidirectional interpretation of existing unidirectional graph transformation language based on structural recursion on unstructured data [3]. The bulk semantics of the structural recursion that allows per-edge independent computation is utilized in the bidirectionalization.

In order to make bidirectional transformation useful, we need round-trip properties [5]. In our setting, given forward transformation $\mathcal{F}[\![e]\!]\rho$ to compute forward transformation e under variable binding ρ (ρ corresponds to the source, and at the start of the transformation it includes the global variable that binds the input graph) and backward transformation $\rho' = \mathcal{B}[\![e]\!](\rho, G')$ that produces updated source ρ' given the updated view graph G' and the original source ρ, we demand that bidirectional transformation satisfy the GetPut and WPutGet (weaker notion of PutGet [7] or Consistency [1] because of the rather arbitrary variable reference allowed in our language) properties [8], which are:

$$\frac{\mathcal{F}[\![e]\!]\rho = G_V}{\mathcal{B}[\![e]\!](\rho, G_V) = \rho} \text{ (GetPut)}$$

$$\frac{\mathcal{B}[\![e]\!](\rho, G'_V) = \rho' \quad \mathcal{F}[\![e]\!]\rho' = G''_V}{\mathcal{B}[\![e]\!](\rho, G''_V) = \rho'} \text{ (WPutGet)}$$

where GetPut says that if the view is not updated after forward transformation, then the result of the following backward transformation agrees with the original source, and W(Weak)PutGet says that, the result G''_V of the forward transformation using the result ρ' of a backward transformation with updated view G'_V may differ from G''_v, but the backward transformation using G''_V with the *original* source ρ produces ρ' again. In other words, the effect of G''_V relative to ρ is the same as the effect of G'_V.

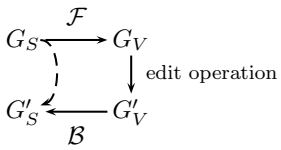

Fig. 1 Scheme of Bidirectional Transformation

One question raised is whether backward transformation should be total or partial. In our case, we want to look explicitly at partial backward transformation which rejects some updates on the view graph.

Given elements of the view such as nodes or edges, it may be hard to track the origin of that element. It is not immediately apparent whether a part has its origin in the source graph or whether it was created new in the transformation query. When looking at the query itself, it is also ambiguous which view elements are created by a query construct at hand.

After edits to the view graph have been made, backward transformation is executed to reflect the view changes back to the source model. We shall restrict the changes that can be made to the view graph to in-place updates where only the edge label is edited, not looking at cases of insertion or deletion of edges. Some updates might be rejected by backward transformation. In particular, edits of an edge label cannot be reflected back if the label appears as a constant of the query. This is because the query itself cannot be updated by the process of backward transformation. If one of the updates made to the view graph is rejected, backward transformation fails as a whole. After a collection of update operations has been piled up, it becomes quite difficult to track back the reason for a failing backward transformation: the transformation itself is unable to give meaningful error messages due to the bulk semantics we use in our implementation. In these bulk semantics, the view graph first needs to be decomposed into individual parts before transformation is applied to it. That way, the error messages presented by backward transformation when it fails are in the local context of the decomposed graph and cannot easily be understood in a global context. With our framework of tracing and classification of view edges, we can tell

for example which part of the query contribute to the view, different edits conflicts (classification), or what part of the query caused the problem (such as constant update violation).

There are some existing work that share our goals. For tracing, Van Amstel et al. [20] proposed the visualization of traces, but in the unidirectional model transformation settings. For the classification, Matsuda and Wang's work [16] proposed the extention of semantic bidirectionalization to cope with limitation of existing semantic bidirectionalization and used mechanism that turned out to be very similar to our classification framework. We discuss the correspondence in the related work section (Section 7) in more detail.

The rest of the paper is organized as follows. Section 2 is the preliminary content for the rest of the paper, summarizing the semantics of our underlying graph data model, core graph language UnCAL and the user-level syntax UnQL. Readers already familiar with them can safely skip the first two subsections of this section. Section 3 motivates our work by examples, which are also used in the following sections. Section 4 proposes tracing mechanism that supports the highlighting of the correspondence between source, transformation and target. Section 5 proposes an algorithm to classify the edges in the target to support showing the editability (whether particular editing of the edge(s) fails or not). Section 6 describes how the proposed mechanisms in the preceding sections are integrated in our bidirectional graph transformation system GRoundTram (Graph Roundtrip Transformation for Models) [10] [11]. Section 7 discusses related work, and Section 8 concludes with future work.

2 Preliminaries

Our graph model builds upon graph constructors (see Fig. 2) that are a subset of the UnCAL (Unstructured CALculus) query language [3]. This language allows us to describe all graphs with respect to bisimulation. A forward transformation query is written in the UnCAL language or alternatively in UnQL (Unstructured Query Language) [3] which is syntactic sugar on top of UnCAL and is transformed to UnCAL before executing. The most important element of the UnCAL language is the structural recursion construct **rec**, which executes

$$
\begin{array}{rll}
e ::= & \{\} \mid \{l : e\} \mid e \cup e \mid \&x := e \mid \&y \mid () & \\
\mid & e \oplus e \mid e \mathbin{@} e \mid \mathbf{cycle}(e) & \{ \text{ constructor } \} \\
\mid & \$g & \{ \text{ graph variable } \} \\
\mid & \mathbf{if}\ l = l\ \mathbf{then}\ e\ \mathbf{else}\ e & \{ \text{ conditional } \} \\
\mid & \mathbf{let}\ \$g = e\ \mathbf{in}\ e \mid \mathbf{llet}\ \$l = l\ \mathbf{in}\ e & \{ \text{ variable binding } \} \\
\mid & \mathbf{rec}(\lambda(\$l, \$g).e)(e) & \{ \text{ structural recursion application } \} \\
l ::= & a \mid \$l & \{ \text{ label } (a \in \mathit{Label}) \text{ and label variable } \}
\end{array}
$$

Fig. 3 Core UnCAL Language

a query fragment in bulk independently on a set of edges. The raw view graph will have trace information stored in the node IDs that allow backward transformation to reverse the bulk semantics algorithm and properly identify the iteration that a view edge was created by.

In the rest of this section, we explain the graph data model we use, the UnCAL and UnQL languages in more detail, and then the additional extension of UnCAL forward transformation used in the following sections.

2.1 UnCAL

The graphs we deal with are multi-rooted, edge-labeled graphs where all information is stored in edge labels, and the label of nodes have no particular meanings other than as identifiers. There is no order between outgoing edges of nodes. The notion of graph equivalence is based on bisimulation, so equivalence between the graphs are efficiently determined [3], and we can always normalize [11] up to isomorphism.

Figure 4 shows examples of our graphs. We represent a graph by a quadruple (V, E, I, O), where V is the set of nodes, E is the set of edges where an edge is represented by a triple of source node, label and destination node, ranges over the set $\mathit{Edge}_\varepsilon$, where labels range over $\mathit{Label} \cup \{\varepsilon\}(\mathit{Label}_\varepsilon)$, and I is the set that identifies the root nodes. Apart from the roots as "entry points" of graphs, we also have "exit points" as represented by the set O. For example, the graph on the left in Figure 4 is represented by (V, E, I, O) where $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, \mathtt{a}, 2), (1, \mathtt{b}, 3), (1, \mathtt{b}, 4), (2, \mathtt{a}, 5), (3, \mathtt{a}, 5), (5, \mathtt{d}, 6), (6, \mathtt{c}, 3)\}$, $I = \{(\&, 1)\}$, and $O = \{\}$.

The roots of the graph are uniquely identified by the input markers denoted like $\&x$ that range over the set Marker. We also call these roots in-

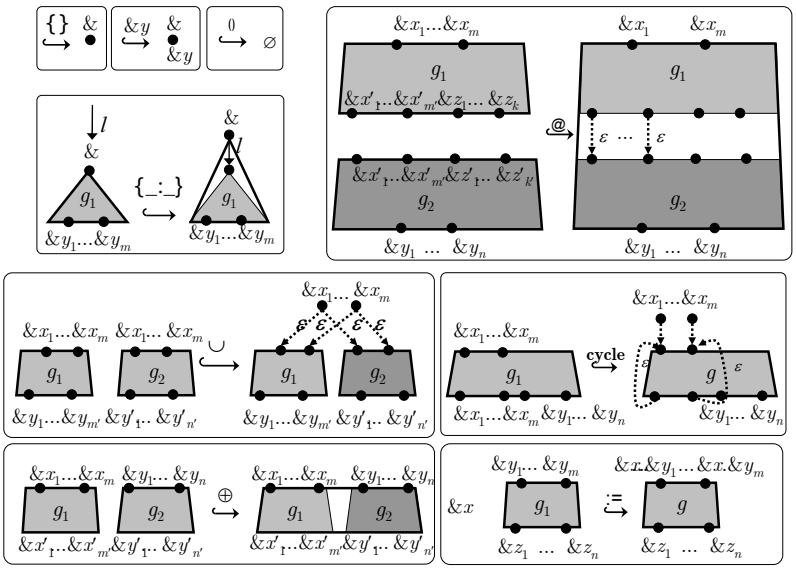

Fig. 2 Graph Constructors of UnCAL

put nodes and denote the input node identified by marker $\&x$ by $I(\&x)$. A special marker $\&$ is called the default marker, and we used the marker in the example. The nodes that correspond to "exit points" are the nodes that are associated with markers, but indicated by O instead of I, like $\{(4, \&y)\}$ if the node 4 has output marker $\&y$. The graph in the example has no output marker. We call the nodes that are identified by O the output nodes. Output nodes are used as connecting points to other graphs, as we will explain shortly. We denote each component of the quadruple using "." syntax, like $g.\mathrm{V}$ for graph g. We represent the type of the graphs by $DB^{\mathcal{X}}_{\mathcal{Y}}$ where $\mathcal{X}$ is the set of input markers and $\mathcal{Y}$ is the set of output markers. The graph in the above example has type $DB^{\&}_{\emptyset}$. We omit the superscript if the set is $\{\&\}$ and subscript if the set is empty. So we also denote the type of the example graph simply by DB. We denote the (static) type of UnCAL expression e similarly by $e :: DB^{\mathcal{X}}_{\mathcal{Y}}$.

