ISSN 1884-0760

GRACE TECHNICAL REPORTS

A Trace-based Approach to Increased Comprehensibility and Predictability of Bidirectional Graph Transformations

Soichiro Hidaka Martin Billes Quang Minh Tran

GRACE-TR 2015–03 February 2015

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

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

A Trace-based Approach to Increased Comprehensibility and Predictability of Bidirectional Graph Transformations*

Soichiro Hidaka[1], Martin Billes[2], Quang Minh Tran[3]
[1]National Institute of Informatics, Japan
Email: hidaka@nii.ac.jp
[2]Augsburg University, Germany
Email: martin.billes@student.uni-augsburg.de
[3]Daimler Center for IT Innovations, Technical University of Berlin, Germany
Email: quang.tranminh@dcaiti.com

September 2014

***Abstract*—Bidirectional graph transformation is expected to play an important role in model-driven software engineering, where artifacts are often refined through compositions of model transformations, from high level specifications down to concrete ones closer to implementation. In such a setting, changes in the artifacts are reflected not only from upstream to downstream, but also the other way round.**

However, it is often difficult to understand the correlation between the transformed artifacts. It is equally hard to predict to which part a change will eventually be propagated, and whether the propagation will succeed at all. That comprehensibility and predictability is crucial for more complex transformations.

In this paper, we propose a well-defined tracing mechanism between source, transformation and target, as well as edge classification mechanism on the target artifacts, in a compositional framework of bidirectional graph transformation.

We implemented these mechanisms in our GUI so that users are informed if an edit on the target is propagable to the source, where to be propagated, if the edit affects other parts in the target, or where in the transformation should be changed to accommodate the edit, without executing a backward transformation. We demonstrate the effectiveness of our approach using non-trivial bidirectional graph transformations.

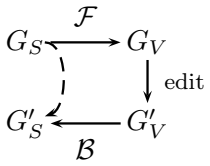
I. Introduction

Models used for Model-Driven Software Engineering can become so large that they become difficult to comprehend for developers. For better comprehension, a view of the model can be used to highlight certain interesting aspects. A view of a model does not have to serve simplification alone: it can also provide a different perspective, such as 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 by bidirectional transformations [1]. This problem is widely studied in the database community [2, 3] and more recently in the programming language community [4, 5].

*IEEE conference style file is used to format this paper.

In software engineering, bidirectional transformation is expected to play an important role in software development [6] such as model synchronization [7, 8, 9], round-trip engineering [10], multiple-view software development [11, 12], and model-code coevolution [13]. Transformations serve to manipulate and generate models and other artifacts. Changes downstream the chain of transformations are expected to be reflected upstream so that, for example, a defect found and fixed downstream can be propagated to the upstream to avoid reproducing the defect.

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



The forward transformation may discard some of the information found in the source and create some new information on its own. It can also 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, since we consider model transformation our primary application and models are represented by graphs [15, 16]. Our bidirectional graph transformation [17] is based on bidirectional interpretation of an existing unidirectional language based on structural recursion on unstructured data [18]. The bulk semantics of structural recursion allowing per-edge independent computation is utilized in the bidirectionalization.

To be useful, bidirectional transformation needs round-trip properties [1, 6]. In our setting, forward transformation $\mathcal{F}[\![e]\!]\rho$ computes the transformation expression e for variable binding ρ where ρ corresponds to the source, and at the start it includes the global variable binding the input graph. The forward transformation outputs the view graph G and backward transformation $\rho' = \mathcal{B}[\![e]\!](\rho, G')$ produces the updated source

ρ' given the updated view graph G' and the original source ρ. Our bidirectional transformation must satisfy the GetPut and WPutGet properties [17], 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 when the view is not updated after forward transformation, the result of the following backward transformation agrees with the original source, and W(Weak)PutGet (a.k.a. *weak invertibility* [19], a weaker notion of PutGet [4] or Correctness [6] or Consistency [2] because of the rather arbitrary variable reference allowed in our language) says that ρ' is returned after being fed into another round-trip without edit operations.

When looking at an edge in the view graph, it is not immediately apparent whether it has its origin in a certain part of the source graph or of the transformation and if yes, which part that is exactly. The exact location where edits will be propagated to is not evident either, which can cause unpredictable results. Backward transformation in particular can fail, which happens if 1) the label of the edited view edge appears as a constant of the transformation, (2) a group of view edges are edited inconsistently or (3) edits of view edges lead to changes in branching behavior in the transformation. If a lot of edits are made at once, it becomes increasingly difficult to predict whether backward transformation will succeed or not.

For better comprehension and prediction of bidirectional transformation, we have successfully integrated trace-based highlighting support into our bidirectional transformation system named GRoundTram (Graph Roundtrip Transformation for Models) [20, 15, 16] (see Fig. 10). In GRoundTram, when the user selects a view edge on the right panel, the system highlights the corresponding source edge if there is one. The edge constructor or graph variable in the transformation that has produced the view edge is highlighted in blue or yellow, respectively. The edges created by constant labels in the transformation are drawn in dashed lines and GRoundTram disables editing them. Groups of view edges that cannot be inconsistently modified to different label names are highlighted in green.

There is some existing work with similar objectives. For tracing, Van Amstel et al. [21] proposed the visualization of traces, but in the unidirectional model transformation setting. For classification of elements in the view, Matsuda and Wang's work [22] in the context of extension of semantic approach [23] to general bidirectionalization, is also capable of similar classification, while we reserve opportunities to recommend variety of consistent changes for more complex branching conditions.

The rest of the paper is organized as follows: Section II summarizes the semantics of our underlying graph data model, core graph language UnCAL and the user-level syntax UnQL.

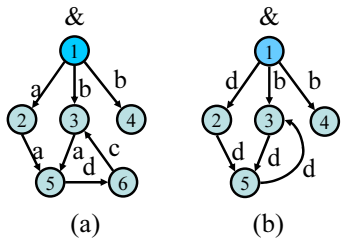


Fig. 1. Cyclic graph examples

Readers already familiar with them can safely skip the first two subsections. Section III motivates our work by an example, which is also used in the following sections. The simplicity of the example is for the sake of explanation, and we have more involved examples related to software engieering in our project website mentioned in Section VI. Section IV proposes a tracing mechanism that supports the highlighting of the correspondence between source, transformation and target. Section V 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 VI describes how the proposed mechanisms in the preceding sections are integrated in GRoundTram. Section VII discusses related work, and Section VIII concludes with future work.

II. Preliminaries

We use the UnCAL (Unstructured CALculus) query language [18]. UnCAL has an SQL-like syntactic sugar called UnQL (Unstructured Query Language) [18]. Bidirectional execution of graph transformation in UnQL is achieved by desugaring the transformation into UnCAL and then bidirectionally interpreting it [17]. This section explains the graph data model we use, the UnCAL and UnQL languages as well as an extension of UnCAL forward transformation.

A. UnCAL

Our graphs are multi-rooted and edge-labeled with all information stored in edge labels ranging over $Label \cup \{\varepsilon\}(Label_\varepsilon)$, the node labels being arbitrary. There is no order between outgoing edges of nodes. The notion of graph equivalence is based on bisimulation, so equivalence between the graphs is efficiently determined [18], and we can always normalize [16] up to isomorphism.

Fig. 1 shows examples of our graphs.

We represent a graph by a quadruple (V, E, I, O). V is the set of nodes, E the set of edges ranging over the set $Edge_\varepsilon$, where an edge is represented by a triple of source node, label and destination node. I is the function identifying the root or input nodes of the graph by the input markers denoted like $\&x$ that range over the set $Marker$. A special marker $\&$ is called the default marker.

Apart from the roots as "entry points" of graphs, a graph may have "exit points" as represented by O which is a relation $V \times Marker$ and $(v, \&m) \in O$ implies the node v is associated with output marker $\&m$. Such nodes are called output nodes.

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

Fig. 2. Core UnCAL Language

For example, the graph in Fig. 1 (a) is represented by (V, E, I, O), where $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, \mathtt{a}, 2), (1, \mathtt{b}, 3), (1, \mathtt{b}, 4), (2, \mathtt{a}, 5), (3, \mathtt{a}, 5), (5, \mathtt{d}, 6), (6, \mathtt{c}, 3)\}$, $I = \{\& \mapsto 1\}$, and $O = \{\}$. This graph has no output node.

Each component of the quadruple is denoted by the "." syntax, such as $g.\mathrm{V}$ for graph g. The type of the graphs is represented by $DB^{\mathcal{X}}_{\mathcal{Y}}$, where $\mathcal{X}$ is the set of input markers or domain of I and $\mathcal{Y}$ is the set of output markers. The graph in the above example has type $DB^{\&}_{\emptyset}$. The superscript is omitted, if the set is $\{\&\}$, and the subscript likewise, if the set is empty. The type of the example graph is simply denoted by DB, and the (static) type of UnCAL expression e by $e :: DB^{\mathcal{X}}_{\mathcal{Y}}$.

In UnCAL, we have nine graph constructors (Fig. 2) whose semantics is illustrated in Fig. 3.

We use hooked arrows ($\hookrightarrow$) stacked with the constructor to denote the computation by the constructors where the left-hand side is the operand(s) and the right-hand side is the result. There are three nullary constructors. $()$ constructs a graph without any nodes or edges, so $\mathcal{F}[\![()]\!] \in DB^{\emptyset}$. The constructor $\{\}$ 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., $\mathcal{F}[\![\&y]\!] \in DB_{\{\&y\}}$.

The edge constructor $\{\_ : \_\}$ takes a label l and a graph $g \in DB_{\mathcal{Y}}$, constructs a new root with the default input marker and extends an edge labeled l from the new root to $g.\mathrm{I}(\&)$, thus $\{l : g\} \in DB_{\mathcal{Y}}$. For the union $g_1 \cup g_2$ of 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 $\mathcal{X} = \{\&x_1, \ldots, \&x_m\}$, m new input nodes for each $\&x_i \in \mathcal{X}$ are constructed, and from each of these nodes, two ε-edges are extended to $g_1.\mathrm{I}(\&x_i)$ and $g_2.\mathrm{I}(\&x_i)$. Here, ε-edges are similar to ε-transitions in automata and used to connect components during the graph construction. Clearly, $g_1 \cup g_2 \in DB^{\mathcal{X}}_{\mathcal{Y}_1 \cup \mathcal{Y}_2}$.

The input node renaming operator $:=$ takes a marker $\&x$ and a graph $g \in DB^{\mathcal{Y}}_{\mathcal{Z}}$ with $\mathcal{Y} = \{\&y_1, \ldots, \&y_m\}$, and returns a graph whose input markers are prepended by $\&x$, thus $(\&x := g) \in DB^{\&x.\mathcal{Y}}_{\mathcal{Z}}$ where the dot "." concatenates markers and forms a monoid with $\&$, 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 assigns a new name to the root of the operand, i.e., $(\&x := g) \in DB^{\{\&x\}}_{\mathcal{Y}}$ 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 $\mathcal{X} \cap \mathcal{Y} = \emptyset$, the resultant graph inherits all the markers, edges and nodes from the operands, thus $g_1 \oplus g_2 \in DB^{\mathcal{X} \cup \mathcal{Y}}_{\mathcal{X}' \cup \mathcal{Y}'}$.

The remaining two constructors connect output and input nodes with matching markers by ε-edges. $g_1 \mathbin{@} g_2$ appends $g_1 \in DB^{\mathcal{X}}_{\mathcal{X}' \cup \mathcal{Z}}$ and $g_2 \in DB^{\mathcal{X}' \cup \mathcal{Z}'}_{\mathcal{Y}}$ by connecting the output and input nodes with a matching subset of markers $\mathcal{X}'$, and discards the rest of the markers, thus $g_1 @ g_2 \in DB^{\mathcal{X}}_{\mathcal{Y}}$. An idiom $\&x'@g_2$ projects (selects) one input marker $\&x'$ and rename it to default ($\&$), while discarding the rest of the input markers (making them unreachable). $\mathbf{cycle}(g)$ for $g \in DB^{\mathcal{X}}_{\mathcal{X} \cup \mathcal{Y}}$ with $\mathcal{X} \cap \mathcal{Y} = \emptyset$ works similarly to $@$ but in an intra-graph instead of inter-graph manner, by connecting output and input nodes of g with matching markers $\mathcal{X}$, and constructs copies of input nodes of g, each connected with the original input node by an ε-edge. 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). Here, the notion of bisimilarity is extended to cope with ε-edges.

The semantics of conditionals is standard, but the condition is restricted to label equivalence comparison. There are two kinds of variables: label variables and graph variables. Label variables, denoted $\$l, \$l_1$ etc., bind labels while graph variables denoted $\$g, \$g_1$ etc., bind graphs. They are introduced by structural recursion operator $\mathbf{rec}$.

We take a look at the following concrete transformation in UnCAL that replaces every label `a` by `d` and removes edges labeled `c`.