In UnCAL, we have nine graph constructors (Fig. 3) whose semantics is illustrated in Fig. 2. We used hooked arrows ($\hookrightarrow$) in the figure stacked with the graph constructor operator symbols to denote the computation by the constructors where left hand side is the operand(s) and the right hand side is the result of the construction. There are three nullary constructors. $()$ constructs a graph without any nodes nor edges. Consequently no input and output marker is present either, i.e., $\mathcal{F}[\![()]\!] \in DB^{\emptyset}$. $\{\}$ constructs a graph with a node with default input marker ($\&$) and no edges, so $\mathcal{F}[\![\{\}]\!] \in DB$. $\&y$ constructs a graph similar to $\{\}$ with additional output marker $\&y$ associated with the node, i.e., $\mathcal{F}[\![\{\}]\!] \in DB_{\{\&y\}}$

$\{\_ : \_\}$ takes a label l and a graph $g \in DB_{\mathcal{Y}}$ and prepend an edge labeled l to the root of the graph g, and the the source of the prepended edge becomes the new root of the resultant graph with default input marker, thus $\{l : g\} \in DB_{\mathcal{Y}}$. The graph union $g_1 \cup g_2$ for graphs $g_1 \in DB^{\mathcal{X}}_{\mathcal{Y}_1}$ and $g_2 \in DB^{\mathcal{X}}_{\mathcal{Y}_2}$ with identical set of input markers $\mathcal{X} = \{\&x_1, \ldots, \&x_m\}$, new m root nodes associated with each element of $\mathcal{X}$ are constructed, and for each of these nodes, two ε-edges are extended from the new root to the original roots of operand graphs for each of such original input marker, thus $g_1 \cup g_2 \in DB^{\mathcal{X}}_{\mathcal{Y}_1 \cup \mathcal{Y}_2}$.

The input node renaming operator $:=$ takes a marker $\&x$ and a graph $g \in DB^{\mathcal{Y}}_{\mathcal{Z}}$ for $\mathcal{Y} = \{\&y_1, \ldots, \&y_m\}$, and returns a graph whose input markers are prepended by $\&x$, thus $(\&x := g) \in DB^{\&x.\mathcal{Y}}_{\mathcal{Z}}$ where the dot "." operation is a marker concatenation operation that, together with default input marker $\&$ forms a monoid, i.e., $\&.\&x = \&x.\& = \&x$ for any marker $\&x \in \mathit{Marker}$, and $\&x.\mathcal{Y} = \{\&x.\&y_1, \ldots, \&x.\&y_m\}$ for

$\mathcal{Y} = \{\&y_1, \ldots, \&y_m\}$. In particular, when $\mathcal{Y} = \{\&\}$, the $:=$ operator just assign a new name to the root of the operand, i.e., $(\&x := g) \in DB_{\mathcal{Y}}^{\{\&x\}}$ for $g \in DB_{\mathcal{Y}}$.

The disjoint union $g_1 \oplus g_2$ of two graphs $g_1 \in DB_{\mathcal{X}'}^{\mathcal{X}}$ and $g_2 \in DB_{\mathcal{Y}'}^{\mathcal{Y}}$ with disjoint set of input markers, i.e., $\mathcal{X} \cap \mathcal{Y} = \emptyset$, the resultant graph inherits all the markers, edges and nodes from the operands, thus $g_1 \oplus g_2 \in DB_{\mathcal{Y}' \cup \mathcal{Y}'}^{\mathcal{X} \cup \mathcal{Y}}$.

The remaining two constructors connect output and input nodes with matching markers by ε-edges. Graph append $g_1 \mathbin{@} g_2$ of two graphs $g_1 \in DB_{\mathcal{X}' \cup \mathcal{Z}}^{\mathcal{X}}$ and $g_2 \in DB_{\mathcal{Y}}^{\mathcal{X}' \cup \mathcal{Z}'}$ with matching subset of markers $\mathcal{Y}$ connects the output nodes and input nodes with matching markers $\mathcal{X}'$ and discards the rest of the markers, thus $g_1 \mathbin{@} g_2 \in DB_{\mathcal{Y}}^{\mathcal{X}}$. As a useful idiom, this operator is often used to project (select) one input marker and rename it as the default input marker, while discarding the rest of the input makers (and thus they are left unreachable), like taking $\mathcal{X} = \{\&\}$, $\mathcal{Z} = \emptyset$ and $\mathcal{X}' = \{\&x'\}$, i.e., $\&x' @ g_2$ to return a graph g_2 with the input marker $\&x'$ renamed to the default one and leave the subgraphs reachable from the rest of the input nodes unreachable. The **cycle** operator does operations similar to @ but in an intra-graph instead of inter-graph manner. For a graph $g \in DB_{\mathcal{X} \cup \mathcal{Y}}^{\mathcal{X}}$ with $\mathcal{X} \cap \mathcal{Y} = \emptyset$, **cycle**$(g)$ connects output and input nodes of g with matching markers $\mathcal{X}$ and construct m nodes with input markers in $\mathcal{X}$, connected with the original input nodes of g with matching input markers by ε-edges. The output markers in $\mathcal{Y}$ are left as is.

It is worth noting that any graph in the data model can be expressed by using these UnCAL constructors (up to bisimilarity). For example the graph on the left in Figure 4 can be represented by

$$\begin{aligned}\&n_1 @ \mathrm{cycle}((&\&n_1 := \{\mathtt{a}{:}\&n_2, \mathtt{b}{:}\&n_3, \mathtt{b}{:}\&n_4\},\\ &\&n_2 := \{\mathtt{a}{:}\&n_5\},\\ &\&n_3 := \{\mathtt{a}{:}\&n_5\},\\ &\&n_4 := \{\},\\ &\&n_5 := \{\mathtt{d}{:}\&n_6\},\\ &\&n_6 := \{\mathtt{c}{:}\&n_3\})).\end{aligned}$$

The semantics of conditionals is standard, except the condition is restricted to label equivalence comparison. There are two kinds of variables: label variables denoted like $\$l$ or $\$l_1$, binds labels, while graph variables denoted by $\$g \ldots \$g_1 \ldots$ binds graphs. They are introduced by structural recursion operator **rec**. For example, the following transformation in UnCAL replaces every label `a` by `d` and remove (shortcut) edges labeled `c`. So if the graph variable $\$db$ is bound to the graph on the left of Figure 4, the result will be (after ε removal which is conducted in a standard manner like those for automata, and flattening the node identifiers) the one on the right of the figure.

$$\begin{aligned}\mathbf{rec}(\lambda(\$l, \$g).\ &\mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then}\ \{\mathtt{d} : \&^1\}^2\\ &\mathbf{else\ if}\ \$l = \mathtt{c}\ \mathbf{then}\ \{\varepsilon : \&^3\}^4\\ &\mathbf{else}\qquad\qquad\quad \{\$l : \&^5\}^6)(\$db)^7\end{aligned}$$

We call the first operand of **rec** the the body expression and the second operand the argument expression. In the above transformation, the body expression is an **if** conditional, and argument expression is the variable reference $\$db$. We use $\$db$ as a distinguished global variable to represent the input of the graph transformation.

For the sake of bidirectionalization (and also used in our tracing in this paper), we allocate for each node of the abstract syntax tree of UnCAL the code position $p \in Pos$. The superscripts of each expression in the above transformation represent such positions.