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

If the graph variable $\$db$ is bound to the graph in Fig. 1 (a), the result of the transformation will be the one in Fig. 1 (b). We call the first operand of $\mathbf{rec}$ the *body* expression and the second operand the *argument* expression. In the above transformation, the body is an $\mathbf{if}$ conditional, while the argument is the variable reference $\$db$. We use $\$db$ as a special 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 superscribe UnCAL expressions with their code position $p \in \mathit{Pos}$ where Pos is the set of position numbers. For instance, in the example above, the numbers 1 and 2 in $\{d : \&^1\}^2$ denote the code positions of the graph constructors $\&$ and $\{d : \&\}$, respectively.

Fig. 4 shows the *bulk* semantics of $\mathbf{rec}$ for the example. It is "bulk" because the body of $\mathbf{rec}$ can be evaluated in parallel for each edge and the subgraph reachable from the target node of the edge (which are correspondingly bound to variables $\$l$ and $\$g$ in the body).

In the bulk semantics, the node identifier carries some information which has the following structure $\mathit{TraceID}$:

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

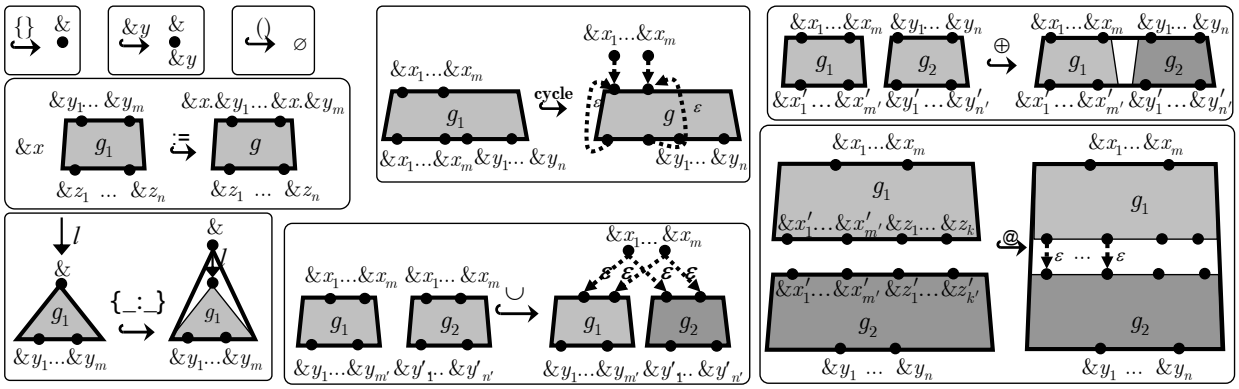


Fig. 3. Graph Constructors of UnCAL

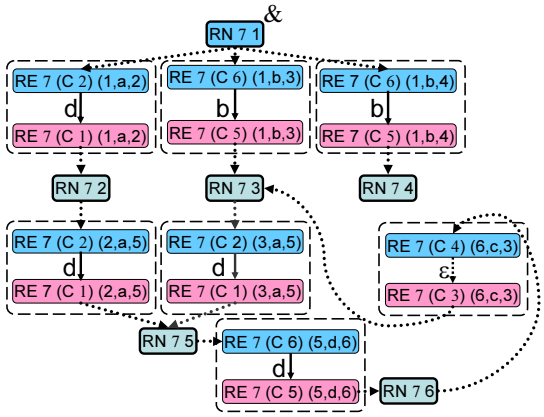


Fig. 4. Bulk semantics by example

(template)	T	::=	$\{L:T,\ldots,L:T\} \mid T \cup T$
			$\mid \$g \mid \textbf{if}\ BC\ \textbf{then}\ T\ \textbf{else}\ T$
			$\mid \textbf{select}\ T\ \textbf{where}\ B,\ldots,B$
			$\mid \textbf{letrec sfun}\ fname(L:\$G)$
			$= \ldots \textbf{in}\ fname(T)$
(binding)	B	::=	$Gp\ \textbf{in}\ \$G \mid BC$
(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 pat.)	Rp	::=	a $\mid$ _ $\mid Rp.Rp \mid (Rp\|Rp)$
			$\mid Rp? \mid Rp* \mid Rp+$

Fig. 5. Syntax of UnQL

where the base case ($SrcID$) 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 of the corresponding input node of the operand(s) of the constructor. Except for $\cup$, the marker is always default and thus omitted. RecN $p\ v\ \&z$ denotes the node created by **rec** at position p for node v of the graph resulting from evaluating the argument expression. For example, in Fig. 4, the node RN 7 1, originating from node 1, is created by **rec** at position 7 (RecN is abbreviated to RN in the figure for simplicity, and similarly Code to C and RecE to RE). We have six such nodes, one for each 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 $(\{(\mathsf{C}\ 2), (\mathsf{C}\ 1)\}, \{(\mathsf{C}\ 2, d, \mathsf{C}\ 1)\}, \{\& \mapsto \mathsf{C}\ 2\}, \{(\mathsf{C}\ 2, \&)\})$, with the nodes C 2 and C 1 constructed by $\{\_:\_\}$ and $\&$, respectively. For the shortcut edges, an ε-edge is generated similarly. Then each node v of such results for edge ζ is wrapped with the trace information RE like RE $p\ v\ \zeta$ for **rec** at position p. These results are surrounded by round squares drawn with dashed lines in Fig. 4. They are then connected together according to the original shape of the graph as depicted in Fig. 4. 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 IDs, we obtain the result graph in Fig. 1 (b).

The variable binders **let** and **llet** having standard meanings are our extensions used for optimization by rewriting [24].

In the backward evaluation of **rec**, ε-elimination process is reversed to restore the shape of Fig. 4, and then the graph is decomposed with the help of the structures of trace IDs, and then the decomposed graph is used for the backward evaluation of each body expression. The backward evaluation produces the updated variable bindings (in this body expression we get the bindings for $\$l$, $\$g$ and $\$db$ and merge them to get the final binding of $\$db$). For example, the update of the edge label of $(1, \mathtt{b}, 3)$ in the view to x is propagated via the backward evaluation of the body $\{\$l : \&\}$, which produces the binding of $\$l$ updated with x and is reflected to the source graph with edge $(1, \mathtt{b}, 3)$, replaced by $(1, \mathtt{x}, 3)$.

B. UnQL as a Textual Surface Syntax of Bidirectional Graph Transformation

We use the surface language UnQL [18] (Fig. 5) for bidirectional graph transformation. An UnQL expression can be translated into UnCAL, a process referred to as desugaring. We highlight the essential part of the translation in the following. Please refer to [18, 25, 26] for details.

The template (directly after 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 corresponding references are translated into combinations of graph variable bindings in nested **rec**s as well as references to them in the body of **rec**s. The following example translates an UnQL expression into an equivalent UnCAL one.

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

$\Rightarrow$

rec($\lambda(\$l,\$g)$. **if** $\$l$ = a
then rec($\lambda(\$l',\$g)$. **if** $\$l'$ = b
then {res:$\$db$}
else {})($\$db$)
else {})($\$db$).

C. Forward Semantics with Traceable View And Intermediate Results

In the extended forward evaluation of UnCAL we additionally track every intermediate result of the operands. These results are represented by the subscripts on the left of expressions, and denoted by $\overline{e}$, ranging over the set $Eval$. The subscripts and/or the overline are only used when necessary. For example, for the expression $\{e_1 : e\}$, we have

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

The special cases are the conditional (**if**) and **rec**. The former, denoted by $\overline{\textbf{if}}\ \overline{{}_b e_b}\ \overline{e}$, only keeps the branch taken, where $b : Bool$ records the condition (true if the **then** branch was taken), and $\overline{e}$ is the taken branch. The latter, denoted by $\overline{\textbf{rec}}(\lambda(\$l, \$g).M)(\overline{e_a})$, keeps the map M from the edge, which is the source of both the bindings of label and graph variable, to the result of the extended forward evaluation for these bindings.

III. Motivating Example

An example should clarify our motivation by illustrating how a bidirectional transformation can be difficult to comprehend and predict and by showing how we propose to improve this situation. We use the transformation shown below and the source graph in Fig. 6 to produce the view graph in Fig. 7. Basically, the transformation shows the countries in Europe along with their languages and ethnic groups in the view graph.

```
select {result: {ethnic: $e, language: $lang, located: $cont} }
where {country:
          {name:$g, people: {ethnicGroup: $e},
           language: $lang, continent: $cont}} in $db,
      {$l:$Any} in $cont,
      $l = Europe
```

Listing 1. Transformation in UnQL

In the view graph (Fig. 7), three edges have identical labels "German" $(3, \text{German}, 1)$, $(4, \text{German}, 2)$ and $(12, \text{German}, 11)$, but have different origins in the source graph and are produced by different parts in the transformation. The user may want to comprehend how one of the edges, say $\zeta = (3, \text{German}, 1)$, is related to the source graph and the transformation. Analyzing the transformation, ζ is the language of Germany and is a copy of the edge $(1, \text{German}, 0)$ of the source graph (Fig. 6). On the other hand, ζ has nothing to do with the edge $(11, \text{German}, 10)$ of the source graph despite identical labels. This later edge denotes the ethnic group instead. In addition, ζ is copied by the graph variable $\$lang$ in the **select** part of the transformation. Other graph variables and edge constructors do not participate in creating ζ. It would be tedious and difficult to track those kinds of correspondence in a highly complex transformation with numerous shared edges and duplicate labels. We believe that bidirectional transformation systems should visually highlight corresponding elements between source graph, view graph and transformation to increase comprehensibility.

In this example, the non-leaf edges of the view graph ("result", "located", "language" and "ethnic") are constant edges in the sense that they cannot be modified by the user. The user may not know this and try to rename, say, "located" to "location", only to find that such modification produces an error. Ideally, the system should make such such constant edges easily recognizable in order to prevent accidental modification in the first place.

In another scenario, the user decides that the language of Germany should better be called "German (Germany)" and the language of Austria be called "Austrian German" and thus rename the view edges $(3, \text{German}, 1)$ and $(4, \text{German}, 2)$ to $(3, \text{German (Germany)}, 1)$ and $(4, \text{Austrian German}, 2)$ accordingly. However, the backward transformation fails. The reason is that the language "German" is stored in a single edge in the source graph, but appears twice in the view graph. We get a binding conflict for the edge in the source graph, since both edits try to override the shared language name edge. The user will probably not realize this until the backward transformation fails. Ideally, the system would highlight groups of edges that cannot be renamed without causing inconsistencies and errors. Furthermore, the system would prohibit the triggering of the backward transformation in that case.

Finally, we want to take a look at the case, when one of the edges labeled "Europe" in the view graph is edited to some other value like "Eurasia". Since the transformation depends on the value of this edge, changing it would lead to the selection of another conditional path in the transformation in a subsequent forward transformation. In particular, renaming "Europe" to "Eurasia" will cause an empty view after a round-trip of backward and forward transformation. To prevent this, branch-changing edits are not permitted at all in our system. Changes in branch behavior are very difficult or impossible to predict, if the transformation is too complex. We can assist the user in predicting possible changes in branching behavior in case the label of a view edge should be changed, by highlighting the conditional branches involved. We can even make all branch-changing edits impossible.

IV. Tracing Mechanisms

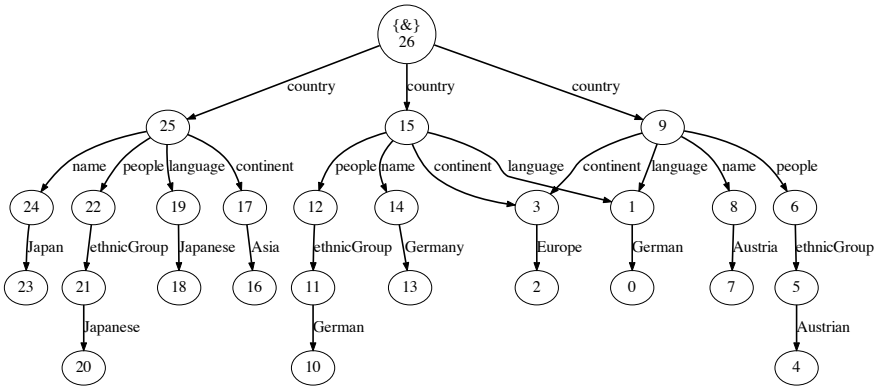


Fig. 6. Example source graph

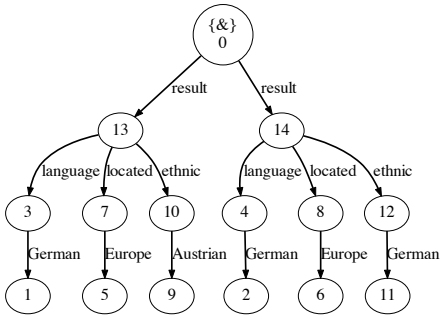
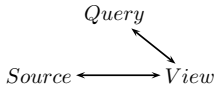


Fig. 7. View graph generated by transformation of the graph in Fig. 6

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 feasible combinations (see to the right).