Figure 5 shows the semantics of **rec** by the above example input graph and transformation. Since the evaluation of **rec** can be executed in parallel for each edge and the subgraph reachable from the target node of the edge (that are correspondingly bound to variables $\$l$ and $\$g$ in the body expression), the semantics is called *bulk semantics*. In the bulk semantics, the node identifier carry some information which has the following structure *TraceID*:

$$\begin{aligned}\mathit{TraceID} ::= &\ \mathit{SrcID}\\ \mid &\ \mathsf{Code}\ \mathit{Pos}\ \mathit{Marker}\\ \mid &\ \mathsf{RecN}\ \mathit{Pos}\ \mathit{TraceID}\ \mathit{Marker}\\ \mid &\ \mathsf{RecE}\ \mathit{Pos}\ \mathit{TraceID}\ \mathit{Edge},\end{aligned}$$

where *SrcID* is the base case and represents the node identifier in the input graph, Code p $\&x$ denotes the nodes constructed by $\{\}$, $\{\_ : \_\}$, $\&y$, $\cup$ and **cycle** where $\&x$ is the marker that inherited by the corresponding input nodes of the operand(s) of the constructor, and for $\{\}$ $\{\_ : \_\}$ and $\&y$ the marker is the default one and we omit. RecN p v $\&z$ denotes the node created by **rec** at position p for the node v of the graph resulted from evaluating the argument expression. For example, the node RN 7 1 in the figure originated

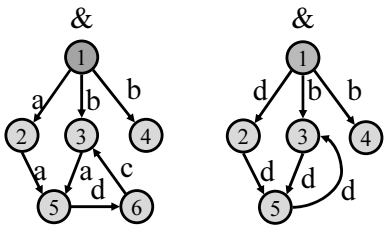

Fig. 4 Cyclic graph examples

from node 1 and 7 is the code position of **rec** in the transformation (RecN is abbreviated to RN in the figure for simplicity, and we do similarly to denote Code by C and RecE by RE). We have six such nodes for each node in the input graph. Then we evaluate the body expression for each binding of $\$l$ and $\$g$. For the edge $(1, \mathtt{a}, 2)$, the result will be $(\{(\mathsf{C}\ 2), (\mathsf{C}\ 1)\}, \{(\mathsf{C}\ 2, d, \mathsf{C}\ 1)\}, \{(\mathtt{\&}, \mathsf{C}\ 2)\}, \{(\mathsf{C}\ 2, \mathtt{\&})\})$, with the nodes C 2 and C 1 constructed by the constructors $\{\_ : \_\}$ and &, respectively. For the shortcut edges, ε-edge is generated similarly. Then each node v of the graph generated by the body expression for edge ζ is wrapped with the trace information RE like RE p v ζ for **rec** at position p. These results are surrounded by round squares drawn with dashed lines in Fig. 5. Then they are connected together according to the original shape of the graph as depicted in Figure 5. For example, the input node RE 7 (C 2) $(1, \mathtt{a}, 2)$ is connected with RN 7 1. After removing the ε-edges and flattening the node identifiers, we obtain the result graph in Figure 4 on the right.

The variable binders **let** and **llet** are our extensions and have standard meanings, and used for optimization by rewriting [9].

The notion of bisimulation is extended to cope with bisimulation, and it is known that every UnCAL expression preserves bisimulation [3], though we do not use this property in this paper. Based on this bulk semantics, it is easy that the termination is always guaranteed, even in the presence of cycles.

In the backward evaluation of **rec** for in-place updates, the process of ε-elimination is reversed to restore the shape like in Figure 5, and then the graph is decomposed with the help of the structures of node with trace IDs, and then the decomposed graph is used for the backward transformation of each body expression. The backward transformation produces as the updated input the variable binding environment (in this body expression we obtain the bindings for $\$l$, $\$g$ and $\$db$ and merge these bindings to produce the final binding of $\$db$. For example, if we update the edge $(1, \mathtt{b}, 3)$ in the view to $(1, \mathtt{x}, 3)$, then it is propagated via the backward transformation of the body expression $\{\$l : \mathtt{\&}\}$, which produces the bindings of $\$l$ updated with x and is reflected to the source graph with edge $(1, \mathtt{b}, 3)$ replaced by $(1, \mathtt{x}, 3)$.

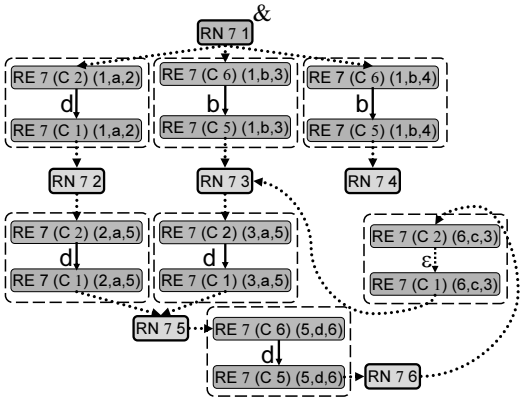

Fig. 5 Bulk semantics by example

2.2 UnQL as a Textual Surface Syntax of Bidirectional Graph Transformation

We use the surface language UnQL [3](Fig. 6), for bidirectional graph transformation. It is bidirectionally interpreted by translation into UnCAL and bidirectional interpretation of the translated UnCAL expression [8]. It is based on the notion of structural recursion. It can be considered as a divide and conquer computation scheme using the following equation:

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

The correspondence with the **rec** in UnCAL with f above is

$$f(g) = rec(e)(g).$$

The structural recursion function f, if given singleton node graph, returns the singleton node graph, or if given a graph with one edge labeled l on the top, followed by subgraph g, then the result will be

(template)	T	::=	$\{L:T,\ldots,L:T\} \mid T \cup T \mid \$g$
			$\mid$ **if** BC **then** T **else** T
			$\mid$ **select** T **where** $B,\ldots,B$
			$\mid$ **letrec sfun** $fname(L:\$G) = \ldots$ **in** $fname(T)$
(binding)	B	::=	Gp **in** $\$G \mid BC$
(condition)	BC	::=	not $BC \mid BC$ and $BC \mid BC$ or BC
			$\mid L = L$
(label)	L	::=	$\$l \mid$ a
(label pattern)	Lp	::=	$\$l \mid Rp$
(graph pattern)	Gp	::=	$\$G \mid \{Lp:Gp,\ldots,Lp:Gp\}$
(regular path pattern)	Rp	::=	a $\mid$ _ $\mid Rp.Rp \mid (Rp\|Rp) \mid Rp? \mid Rp* \mid Rp+$

Fig. 6 Syntax of UnQL

the concatenation of the result of some computation e using l and g, combined by append operation (@) with the recursive result with g. If given horizontal union of two graphs g_1 and g_2, then the result will be the horizontal union of recursive results with g_1 and g_2. Since the first and third equation hold for any f, we only use the second equation for the programming in UnQL.

Thanks to the translation to UnCAL with bulk semantics, we can program graph transformations as if we are programming on (infinite) trees as a result of (conceptually) unfolding the input graphs.

We can combine the structural recursions to extract various graph patterns like regular expressions of edge labels using mutual structural recursion.

A join operation of multiple graph patterns is achieved by nesting of structural recursions. The following UnQL query returns the input graph bound to the variable $\$db$ if the graph has both edge labeled `a` and that labeled `b` at the root, and otherwise returns a graph with single node denoting empty result.

```
letrec sfun f({a:$g})
  letrec sfun f({b:$g}) = {res:$db}
  in f($db)
in f($db)
```

Higher syntactic sugar **select** allows the combination of these query patterns by simple SQL-like syntax, combining querying and prepending edges to the results arbitrarily. For example, the following transformation

```
select {result:$g}
where {a.b:$g} in $db
```

extracts subgraphs under consecutive edges labeled `a` and `b`, and prepends an edge labeled `result` for each of these subgraphs. Next transformation

```
select {result:$name}
where {Class.(srcof.Assoc.dst)+:$class} in $db,
      {isAbstract:{$b:$Any}} in $class,
      $b = true
```

computes the subgraphs with respect to transitive closure of edges by traversing the label pattern `srcof.Assoc.dst` more than once, and choose those having an edge labeled `isAbstract`. This is extracted from an example of Class to Relational transformation appeared in our previous work [11].

All language constructs in UnQL are translated into UnCAL [3][13][12]. In particular, **select** to structural recursion, and structural recursion to **rec** in UnCAL. We highlight the essential part of the translation in the following. Please refer to [3][13][12] for details.

The template part (directly following the **select** clause) appears in the innermost body of the nested **rec** in the translated UnCAL. The edge constructor expression is directly passed through, while the graph variable pattern in the **where** clause and the corresponding reference are translated into a combination of graph variable bindings in nested **rec**s as well as a reference to them in the body of **rec**. For example,

```
select {res:$db}
where {a:$g} in $db,
      {b:$g} in $db
```

is translated into

```
rec(λ($l,$g).if $l = a then
    rec(λ($l,$g).if $l = b
                then {res:$db} else {})($db)
    else {})($db).
```

Regular expression of labels are translated

first into non-deterministic finite automata, and then they are translated into mutually recursive **letrec** **sfun**s, and finally into **rec** in UnCAL using multiple input/output markers and disjoint unions using tupling [15] technique, where tuple construction is encoded by disjoint union, and projection is encoded by graph append operation and output marker expression using idiom described in the previous subsection.