Even though the tracing mechanisms introduced here are defined for UnCAL, they also work straightforwardly for UnQL, based on the following observation: When an UnQL query is translated into UnCAL, all edge constructors and graph variables in the UnQL query creating edges in the view graph are preserved in the UnCAL query. For instance, the edge constructor $\{\texttt{language} : \$lang\}$ and the graph variable $\$lang$ of the UnQL query in Listing 1 are transferred to the generated UnCAL query in Listing 2.

```
rec(λ ($L,$fv1). if $L = country
  then rec(λ ($L,$g). if $L = name
    then rec(λ ($L,$fv2). if $L = people
      then rec(λ ($L,$e). if $L = ethnicGroup
        then rec(λ ($L,$lang). if $L = language
          then rec(λ ($L,$cont). if $L = continent
            then rec(λ ($l,$Any). if $l = Europe
              then {result: {ethnic: $e,
                             language: $lang,
                             located: $cont}}
              else {})($cont)
            else {})($fv1)
          else {})($fv1)
        else {})($fv2)
      else {})($fv1)
    else {})($fv1)
  else {})($db)
```

Listing 2. UnCAL expression of UnQL query in Listing 1

One limitation is: in our system, the bidirectional interpreter of UnCAL optionally rewrites expressions for efficiency. However, due to excessive reorganization of expressions during the rewriting, we currently support neither tracing UnCAL nor tracing UnQL if the rewriting is activated.

In GRoundTram, when a view edge is selected, the corresponding source edge or query code position is highlighted, and vice versa.

A. Definition of Applied Edge and Origin Edge

In this section, we introduce two definitions that are later used when discussing the tracing mechanisms: applied edge and origin edge. As can be derived from Section II-A, a view edge ζ from the source graph must have a trace ID of the form (n_1, a, n_2), where n_1 and n_2 are $SrcID$s. If the edge is created inside the body of a structural recursion at code position p, it will have the form $(\mathsf{RecE}\, p\ u\ \zeta', a, \mathsf{RecE}\, p\ v\ \zeta')$. We call ζ' an *applied edge* in the sense that it is applied by the structural recursion. Also recall that nested **rec**- constructs produce nested RecE traceable view wrapping around ζ.

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. Moreover, the edge wrapped in the innermost RecE is called the *origin edge* of ζ. The origin edge of a view ζ can be understood as the origin, from which ζ comes from (hence the name).

Formally, the following function, when applied to a view edge ζ, returns a pair whose first and second components are the sequence of applied edges and origin edge of ζ, respectively.

$$\mathsf{tr\_eval} : Edge \to ([Edge], Edge)$$
$$\mathsf{tr\_eval}(((\mathsf{RecE}\, p\ u\ \zeta'), a, (\mathsf{RecE}\, p\ v\ \zeta'))) =$$
$$\quad \mathsf{let}\ (aes, oe) = \mathsf{tr\_eval}((u, a, v))\ \mathsf{in}\ (\zeta' : aes, oe)$$
$$\mathsf{tr\_eval}(\zeta) = ([], \zeta)\ \text{otherwise}$$

where $\zeta' : aes$ prepends ζ' to aes. Additionally, we denote $\mathsf{tr\_applEdgs} = \pi_1 \circ \mathsf{tr\_eval}$ and $\mathsf{tr\_originEdg} = \pi_2 \circ \mathsf{tr\_eval}$ the functions that return the sequence of applied edges and the origin edge of a given view edge ζ. Here, π_i is a projection function on the i-th component of a tuple.

B. Tracing Between Source And View

Given a view edge ζ, if $\zeta' = \mathsf{tr\_originEdg}(\zeta)$ has the form (n_1, a, n_2) where n_1 and n_2 are $SrcID$, then ζ is a copy of ζ' in the source graph. In that case, we denote such source edge ζ' of ζ as $\mathsf{tr\_srcEdg}(\zeta) = \mathsf{tr\_originEdg}(\zeta)$. Otherwise, $\mathsf{tr\_srcEdg}(\zeta)$ fails.

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:

$$\mathsf{tr\_viewEdg}(\zeta_s) = \{\zeta_t \mid \zeta_t \in E_V, \zeta_s = \mathsf{tr\_srcEdg}(\zeta_t)\},$$

where E_V is the set of view edges. For example, the application of tr_originEdg to the view edge $(4, \text{German}, 2)$ in Fig. 7 results in the origin edge $(1, \text{German}, 0)$ in Fig. 6 which is then the source edge of that view edge. In GRoundTram, we highlight this correspondence.

C. Tracing Between View And Query

1) Tracing Between View And Edge Constructor: If a view edge ζ is constructed by an edge constructor at the code position p in the query, $\mathsf{tr\_originEdg}(\zeta)$ has the form $(\mathsf{Code}\ \ p\ \ m, \_, \_)$. In that case, we denote $\mathsf{tr\_edgCons}(\zeta) = \mathsf{tr\_originEdg}(\zeta)$. Otherwise, $\mathsf{tr\_edgCons}(\zeta)$ fails.

In GRoundTram, when a view edge created by an edge constructor is selected, we highlight the latter. For example, if the user clicks on the view edge $(14, \texttt{language}, 4)$ in Fig. 7, the edge constructor $\{\texttt{lang} : \$e\}$ of the **select** part in Listing 1 is highlighted because this constructor creates the view edge. No edge in the source graph (Fig. 6) is highlighted because the view edge is not a copy of any edge in the source graph.

2) Tracing Between View And Graph Variable: If a view edge ζ is a copy of a source edge (as discussed in Section IV-B), it must have been copied by a graph variable. Since graph variables do not leave tracing information on the edges, we need to track the execution path leading to the creation of ζ in the intermediate results produced during forward transformation (Section II-C). Moreover, since expressions in a query may contain multiple input marker types, identification of the input marker is required for correct identification of the graph variable reference responsible for ζ.

The set of input markers $\&m$ of ζ can be identified by traversing the view graph from ζ backwards to a $\mathsf{RecN}\ \ p\ \ \&m$ or $\mathsf{Code}\ \ p\ \ \&m$ and collecting the input markers in those nodes. If no such node exists, the input marker defaults to &. The algorithm for tracing graph variables is then as follows:

First, following the sequence of applied edges $\mathsf{tr\_applEdgs}(\zeta)$, 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. Additionally, let $Z = \mathsf{tr\_applEdgs}(\zeta)$ and $\zeta_S = \mathsf{tr\_originEdg}(\zeta)$ be the sequence of applied edges of ζ and the origin edge of ζ, respectively.

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

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

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

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

Fig. 8. Marker-oriented Tracing of Graph Variables

Then, we identify the corresponding variable reference by traversing the corresponding marker component in the UnCAL expression $\overline{e}$, resulting from applying tr_brec to Z, using the function tgv (Fig. 8) and $\pi_2 \circ \mathsf{tgv}$ returns the set of positions of the graph variables, while $\pi_1 \circ \mathsf{tgv}$ 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 [24]. In the definition in Fig. 8, the first case is the most important one since it checks whether the origin edge is in the evaluation result of the graph variable $\$g$. In GRoundTram, we can then highlight the found code positions of graph variables.

For example, applying tgv to the view $(3, \texttt{German}, 1)$ in Fig. 7 will return $(\{\}, \{p\})$ where p is the code position of the graph variable $\$lang$ of the **select** part in Listing 1. This graph variable copies the source edge $(1, \text{German}, 0)$ to the view graph.

V. Determining Editability

We introduce the notion of equivalence classes, with two view edges in the same equivalence class if backward transformation causes edits to them to propagate to the same source edge. There is one such equivalence class for each source edge that the changes can be propagated back to. Additionally, there are edges created with a constant label value defined in the UnCAL query: edits to those edges are not allowed at all.

We want to create a mapping $m_{\mathrm{All}} : \mathit{Edge} \to \mathit{Edge}_\perp$ that maps each view edge to its equivalence class or a $\perp$ symbol in case the edge is constant. Each equivalence class corresponds to exactly one source edge (where the changes to view edges of this class are propagated back to), hence it is feasible to use source edge identifiers as equivalence class identifiers.

To calculate m_{All} we need to do a dynamic evaluation based on the intermediate results of forward transformation as defined in Section II-C. First, a given query is evaluated with forward transformation. The intermediate results are then used as an input for the algorithm to calculate m_{All}. For this dynamic evaluation, we carry along a variable environment $\eta : \mathit{env}$ which is composed of a label variable environment $\eta_{\mathrm{L}} : \mathit{var}_{\mathrm{L}} \to \mathit{Edge}_\perp$ and a graph variable environment $\eta_{\mathrm{G}} : \mathit{var}_{\mathrm{G}} \to (\mathit{Edge} \to \mathit{Edge}_\perp)$. In UnCAL, a label variable might contain a label taken from an edge of the source graph, or it can be assigned to a constant value; our label variable environment thus maps each label to either its source edge or $\perp$, in case it is constant, whereas graph variables are mapped to equivalence class mappings of the same type as m_{All} in the graph variable environment.

We define $\mathcal{V}$ as the dynamic evaluation of the query which takes an intermediate result of forward transformation and an env as an input and returns a mapping like m_{All}. When applied to the intermediate results of the whole query, $\mathcal{V}$ returns m_{All}.

$$\mathcal{V} : \mathit{Eval} \to \mathit{env} \to (\mathit{Edge} \to \mathit{Edge}_\perp)$$

$\mathcal{V}$ does a structural recursion on the query constructs, as seen in Fig. 9. This definition makes use of several notational conventions and functions which are defined as follows:

- Let $f_\emptyset : \mathit{Edge} \to \mathit{Edge}_\perp$ be the partial function with an empty domain.
- Define $f\{x \mapsto y\}$ as
$$f\{x \mapsto y\}(a) = \begin{cases} y \text{ if } a = x \\ f(a) \text{ if } a \neq x \text{ and } f \text{ defined for } a \end{cases}$$
- 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 $\mathit{Edge} \to \mathit{Edge}_\perp$ mapping with the RecE trace information in order to make the result of $\mathcal{V}$ consistent with the view graph.

In the evaluation of $\mathcal{V}$, new variables are created in the **rec**-construct. For each iteration of $\overline{\mathbf{rec}}(\lambda(\$l, \$g).M)(_{G_1}\overline{e_{\mathrm{a}}})$, we configure a new variable environment η'_ζ which binds $\$l$ to the equivalence class of the key of M which we look up in the equivalence mapping m of the argument graph G_1. We also bind $\$g$ to the graph reachable from ζ. The evaluation results of each iteration of **rec** are then unified to produce a map for all edges created by the **rec** construct (not counting ϵ-edges).

$$\begin{aligned}
&\mathcal{V}\ _G\{\} && \eta = f_\emptyset \\
&\mathcal{V}\ _G() && \eta = f_\emptyset \\
&\mathcal{V}\ _G\{_L\$l : {}_{G_1}\overline{e}\} && \eta = \\
&\qquad && (\mathcal{V}\ _{G_1}\overline{e}\ \eta)\{(G.I(\&), L, G_1.I(\&)) \mapsto \eta_{\mathrm{L}}(\$l)\} \\
&\mathcal{V}\ _G\{a : {}_{G_1}\overline{e}\}\ (a \in \mathit{Label}) && \eta = \\
&\qquad && (\mathcal{V}\ _{G_1}\overline{e}\ \eta)\{(G.I(\&), a, G_1.I(\&)) \mapsto \perp\} \\
&\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_{\mathrm{G}}(\$g) \\
&\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_{\mathrm{L}}, \eta_{\mathrm{G}}\{\$g \mapsto \mathcal{V}\ _{G_1}\overline{e_1}\ \eta\}) \\
&\mathcal{V}\ _G(\mathbf{llet}\ \$l = \mathsf{a}\ \mathbf{in}\ {}_{G_1}\overline{e}) && \eta = \mathcal{V}\ _{G_1}\overline{e}\ (\eta_{\mathrm{L}}\{\$l \mapsto \perp\}, \eta_{\mathrm{G}}) \\
&\mathcal{V}\ _G(\mathbf{llet}\ \$l = \$l'\ \mathbf{in}\ {}_{G_1}\overline{e}) && \eta = \mathcal{V}\ _{G_1}\overline{e}\ (\eta_{\mathrm{L}}\{\$l \mapsto \eta_{\mathrm{L}}(\$l')\}, \eta_{\mathrm{G}}) \\
&\mathcal{V}\ _G\overline{\mathbf{rec}}(\lambda(\$l, \$g).M)(_{G_1}\overline{e_{\mathrm{a}}}) && \eta = \\
&\qquad \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_{\mathrm{L}}\{\$l \mapsto m(\zeta)\}, \eta_{\mathrm{G}}\{\$g \mapsto \mathsf{aux}(m, \zeta)\}) &&
\end{aligned}$$

Fig. 9. The algorithm for calculating the equivalence class mappings.

The **llet**-construct introduces a new label variable that is either a constant or copies the equivalence class from the right-hand side label variable. For graphs, the **let**-construct assigns the right-hand side graph's equivalence class mapping to the left-hand side variable.

Edges are created by graph variables $\$g$ or by edge constructors. In case of graph variables, the equivalence class mapping has been precomputed at the time of the declaration of the variable and only has to be looked up in the variable environment. In case of an edge constructor, a label variable or constant can be used as the label. In case of a constant we assign the edge to the class $\perp$, otherwise we look up the variable's equivalence class in the label variable environment. In any case, the edge created by the constructor has to be extracted from the intermediate graph results by getting its source and target node with the $\&$ input marker.

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.