2.3 Forward Semantics with Traceable View and Intermediate Results

In the extended forward evaluation of UnCAL we leave not only the trace information in the node identifiers but also every intermediate result of the operands. We represent this information by extending the abstract syntax with the results subscripted on the bottom left. We annotate an UnCAL expression e with overline $\overline{e}$ to denote this extended abstract syntax, but omit the overline if it is clear from the context. Moreover, if the subscripts are not used, we omit them as well. For example, for the edge constructor expression $\{e_1 : e\}$, we have

$$\begin{aligned} \mathcal{F}[\![\{e_1 : e\}]\!]\rho &= {}_G\{\overline{e_1} : \overline{e}\} \\ \text{where } G &= \{L : G_0\} \\ {}_L\overline{e_1} &= \mathcal{F}[\![e_1]\!]\rho \\ {}_{G_0}\overline{e} &= \mathcal{F}[\![e]\!]\rho. \end{aligned}$$

The special cases are the conditional expression and structural recursion **rec**. For the former, we keep only the branch that is actually evaluated. It is denoted by $\overline{\textbf{if}}\ \overline{{}_b e_{\mathrm{b}}}\ \overline{e}$ where $b : \mathit{Bool}$ represents the run-time branching behavior and it is true if the **then** branch has been taken and false otherwise, and $\overline{e}$ is the extended evaluation result of the body expression for the corresponding branch. For the latter, we keep the map from the edge from which the bindings of the label and the graph variable are created, to the result of the extended forward evaluation for the binding. It is denoted by $\overline{\textbf{rec}}(\lambda(\$l, \$g).M)(\overline{e_{\mathrm{a}}})$ where M denotes this mapping.

We denote the set of such extended expressions $\overline{e}$ by *Eval*.

3 Motivating Example

In this section, we want to give an example of a case where the result of backward transformation is rather unpredictable. We use the transformation query shown below and the source graph of Fig. 7 to produce the view graph in Fig. 8, then modify the view graph so that the result is as in Fig. 9.

```
select (if $l = a then {a:$g} else {d:$g})
where {$l:$g} in $db
```

Listing 1 Transformation query in UnQL

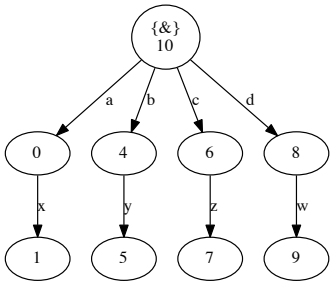

Fig. 7 Example source graph

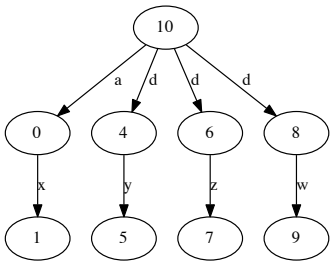

Fig. 8 Graph genrerated by transformation of the graph in Fig. 7

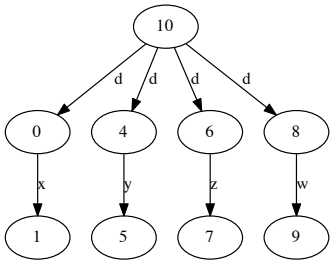

Fig. 9 Modified target graph

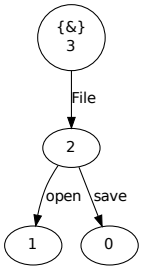

Fig. 10 Class Members

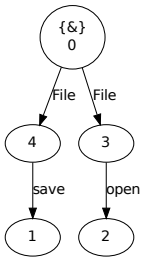

Fig. 11 Qualified Names

As the result of backward transformation, there are serveral possibilities that satisfy bidirectional properties. One is to accept the change, reflecting it by changing a to d. Another forward transformation would produce the same graph as the modified view graph. Another possibility is to change the label a to anything other than a. Another forward transformation would still produce the same graph as the modified view graph. What happens in the backward transformation depends on the underlying backward transformation logic. (The system we implemented rejects the update.)

As a second point of consideration, forward transformation allows us to replicate information from the source graph to produce a view graph which contains some information duplicate. We want to motivate how this can lead to the possibility of inconsistent edits which make backward transformation fail.

```
select {$l:{$l′:$g′}}
where  {$l:$g} in $db,
       {$l′:$g′} in $g
```

Listing 2 Class to Qualified Name

The input graph from Fig. 10 represents a class with several methods. Applying the query above we can duplicate the top-level edge to produce the view graph in Fig. 11 showing the qualified names of the methods. Now the class name "File" can be inconsistently modified: If one label "File" is modified to "Stream" and the other to "Buffer", backward transformation will try to propagate both changes to the same source edge with label "File" and fail because of the inconsistency of the edit operations.

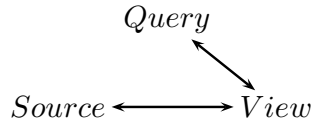

Fig. 12 Tracing mechanisms

We introduce the notion of equivalence classes and say that two view edges are considered to be in the same equivalence class if backward transformation would cause edits to those view edges to propagate to the same source edge, allowing for the possibility of inconsistent edits which make backward transformation fail. There is one such equivalence class for each source edge that the changes can be propagated back to. Additionally, there is a class of view edges which cannot be modified at all because their label originates from a constant defined in the transformation query, as is seen with the edges with label d in Fig. 8.

We (1) mark the edge that reject any updates, (2) highlight the source edge to which the modification would be propagated. (3) highlight the code fragment that generates the edge in the view graph on which user is trying to update. (4) classify the edges in the target edges with the same origin in the source are identified. Inconsistent updates on them would be rejected by the backward transformation.

4 Tracing Mechanisms

In this section, we will elaborate on mechanisms that allow us to tell the correspondence between elements of the source graph, code positions of the transformation query and elements of the view graph in all feasable combinations (Fig. 12).

The UnQL query in Listing 1 is converted into the following UnCAL expression.

```
rec (λ($l,$g).
    if $l = a then {a:$g}
    else {d:$g})($db)
```

Listing 3 UnCAL expression of UnQL query in Listing 1

In the graphical user interface of our system, when a view edge is selected, the corresponding source edge or corresponding query code point is highlighted, and vice versa.

4.1 Tracing between Source and View

A view edge may either be freshly created by a edge constructor in the query or originate from an edge in the source graph. If a view edge originates from a source graph, it is very helpful for the user to be able to visually see the correspondending source edge of a view edge, particularty in complex transformation scenario.

The trace information associated with node identifiers that is used during backward transformation is sufficient for directly identifying the source of a view edge without executing backward transformation in its whole. If a view edge ζ is from the source graph, it must have a trace ID of the form (n_1, a, n_2) where n_1 and n_2 are source IDs. If the edge is created inside the body of a structural recursion at code position p, it will have the form $(\mathsf{RecE}\, p\ u\ \zeta', a, \mathsf{RecE}\, p\ v\ \zeta')$. Here, the edge is the result of the evaluation of the structural recursion at ζ' and (u, a, v) is edge within the graph created by the body of the structural recursion. Hence, we expand the tracing information by looking into (u, a, v) to detect the source edge. Nested **rec**-constructs generate nested RecE traceable view which can be expanded in the same way.

Formally, the following function

$$\mathsf{tr\_corr} : \mathit{Edge} \to \mathit{Edge}$$

returns the source edge of a given view edge if it exists and fails otherwise.

$$\mathsf{tr\_corr}((u, a, v)) = (u, a, v) \text{ if } u, v \text{ in } \in \mathit{SrcID}$$
$$\mathsf{tr\_corr}(((\mathsf{RecE}\, p\ u\ \zeta'), a, (\mathsf{RecE}\, p\ v\ \zeta'))) = \mathsf{tr\_corr}((u, a, v))$$
$$\mathsf{tr\_corr}(\zeta) = \mathsf{FAIL} \text{ otherwise}$$

After a corresponding source edge has been found for each view edge that has one, this relationship can be easily reversed to allow tracing from the source to the view:

$$\{\zeta_t \mid \zeta_t \in E_V, \zeta_s = \mathsf{tr\_corr}\, \zeta_t\},$$

where E_V is the set of view edges. The notion of "corresponding source edge" defined here is the copy relationship. The source edge calculated by this algorithm is a true copy of the view edge fed to the algorithm.

For example, if we apply $\mathsf{tr\_corr}$ to the edge $(0, \mathtt{x}, 1)$ in the view graph (Fig. 8), we obtain the edge $(0, \mathtt{x}, 1)$ in the source graph (Fig. 7) which is the source edge from which the view edge originates from. This obtained edge can then be highlighted accordingly.

We can relax the notion of corresponding source edge by weakening the second case:

$$\mathsf{tr\_corr}(((\mathsf{RecE}\, p\ u\ \zeta'), a, (\mathsf{RecE}\, p\ v\ \zeta'))) = \begin{cases} \mathsf{tr\_corr}((u, a, v)) \text{ if } \mathsf{tr\_corr}((u, a, v)) \neq \mathsf{FAIL} \\ \mathsf{tr\_corr}(\zeta') \text{ otherwise} \end{cases}$$

In this definition, the view edge is either a copy of the source edge or it is the result of applying a structural recursion to the source edge. Note that, if tracing (u, a, v) fails, we trace back to the source edge of ζ', the edge being applied by the structural recursion.