Consider the example in Listing 1 with the source graph of Figure 6 and view graph of Figure 7. In order to apply $\mathcal{V}$, the UnQL query needs to be desugared to UnCAL, as seen in Listing 2. When applying $\mathcal{V}$ to our example, we get four equivalence classes, one each for the source edges $(1, \texttt{German}, 0)$, $(3, \texttt{Europe}, 2)$, $(5, \texttt{Austrian}, 4)$ and $(11, \texttt{German}, 10)$, as well as the $\perp$ class. Both view edges $(3, \texttt{German}, 1)$ and $(4, \texttt{German}, 2)$ are in the equivalence class of edge $(1, \texttt{German}, 0)$, so inconsistent edits can be prevented. Additionally, edges like $(0, \texttt{result}, 14)$ are in the equivalence class $\perp$ and marked as constant.

VI. Our GRoundTram System

We have integrated the tracing mechanisms in the form of highlighting and editability support described above into our GRoundTram system. We plan to publish the implementation on our project website at http://www.prg.nii.ac.jp/projects/gtcontrib/cmpbx/. In the following, we summarize the new features available to the users. Fig. 10 shows a screenshot of GRoundTram.

View edges have their origins highlighted when they are selected, be it a single view edge in the selection or a whole collection of edges. Direct copies from the source graph will have their corresponding source edge highlighted, as well as the graph variable in the query that produced them. Other view edges have the relevant edge constructor highlighted in the query instead. In Fig. 10, an edge constructor with a constant label as well as a graph variable are highlighted (in different colors for the benefit of the user). In addition, several source edges are identified as the origins of direct copies that are part of the view selection.

Highlighting also works in reverse to find the copies of source edges in the view as well as the corresponding view edges for a given graph constructor or graph variable in the query.

Constant edges are marked with dashed lines and editing their label is prohibited. When a view edge is selected, corresponding view edges within the same equivalence class are highlighted in a green color (also seen in Fig 10). The system prevents inconsistent edits from happening.

VII. Related Work

Tracing mechanism. Van Amstel et al. proposed a visualization framework for chains of ATL model transformations [21]. Systematic augmentation of the trace-generating capability with model transformations is achieved by higher-order model transformations [27]. Although they also trace between source, transformation and target, they deal with *unidirectional* transformations. Our own previous work [17] introduced the trace generation mechanism, but the main objective was the bidirectionalization itself. The notion of traces has been extensively studied in a more general context of computations, like provenance traces [28] for the nested relational calculus.

Lifting traces from the core language to its surface language (syntactic sugar), like UnCAL to UnQL in our work, is not easy in general due to the generally big gap between the two languages. Pombrio and Krishnamurthi [29] tackle with this gap by automatically reproducing an evaluation sequence of the core language in the surface language, which may provide a partial solution for us to cope with higher level sugar like **replace** in our previous work [25, 16].

Triple Graph Grammars (TGG) [30] and frameworks based on them are studied extensively and are applied to model-driven engineering [31, 32]. They are based on graph rewriting rules consisting of triples of source and target graph pattern, and the correspondence graph in-between which explicitly contains the trace information.

Our tracing is designed to work in compositional setting where arbitrary intermediate graph can be produced in the transformation.

Editability Analysis. Another well-studied bidirectional transformation framework called semantic bidirectionalization [23] generates a table of correspondence between elements in the source and those in the target to guide the reflection of updates over *polymorphic* transformations, and does not have to inspect the forward transformation code at all (thus called semantic). The entries in the target side of the table can be considered as equivalence classes to detect inconsistent updates on multiple target elements corresponding identical source element. Although UnCAL transformations are not polymorphic in general because of the label comparison in the **if** conditionals with constant labels, prohibiting the semantic bidirectionalization approach, Matsuda and Wang [22] relaxed this limitation by run-time recording of the branching behaviors and checking the change of the behavior to reject updates causing such change. They also cope with data constructed during transformation (corresponding to constant edges in our transformation). So our framework is close to theirs, though we utilize the syntax of transformation and reserve opportunities to recommend variety of possible consistent changes to edges in a equivalence class for more complex branching conditions.

VIII. Conclusion

Bidirectional transformations sometimes receive a reputation that the result of backward transformation is difficult to understand or predict. This applies to frameworks 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 propose, 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

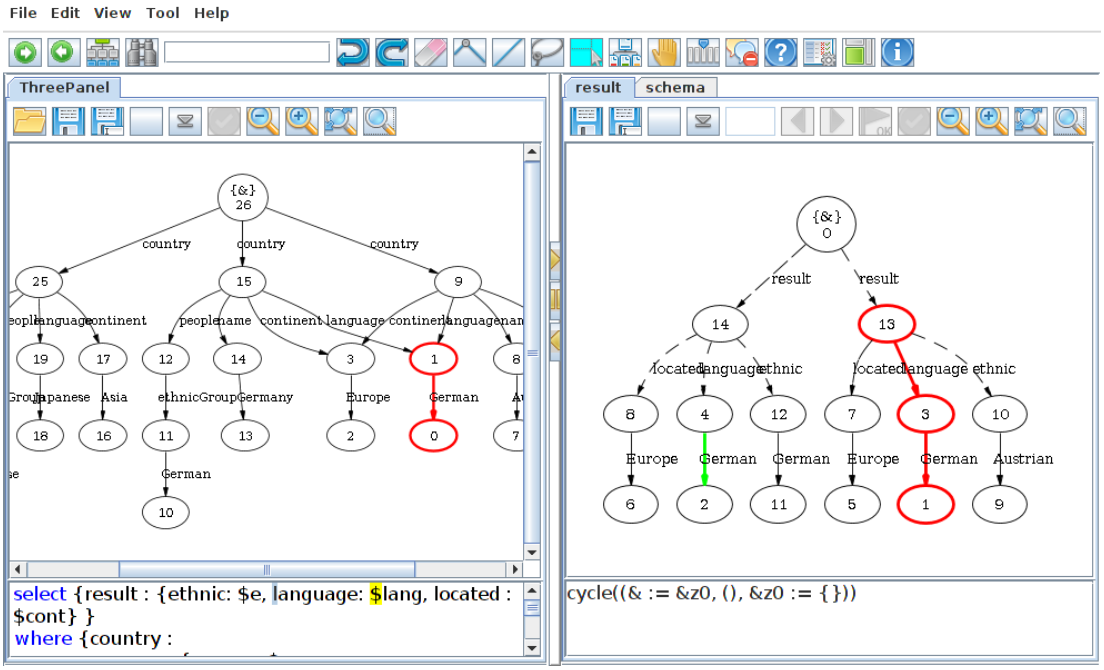


Fig. 10. Screenshot of the GRoundTram system showing traces between source graph, UnQL transformation and view graph

GRoundTram. In GRoundTram, the correspondence between edges in the view graph, edges in the source graph and graph constructors as well as graph variables in the transformation is visually highlighted. We leverage the trace information required by bidirectionalization to achieve this tracing and highlighting.