4.2 Tracing between View and Query

Since view edges can be copied by graph variables $\$g$ or created by edge constructors $\{l : e\}$, we can highlight the correspondence between view edges and graph variables as well as edge constructors in the query.

4.2.1 Tracing between View and Edge Constructor

If the edge is constructed by an edge constructor in the query, we can reutilize the function for tracing to the source edge in Section 4.1 and, instead of looking for a trace ID with a source ID, look for an edge ID of the form $(\mathsf{Code}\, p\ m, \_, \_)$. The expansion of **rec**-constructs (RecE) works the same way.

In particular, given a view edge ζ, the following function returns the code position p of the edge constructor that creates ζ

$$\mathsf{tr\_cp} : \mathit{Edge} \to \mathit{Pos}$$
$$\mathsf{tr\_cp}((\mathsf{Code}\ \ p\ \_, \_, \_)) = p$$
$$\mathsf{tr\_cp}(((\mathsf{RecE}\, p\ u\ \zeta'), a, (\mathsf{RecE}\, p\ v\ \zeta'))) = \mathsf{tr\_cp}((u, a, v))$$
$$\mathsf{tr\_cp}(\zeta) = \mathsf{FAIL} \text{ otherwise}$$

The edge constructor at p can then be highlighted when selecting the view edge. For example, if the user clicks on the view edge $(10, \mathtt{a}, 0)$ in Fig. 8, the edge constructor $\{\mathtt{a} : \$g\}$ in Listing 3 is highlighted because this constructor creates the view edge ($\mathsf{tr\_cp}$ returns p when applied to the view edge). No edge in the source graph (Fig. 7) is highlighted because the view edge is not copy of any edge in the source graph.

4.2.2 Tracing between View and Graph Variable

If the view edge is a direct copy from the source, a graph variable can be highlighted in the query

instead. This graph variable was used to output the view edge to the view graph.

For a view edge $\zeta = (u, a, v)$ with $a \neq \epsilon$, we define the *sequence of applied edges* of ζ as the sequence of edges that were successively applied by nested structural recursions to create ζ as follows:

$$\mathsf{tr\_eval} : \mathit{Edge} \rightarrow [\mathit{Edge}]$$
$$\mathsf{tr\_eval}(((\mathsf{RecE}\ p\ u\ \zeta'), a, (\mathsf{RecE}\ p\ v\ \zeta'))) = \mathsf{tr\_eval}((u, a, v)) +\!\!+ [\zeta']$$
$$\mathsf{tr\_eval}(\zeta) = [] \text{ otherwise}$$

In the first case, if ζ has the form $((\mathsf{RecE}\ p\ u\ \zeta'), a, (\mathsf{RecE}\ p\ v\ \zeta'))$, it is produced by the structural recursion at code position p when applying the body to the edge ζ'. Then, we recursively collect applied edges in (u, a, v).

We also define the *source edge* of ζ as the result of applying the following function to ζ.

$$\mathsf{tr\_srcEdg} : \mathit{Edge} \rightarrow \mathit{Edge}$$
$$\mathsf{tr\_srcEdg}(((\mathsf{RecE}\ p\ u\ \zeta'), a, (\mathsf{RecE}\ p\ v\ \zeta'))) = \mathsf{tr\_srcEdg}((u, a, v))$$
$$\mathsf{tr\_srcEdg}(\zeta_S) = \zeta_S$$

In order to trace to graph variable references within expressions with multiple input marker types, identification of the input marker is required to identify the correct occurrence of the reference that produced the edges in the view.

The input marker $\&m$ of a view edge can be identified by traversing the edge backwards to a $\mathsf{RecN}\ p\ \&m$ or $\mathsf{Code}\ p\ \&m$. If no such node exists, the input marker is the input marker of the root node reachable from the edge.

The algorithm for tracing graph variables is then as follows:

First, following the sequence of applied edges, we can trace to the structural recursion body that created ζ. Formally, let $\overline{e}$ be the result of the forward semantics to the input query and ζ a view edge. Moreover, let $Z = \mathsf{tr\_eval}(\zeta)$ and $\zeta_S = \mathsf{tr\_srcEdg}(\zeta)$ be the sequence of applied edges of ζ and the source edge of ζ, respectively.

The body of the structural recursion in which ζ is created is defined as

$$\mathsf{tr\_brec} : \mathit{Eval} \rightarrow [\mathit{Edge}] \rightarrow \mathit{Eval}$$
$$\mathsf{tr\_brec}(\overline{\mathbf{rec}}(\lambda(\$l, \$g).M)(\overline{e_{\mathrm{a}}}))[\zeta' : \zeta s'] = \mathsf{tr\_brec}(M\zeta')\ \zeta s'$$
$$\mathsf{tr\_brec}(\overline{e})[] = \overline{e}$$

Recal that M maps the applied edge to the corresponding (extended) body expresssion so $M\zeta'$ denotes such expression.

Then, we identify the corresponding variable reference by traversing the corresponding marker component in the UnCAL expression $\overline{e}$, resulting from applying $\mathsf{tr\_brec}$ to Z, using the function $\mathsf{tr\_gvar}$ where $\zeta_S = \mathsf{tr\_srcEdg}(\zeta)$ (Fig. 13) and $\pi_2 \circ \mathsf{tr\_gvar}$ returns the set of positions of the graph variables, while $\pi_1 \circ \mathsf{tr\_gvar}$ is used to trace beyond @ and **cycle** expressions that operate on markers. We assume that the type of the expression is annotated for every subexpression using type inference proposed in our previous work [9].

The positions in $\pi_2 \circ \mathsf{tr\_gvar}$ can be highlighted to tell which graph variables produce ζ. For example, applying $\mathsf{tr\_gvar}$ to the view $(0, \mathtt{x}, 1)$ in Fig. 8 will result in $(\{\}, \{p\})$ where p is the code point of the graph variable $\$g$ of the **then** branch, but not of the **else** branch, in Listing 3. We can then highlight this variable.

4. 2. 3 Tracing UnQL

The tracing mechanisms introduced above are defined for UnCAL. However, the user usually formulates a transformation in the more high-level language UnQL described in Section 2. 2. For practical use, it is desired that tracing between view and query also works for UnQL.

The tracing mechanism also works straightforwardly for UnQL based on the following observation: When an UnQL query is desugared to UnCAL, all edge constructors and graph variables in the UnQL query that create edges in the view graph are preserved in the UnCAL query. For instance, edge constructors $\{\mathtt{a} : \$g\}$ and $\{\mathtt{d} : \$g\}$ as well as graph variables $\$g$ of the UnQL query in Listing 1 are transfered to the UnCAL query in Listing 3.

As a result, we can easily highlight edge constructors and graph variables in an input UnQL query. Furthermore, thank to our marker-based tracing of graph variables (Fig. 13), we can handle UnQL queries containing regular expressions which can lead to the generation of multiple markers.

In our system, the user can activate an optimizer that can optimize an UnCAL query to increase the efficiency. However, because of the excessive reorganization of UnCAL expressions during the optimization phase, we currently support neither tracing UnCAL nor tracing UnQL if optimization is activated.

$$
\begin{array}{llll}
 & \mathsf{tr\_gvar} : \{Marker\} \to \{Marker\} \times \{Pos\} & & \\
\mathsf{tr\_gvar}\ \{\&\} & {}_G\overline{\$v^p} & = & (\{\},\{p\}) \text{ if } \zeta_S \in G.\mathrm{E} \\
 & & & (\{\},\{\}) \quad \text{otherwise} \\
\mathsf{tr\_gvar}\ \{\&\} & \{\} & = & (\{\},\{\}) \\
\mathsf{tr\_gvar}\ \{\&\} & \{\_ : \overline{e}\} & = & \mathsf{tr\_gvar}\ \{\&\}\ \overline{e} \\
\mathsf{tr\_gvar}\ \{\&m\} & \overline{e_1} \cup \overline{e_2} & = & \text{let } (\mathcal{Z}_1, P_1) = \mathsf{tr\_gvar}\ \{\&m\}\ \overline{e_1} \text{ in} \\
 & & & \quad (\mathcal{Z}_2, P_2) = \mathsf{tr\_gvar}\ \{\&m\}\ \overline{e_2} \text{ in} \\
 & & & (\mathcal{Z}_1 \cup \mathcal{Z}_2, P_1 \cup P_2) \\
\mathsf{tr\_gvar}\ \{\&\} & \&z & = & (\{\&z\},\{\}) \\
\mathsf{tr\_gvar}\ \{\&m\} & (\overline{e_1} \mathbin{@} \overline{e_2}) & = & \text{let } (\mathcal{Z}_1, P_1) = \mathsf{tr\_gvar}\ \{\&m\}\ \overline{e_1} \text{ in} \\
 & & & \text{let } (\mathcal{Z}_2, P_2) = \mathsf{tr\_gvar}\ \mathcal{Z}_1 \quad \overline{e_2} \text{ in} \\
 & & & (\mathcal{Z}_2, P_1 \cup P_2) \\
\mathsf{tr\_gvar}\ \{\&m\} & \mathbf{cycle}(\overline{e}) & = & \text{let } (\mathcal{Z}, P) = \mathsf{tr\_gvar}\ \{\&m\}\ \overline{e} \text{ in} \\
 & & & (\mathcal{Z} \setminus \mathcal{X}, P) \quad \text{where} \quad e{::}DB^{\mathcal{X}}_{\mathcal{Y}} \\
\mathsf{tr\_gvar}\ \{\&\} & () & = & (\{\},\{\}) \\
\mathsf{tr\_gvar}\ \{\&x.\&m\} & (\&x := \overline{e}) & = & \mathsf{tr\_gvar}\ \{\&m\}\ \overline{e} \\
\mathsf{tr\_gvar}\ \{\&m\} & (\overline{e_1} \oplus \overline{e_2}) & = & \mathsf{tr\_gvar}\ \{\&m\}\ \overline{e_1} \\
 & & & \quad \text{if } e_1{::}DB^{\mathcal{X}}_{\mathcal{Y}} \wedge \&m \in \mathcal{X} \\
 & & & \mathsf{tr\_gvar}\ \{\&m\}\ \overline{e_2} \text{ otherwise} \\
\mathsf{tr\_gvar}\ \{\&x.\&z\} & \overline{\mathbf{rec}}(\overline{e_\mathrm{b}})(\overline{e_\mathrm{a}}) & = & \mathsf{tr\_gvar}\ \{\&z\}\ \overline{e_\mathrm{b}} \\
 & & & \quad \text{where} \quad e_\mathrm{a}{::}DB^{\mathcal{X}}_{\mathcal{Y}} \wedge e_\mathrm{b}{::}DB^{\mathcal{Z}}_{\mathcal{Z}} \\
\mathsf{tr\_gvar}\ \mathcal{Z} & \overline{\mathbf{if}}\ b\ \overline{e} & = & \mathsf{tr\_gvar}\ \mathcal{Z}\ \overline{e} \\
\mathsf{tr\_gvar}\ (\mathcal{X} \cup \mathcal{Y}) & \overline{e} & = & \text{let } (\mathcal{Z}_1, P_1) = \mathsf{tr\_gvar}\ \mathcal{X}\ \overline{e} \text{ in} \\
 & & & \quad (\mathcal{Z}_2, P_2) = \mathsf{tr\_gvar}\ \mathcal{Y}\ \overline{e} \text{ in} \\
 & & & (\mathcal{Z}_1 \cup \mathcal{Z}_2, P_1 \cup P_2) \\
\mathsf{tr\_gvar}\ \{\&m\} & \overline{e} & = & (\{\},\{\}) \text{ if } e{::}DB^{\mathcal{X}}_{\mathcal{Y}} \wedge \&m \notin \mathcal{X}
\end{array}
$$

Fig. 13 Marker-oriented Tracing of Graph Variables

5 Determining Editability

For the following section, we look at the underlying logic of the query in the UnCAL language, which it is translated to if it was originally an UnQL query. We want to use the following query which is translated from Listing 2:

$$
\begin{array}{l}
\mathbf{rec}(\lambda(\$l, \$g). \\
\quad \mathbf{rec}(\lambda(\$l', \$g').\{\$l : \{\$l' : \$g'\}\}(\$g))(\$db)
\end{array}
$$

Whether an edit is accepted or not is dependent on the query construct used to create the individual view edge during forward transformation.

In the UnCAL query language, edges can be created by one of the following:

1. Graph variable usage $\$g$. Edges in a graph variable might be direct copies from the source or created elsewhere by edge constructors in the query.
2. Edge constructors $\{l : e\}$. The label can be
   (a) an explicit label constant $a \in \mathit{Label}$
   (b) a label variable which might be bound to a source edge via structural recursion or equal to another label constant or label variable in an **llet** construct.

Only in the case of direct copies from the source via graph variable usage, the trace information found in these copy edges can directly identify the corresponding source edge as seen above in section 4.1. For 2a, a view edge created by this edge contructor should be considered a constant edge. Updating its label is not allowed, since backward transformation prevents updates to constant labels. For everything else, there needs to be a dynamic (re-)evaluation of the query to determine the exact value of the label or graph variable. The **rec** construct is executed one time for each possible binding of $(\$l, \$g)$ found in the argument e_a. The dynamic evaluation is performed on the

intermediate evaluation results defined above in section 2. 3.

For dynamic evaluation, we define a variable environment $\eta : env$ which is composed of a label variable environment $\eta_L : var_L \rightarrow Edge_\perp$ and a graph variable environment $\eta_G : var_G \rightarrow (Edge \rightarrow Edge_\perp)$. We define $\mathcal{V}$ as the dynamic evaluation of the query which returns a mapping of all edges of the view graph to their respective source edge or a $\perp$ symbol in case it is a constant edge. This respective source edge could be considered the *equivalence class id* of the view edge at hand.

$$\mathcal{V} : Eval \rightarrow env \rightarrow (Edge \rightarrow Edge_\perp)$$

Let $f_\emptyset : Edge \rightarrow Edge_\perp$ be the partial function with an empty domain and define $f\{x \rightarrow y\}$ as

$$f\{x \rightarrow y\}(a) = \begin{cases} y \text{ if } a = x \\ f(a) \text{ if } a \neq x \text{ and } f \text{ defined for } a \end{cases}$$

Further define a partial function $f_1 \cup f_2$ for two partial functions f_1 and f_2 as

$$(f_1 \cup f_2)(x) = \begin{cases} f_1(x) \text{ if } f_1(x) \text{ defined} \\ f_2(x) \text{ if } f_2(x) \text{ defined but not } f_1(x) \end{cases}$$

The function $\mathsf{aux}(m, \zeta)$ uses $\mathsf{reachable}$ to calculate the edges reachable from the destination of the edge ζ (it may include ζ itself because of cycles). The definition of aux is:

$$\mathsf{aux}(m, \zeta) = \{(\zeta' \mapsto eq) \in m \mid \zeta' \in \mathsf{reachable}(\zeta)\}$$

where eq denotes the equivalence class for edge ζ'.

The wrap function wraps each node ID in the key of the $Edge \rightarrow Edge_\perp$ mapping with the RecE trace information, to make the result of $\mathcal{V}$ consistent with the view graph of the forward transformation.

Then $\mathcal{V}$ is recursively defined according to the structure of the top-level query construct in Fig. 14.

When $\mathcal{V}$ is then applied to the query as a whole and an initial environment which maps $\$db$ to a mapping of each source edge to itself, the output is a mapping of all view edges to their equivalence class. For those equivalence classes with more than one member, there is then a possibility for inconsistent edits. For an equivalence class eq_1, backward transformation will fail if there are two edges that are part of the equivalence class, both have been updated by the edit operation to the view, but the resulting edge labels are different. Notably, backward transformation does not fail in case some edges have been updated and others have not, just as long as the updates all have the same new label. The GUI editor for the view graph can be modified to take these equivalence classes into account.

When applying $\mathcal{V}$ to our example, we get three equivalence classes, one for every source edge. Forward evaluation features one iteration of the outmost **rec** construct which contains two iteration of the inner **rec** construct. That way, the $\$l$ for both view edges with label File are in the equivalence class of source edge File. If the two File edges in the view are modified inconsistently, backward transformation fails.

6 Our GRoundTram System

We have integrated the tracing mechanisms in the form of highlighting and editability support described in this paper into our GRoundTram system. We plan to publish the implementation from our project website at `http://www.prg.nii.ac.jp/projects/gtcontrib/cmpbx/`.

In the following, we briefly summarize these features of GRoundTram from the user's point of view. Fig. 15 shows the screenshot of our GRoundTram system.

When the user clicks on a view edge, there are two possibilities.

1. If the view edge has been copied from the source graph, the system highlights the corresponding edge in the source graph from which the view edge originates. The system also highlights the graph variables in the query that have produced the view edge.
2. If the view edge has been created by an edge constructor in the query, that edge constructor is highlighted. In this case, no edge in the source graph is highlighted. All graph variables that have contributed to creating the view edges are also highlighted.

The highlighting also works in the other direction. When the user clicks on an edge in the source graph, all view edges that are copies of that source edge are highlighted. Additionally, when the user marks an edge constructor or a graph variable in the query, all view edges created by the constructor or variable are highlighted.

Multiple selection of edges in the view and source graph as well as multiple marking of edge constructors and graph variables in the query are also supported.

In the screenshot (Fig. 15), the user selected mutiple view edges at the pane on the right. They are

$$
\begin{array}{ll}
\mathcal{V}\ _G\{\} & \eta = f_\emptyset \\
\mathcal{V}\ _G() & \eta = f_\emptyset \\
\mathcal{V}\ _G\{_L\$l : {}_{G_1}\overline{e}\} & \eta = (\mathcal{V}\ _{G_1}\overline{e}\ \eta)\{(G.I(\&), L, G_1.I(\&)) \mapsto \eta_L(\$l)\} \\
\mathcal{V}\ _G\{a : {}_{G_1}\overline{e}\}\ (a \in \mathit{Label}) & \eta = (\mathcal{V}\ _{G_1}\overline{e}\ \eta)\{(G.I(\&), a, G_1.I(\&)) \mapsto \bot\} \\
\mathcal{V}\ _G({}_{G_1}\overline{e_1} \cup {}_{G_2}\overline{e_2}) & \eta = (\mathcal{V}\ _{G_1}\overline{e_1}\ \eta) \cup (\mathcal{V}\ _{G_2}\overline{e_2}\ \eta) \\
\mathcal{V}\ _G({}_{G_1}\overline{e_1} \oplus {}_{G_2}\overline{e_2}) & \eta = (\mathcal{V}\ _{G_1}\overline{e_1}\ \eta) \cup (\mathcal{V}\ _{G_2}\overline{e_2}\ \eta) \\
\mathcal{V}\ _G(\&m := {}_{G_1}\overline{e}) & \eta = \mathcal{V}\ _{G_1}\overline{e}\ \eta \\
\mathcal{V}\ _G\&m & \eta = f_\emptyset \\
\mathcal{V}\ _G({}_{G_1}\overline{e_1} \mathbin{@} {}_{G_2}\overline{e_2}) & \eta = (\mathcal{V}\ _{G_1}\overline{e_1}\ \eta) \cup (\mathcal{V}\ _{G_2}\overline{e_2}\ \eta) \\
\mathcal{V}\ _G\mathbf{cycle}({}_{G_1}\overline{e}) & \eta = \mathcal{V}\ _{G_1}\overline{e}\ \eta \\
\mathcal{V}\ _G(\overline{\mathbf{if}\ b}\ {}_{G_1}\overline{e}) & \eta = \mathcal{V}\ _{G_1}\overline{e}\ \eta \\
\mathcal{V}\ _G\$g & \eta = \eta_G(\$g) \\
\mathcal{V}\ _G\overline{\mathbf{rec}}(\lambda(\$l, \$g).M)({}_{G_1}\overline{e_\mathrm{a}}) & \eta = \text{let } m = \mathcal{V}\ _{G_1}\overline{e_\mathrm{a}}\ \eta \text{ in } \bigcup_{(\zeta \mapsto {}_{G_b}\overline{e_\mathrm{b}}) \in M} \mathsf{wrap}(\mathcal{V}\ _{G_b}\overline{e_\mathrm{b}}\ \eta'_\zeta) \\
 & \qquad \text{with } \eta'_\zeta = (\eta_L\{\$l \mapsto m(\zeta)\}, \eta_G\{\$g \mapsto \mathsf{aux}(m, \zeta)\}) \\
\mathcal{V}\ _G(\mathbf{let}\ \$g = {}_{G_1}\overline{e_1}\ \mathbf{in}\ {}_{G_2}\overline{e_2}) & \eta = \mathcal{V}\ _{G_2}\overline{e_2}\ \eta' \qquad \text{with } \eta' = (\eta_L, \eta_G\{\$g \mapsto \mathcal{V}\ _{G_1}\overline{e_1}\ \eta\}) \\
\mathcal{V}\ _G(\mathbf{llet}\ \$l = \mathtt{a}\ \mathbf{in}\ {}_{G_1}\overline{e}) & \eta = \mathcal{V}\ _{G_1}\overline{e}\ (\eta_L\{\$l \mapsto \bot\}, \eta_G) \\
\mathcal{V}\ _G(\mathbf{llet}\ \$l = \$l'\ \mathbf{in}\ {}_{G_1}\overline{e}) & \eta = \mathcal{V}\ _{G_1}\overline{e}\ (\eta_L\{\$l \mapsto \eta_L(\$l')\}, \eta_G)
\end{array}
$$

Fig. 14 The algorithm for calculating the equivalence class mappings.

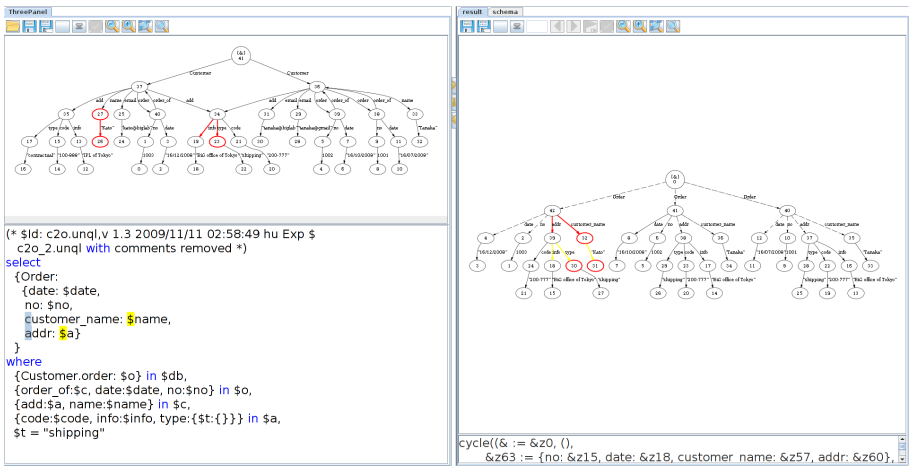

Fig. 15 Screenshot of GRoundTram system showing traces between source graph, UnQL transformation and view graph

rendered in two different colors:

- yellow (`info`,`type`, "`Kato`") if if they are directly copied from the source, as the corresponding edges are highlighted in the left top pane and the graph varaible references that produced these edges are highlighted in yellow in the UnQL transformation in the left bottom pane,
- red (`addr`,`customer_name`) if they are constructed in the transformation, and the corresponding labels (all constants in this case) are highlighted in blue color in the UnQL transformation in the left lower pane.

In addition to highlighting the correspondence between source graph, view graph and query, GRoundTram also renders view edges in a classified manner. In particular, constant edges are shown in a special way that can be recognized easily by the user. Editing these edges is disabled by the system. Moreover, when the user clicks

on a view edge, all view edges that cannot be simultaneously edited in an inconsistent way are highlighted.

These features have proven to be very useful to better comprehend and predict the behavior of bidirectional transformations, especially of those that are non-trivial. Particularly useful is when a transformation query contains duplicate label names or graph variable names, the highlighting clearly shows which name actually creates a selected view edge. The editability allows the user to avoid changes to the view graph that would eventually lead to a failure without waiting until the backward transformation is executed.

7 Related Work

The notion of traces has been extensively studied in a more general context of computations, like provenance traces [4] for the nested relational calculus, as the importance of traceability is widely recognized as the key to know the origin, ownership, or history of an object. One of the latest related works that shares our motivation in the model driven engineering community is the work by Van Amstel et al. who proposed a visualization framework for chains of ATL model transformations [20]. Although their work supports unidirectional transformations, they also consider it important not only to trace from source to target but the transformation query should also be involved. Systematic augmentation of the trace-generating capability with model transformations is achieved by higher-order model transformations [19]. We have already introduced the trace generation mechanism in our previous work [8] but the main objective was to the bidirectionalization itself.

Generation of tracing information is also studied in the reversible computation [22] community where minimization of the trace information is also seriously considered as in [23]. Currently the size of the trace information in our GRoundTram for the proposed framework to work is prohibitive in terms of performance even for middle scale examples. In the normal mode of GRoundTram we internally compress the trace information and it is part of our future work to make the proposed tracing work under the compression as well.

Tracing between a surface language (syntactic sugar) and its core language, like between UnQL and UnCAL in our work, is not easy in general due to the big gap between the two languages. Pombrio and Krishnamurthi [17] tackle this gap by proposing an approach to automatically reproducing an evaluation sequence in the core language in the surface language. This work may provide a partial solution for traceability between UnQL and UnCAL in our graph transformation settings.

Triple Graph Grammars (TGG) [18] and frameworks based on it are studied extensively by the graph transformation research community and applied to model-driven engineering. It is based on graph rewriting rules that consist of triples of source graph pattern, target graph pattern, and the correspondence graph in-between. The triple of source graphs, correspondence graphs and target graphs grow in parallel in the graph grammars described by the rewriting rules. The trace information is explicit in the link between correspondence graphs and source/target graphs.

As another well-studied bidirectional transformation framework, semantic bidirectionalization [21] makes use of the polymorphism of the forward transformation to derive the backward transformation. It generates the table of correspondence between elements in the source and those in the target to guide the reflection of updates, and does not even require inspection of the syntax of the forward transformation at all (thus called semantic bidirectionalization). Matsuda and Wang [16] addressed the problem of the inability of branching by comparison with constant values that breaks the polymorphism, by run-time recording of the branching behaviors. They also cope with data constructed during transformation. Ours framework is close to both of the above ([21] and [16]) frameworks in the sense that they also utilize a particular run of a forward transformation. It is also worth noting that UnCAL transformation is also monomorphic in general because of the label comparison in the **if** conditionals. Our bottom equivalence class corresponds to Matsuda and Wang's location identifier #, and our ordinary equivalence classes. correspond to their ordinary numerical location identifiers.

Since the semantic bidirectionalization does not have to walk through the backward transformation syntax, what we are trying to do in our present paper before execution is very close to what they

are doing in their backward transformation. For example, they are able to detect inconsistent updates based on the location identifiers, and reject the attempt to update data created during transformation by finding the attempt to update the data on a location identified by #.

In the (purely) semantic bidirectionalization framework, the source-target trace is generated by just running the transformation giving "artificial" input in which every occurrence of an element is replaced by a unique identifier. This is possible because polymorphic transformation does not inspect the concrete element values (labels in our case). But as Matsuda and Wang addressed, the inspection of labels breaks the framework. So Matsuda and Wang instead make use of the combination of location identifiers and the actual input value. We instead use input edges as the combination of input and location identifier because input edges include not only the label value but also node identifiers on both ends. With respect to run-time recording, Matsuda and Wang have only to record the branching behaviors that inspected element values. This is implemented by seamlessly wrapping the comparison operation that takes care of that recording. Instead we currently record all the intermediate results.

8 Conclusion

Currently bidirectional transformations sometimes receive a reputation that the result of backward transformation is difficult to understand or predict. This applies to a framework like our GRoundTram system in which the backward semantics of the given forward transformation is completely provided, rather than (partially) left to the users, so the logic of backward transformation is rather fixed and hidden by the semantics. Once the transformation gets more complex, the prediction is even more difficult.

In this paper, we proposed, within a compositional bidirectional graph transformation framework based on structural recursion, a comprehensive externalization of tracing between source graphs, transformation and view graphs in our system GRoundTram. The intermediate results kept during forward transformations as well as trace information used for backward transformation, were utilized for the tracing.

We also classify every edge in the target graph so that users can know the attempts to edit every edge that is classified as constant edge lead to failure of backward transformation, and for other edges, the inconsistent updates for the edges classified identically will also cause the failure of backward transformation. Since the language we deal with allows arbitrary variable references that causes copies, identification of copies in the view had not been that trivial, but our proposal is simple as it just maintains the mapping from source edge to the view edge per variables and updates the mappings for each variable binding construct.

We are now improving our implementation and plan to distribute it in our project website.

As our future work, we have found some potential in the classification framework in that it can detect the branching behavior change that current GRoundTram fails to detect. Unlike point-free approaches, we can always refer to non-local variables in the transformation, which means that updates of the binding during backward transformation may cause side effect indirectly to the value referred by other variables. We would like to incorporate this fix in the next release of GRoundTram. Another potential is the capability of tracing the occurrence position of graph variables as well as constants by extending the entry of the table maintained by the classifier to store the code positions and graph variables. We would like to investigate this direction towards unifying tracing with classification. We also plan to overcome the limitation of tracing with optimization activated mentioned in Section 4. 2. 3 by defining rules of how to pass position information of edge constructors and graph variables in an UnCAL expression to the corresponding elements in the optimized one. Our partial implementation in this direction shows promising results.

We would like to investigate the possibility of accommodating update operations other than edge renaming, like insertion of subgraphs, using the same backward transformation semantics, because we currently handle insertions using separate general inversion strategy which is costly. We currently have limited support, however we do not have bidirectional property for complex expressions. One obvious case that we can support is subgraph extraction like **select** $\{a : \$g\}$ **where** $\{a : \$g\}$ **in** $\$db$

in which we could insert an arbitrary subgraph below the top level edge labeled `a`, because the subgraph is not "observed" in the transformation so update will never interfere with the branching behavior. Even though that part is "observed" by the bulk semantics, that part is left unreachable, so update does not affect the computation of the reachable part. If we extend the classifier function to indicate which edge involved in the branching behavior, inspired by [16], we could safely determine the part that accept the insertion or deletion of that part reusing the in-place update semantics, thus achieving "cheap backward transformation".

We have not discussed the performance aspect in the paper. The trace based on algebraic data constructors increases the size of the trace dramatically as the composition of structural recursion increases. It prohibits the execution of even middle-scale examples to work. We plan to compress the trace information to restore scalability.

Acknowledgement The authors would like to thank Zhenjiang Hu and Hiroyuki Kato for their valuable comments, and Kazutaka Matsuda for discussing the branching behavior change detection. The project was supported by the International Internship Program of the National Institute of Informatics.

References

[1] Bancilhon, F. and Spyratos, N.: Update Semantics of Relational Views, *ACM Trans. Database Syst.*, Vol. 6, No. 4(1981), pp. 557–575.

[2] Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., and Schmitt, A.: Boomerang: Resourceful lenses for string data, *POPL '08*, 2008, pp. 407–419.

[3] Buneman, P., Fernandez, M., and Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion, *The VLDB Journal*, Vol. 9, No. 1(2000), pp. 76–110.

[4] Cheney, J., Acar, U. A., and Ahmed, A.: Provenance Traces, *CoRR*, Vol. abs/0812.0564(2008).

[5] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F.: Bidirectional Transformations: A Cross-Discipline Perspective, *ICMT*, Paige, R. F.(ed.), Lecture Notes in Computer Science, Vol. 5563, Springer, 2009, pp. 260–283.

[6] Dayal, U. and Bernstein, P. A.: On the Correct Translation of Update Operations on Relational Views, *ACM Trans. Database Syst.*, Vol. 7, No. 3(1982), pp. 381–416.

[7] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, *ACM Trans. Program. Lang. Syst.*, Vol. 29, No. 3(2007).

[8] Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., and Nakano, K.: Bidirectionalizing graph transformations, *ICFP'10*, 2010, pp. 205–216.

[9] Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K., and Sasano, I.: Marker-directed optimization of UnCAL graph transformations, *Logic-Based Program Synthesis and Transformation, 21st International Symposium, LOPSTR 2011, Odense, Denmark, Revised Selected Papers, Lecture Notes in Computer Science*, Vol. 7225, July 2012, pp. 123–138.

[10] Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K.: GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations (short paper), *26th IEEE/ACM International Conference On Automated Software Engineering*, IEEE, 2011, pp. 480–483.

[11] Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K.: GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations, *Progress in Informatics*, No. 10(2013), pp. 131–148.

[12] Hidaka, S., Hu, Z., Kato, H., and Nakano, K.: Towards Compostional Approach to Model Transformations for Software Development, Technical Report GRACE-TR08-01, GRACE Center, National Institute of Informatics, August 2008.

[13] Hidaka, S., Hu, Z., Kato, H., and Nakano, K.: Towards a compositional approach to model transformation for software development, *SAC'09: Proceedings of the 2009 ACM symposium on Applied Computing*, New York, NY, USA, ACM, 2009, pp. 468–475.

[14] Hidaka, S. and Terwilliger, J. F.: Preface to the Third International Workshop on Bidirectional Transformations, *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014) (EDBT/ICDT)*, Candan, K. S., Amer-Yahia, S., Schweikardt, N., Christophides, V., and Leroy, V.(eds.), CEUR Workshop Proceedings, No. 1133, Aachen, 2014, pp. 61–62.

[15] Hu, Z., Iwasaki, H., Takeichi, M., and Takano, A.: Tupling Calculation Eliminates Multiple Data Traversals, *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, Amsterdam, The Netherlands, ACM Press, June 1997, pp. 164–175.

[16] Matsuda, K. and Wang, M.: Bidirectionalization for Free with Runtime Recording: Or, a Lightweight Approach to the View-update Problem, *Proceedings of the 15th Symposium on Principles and*

Practice of Declarative Programming, PPDP '13, New York, NY, USA, ACM, 2013, pp. 297–308.

[17] Pombrio, J. and Krishnamurthi, S.: Resugaring: Lifting Evaluation Sequences Through Syntactic Sugar, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, New York, NY, USA, ACM, 2014, pp. 361–371.

[18] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany*, Mayr, E. W., Schmidt, G., and Tinhofer, G.(eds.), Lecture Notes in Computer Science, Vol. 903, Springer, June 1995, pp. 151–163.

[19] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J.: On the Use of Higher-Order Model Transformations, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, Springer-Verlag, 2009, pp. 18–33.

[20] van Amstel, M. F., van den Brand, M. G. J., and Serebrenik, A.: Traceability Visualization in Model Transformations with Tracevis, *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations*, ICMT'12, Berlin, Heidelberg, Springer-Verlag, 2012, pp. 152–159.

[21] Voigtländer, J.: Bidirectionalization for Free! (Pearl), *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, New York, NY, USA, ACM, 2009, pp. 165–176.

[22] Yokoyama, T.: Reversible Computation and Reversible Programming Languages, *Electron. Notes Theor. Comput. Sci.*, Vol. 253, No. 6(2010), pp. 71–81.

[23] Yokoyama, T., Axelsen, H. B., and Gluck, R.: Minimizing Garbage Size by Generating Reversible Simulations, *Proceedings of the 2012 Third International Conference on Networking and Computing*, ICNC '12, Washington, DC, USA, IEEE Computer Society, 2012, pp. 379–387.