We also classify every edge in the target graph. Users can identify edges where editing is prohibited. They can also see groups of edges that cannot be inconsistently updated. Since the language we deal with allows arbitrary variable references that causes copies, identification of copies in the view has not been that trivial. Nonetheless, our proposal is simple because it just maintains the mapping from source edge to the view edge per variables and updates the mappings for each variable binding construct.

For better comprehension and prediction of bidirectional transformations, our GRoundTram system has been extended with the features introduced in this paper. The system is available for download on our project website.

As our future work, we have found a potential in the classification framework for tracing the occurrence positions of graph variables as well as edge constructors by extending the entry of the table maintained by the classifier to store the code positions. We would like to investigate this direction towards unifying tracing with classification. We also plan to overcome the limitations of tracing UnQL with optimization activated 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 expression. Our foray in this direction shows promising results.

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.

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 a separate general inversion strategy which is costly. We currently have limited support, however we do not have bidirectional properties for complex expressions. One obvious case we can support is subgraph extraction like $\mathbf{select}\ \{a : \$g\}\ \mathbf{where}\ \{a : \$g\}\ \mathbf{in}\ \$db$, in which we could insert an arbitrary subgraph below the top level edge labeled a. Because the subgraph is not "observed" in the transformation, an update will never interfere with the branching behavior. Even though that part is "observed" by the bulk semantics, that part is left unreachable, so the update does not affect the computation of the reachable part. If we extend the classifier function to indicate which edge is involved in the branching behavior, inspired by [22], we could safely determine the part that accepts the insertion or deletion of that part reusing the in-place update semantics, thus achieving "cheap backward transformation".

Acknowledgment

The authors would like to thank Zhenjiang Hu and Hiroyuki Kato for their valuable comments, Kazutaka Matsuda for discussing the branching behavior change detection as well as Jonas Winkler, Martin Beckmann und Kerstin Hartig for

reviewing the paper. The project was supported by the International Internship Program of the National Institute of Informatics.

References

[1] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *International Conference on Model Transformation (ICMT)*, ser. Lecture Notes in Computer Science, R. F. Paige, Ed., vol. 5563. Springer, 2009, pp. 260–283.

[2] F. Bancilhon and N. Spyratos, "Update semantics of relational views," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 557–575, 1981.

[3] U. Dayal and P. A. Bernstein, "On the correct translation of update operations on relational views," *ACM Trans. Database Syst.*, vol. 7, no. 3, pp. 381–416, 1982.

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

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

[6] P. Stevens, "Bidirectional model transformations in QVT: semantic issues and open questions," *Software and System Modeling*, vol. 9, no. 1, pp. 7–20, 2010.

[7] H. Giese and R. Wagner, "Incremental model synchronization with triple graph grammars," in *MoDELS 2006: Proceedings of the 9th nternational Conference on Model Driven Engineering Languages and Systems*. Springer Verlag, 2006, pp. 543–557.

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

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

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

[11] J. Grundy, J. Hosking, and W. B. Mugridge, "Inconsistency management for multiple-view software development environments," *IEEE Trans. Softw. Eng.*, vol. 24, no. 11, pp. 960–981, 1998.

[12] M. Garcia, "Bidirectional synchronization of multiple views of software models," in *Proceedings of DSML-2008*, ser. CEUR-WS, vol. 324, 2008, pp. 7–19.

[13] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "blinkit: Maintaining Invariant Traceability through Bidirectional Transformations," in *ICSE'12*, Zurich, Switzerland, Jun. 2012, pp. 540–550.

[14] S. Hidaka and J. F. Terwilliger, "Preface to the third international workshop on bidirectional transformations," in *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014)*, ser. CEUR Workshop Proceedings, K. S. Candan, S. Amer-Yahia, N. Schweikardt, V. Christophides, and V. Leroy, Eds., no. 1133, Aachen, 2014, pp. 61–62. [Online]. Available: http://ceur-ws.org/Vol-1133#paper-09

[15] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano, "GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations (short paper)," in *26th IEEE/ACM International Conference On Automated Software Engineering*. IEEE, 2011, pp. 480–483.

[16] ——, "GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations," *Progress in Informatics*, no. 10, pp. 131–148, March 2013, journal version of [15]. [Online]. Available: http://dx.doi.org/10.2201/NiiPi.2013.10.7

[17] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano, "Bidirectionalizing graph transformations," in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2010, pp. 205–216.

[18] P. Buneman, M. Fernandez, and D. Suciu, "UnQL: a query language and algebra for semistructured data based on structural recursion," *The VLDB Journal*, vol. 9, no. 1, pp. 76–110, 2000.

[19] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas, "From state- to delta-based bidirectional model transformations: The symmetric case," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, J. Whittle, T. Clark, and T. Kühne, Eds. Springer Berlin Heidelberg, 2011, vol. 6981, pp. 304–318.

[20] S. Hidaka, Z. Hu, H. Kato, and K. Nakano, "A compositional approach to bidirectional model transformation," in *ICSE New Ideas and Emerging Results track, ICSE Companion*. IEEE, 2009, pp. 235–238.

[21] M. F. van Amstel, M. G. J. van den Brand, and A. Serebrenik, "Traceability visualization in model transformations with tracevis," in *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations*, ser. ICMT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 152–159. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30476-7_10

[22] K. Matsuda and M. Wang, "Bidirectionalization for free with runtime recording: Or, a light-weight approach to the view-update problem," in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '13. New York, NY, USA: ACM, 2013, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/2505879.2505888

[23] J. Voigtländer, "Bidirectionalization for free! (pearl)," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. New York, NY, USA: ACM, 2009, pp. 165–176. [Online]. Available: http://doi.acm.org/10.1145/1480881.1480904

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

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

[26] ——, "Towards compostional approach to model transformations for software development," GRACE Center, National Institute of Informatics, Tech. Rep. GRACE-TR08-01, Aug. 2008.

[27] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ser. ECMDA-FA '09. Springer-Verlag, 2009, pp. 18–33.

[28] J. Cheney, U. A. Acar, and A. Ahmed, "Provenance traces," *CoRR*, vol. abs/0812.0564, 2008.

[29] J. Pombrio and S. Krishnamurthi, "Resugaring: Lifting evaluation sequences through syntactic sugar," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 361–371. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594319

[30] A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany*, ser. Lecture Notes in Computer Science, E. W. Mayr, G. Schmidt, and G. Tinhofer, Eds., vol. 903. Springer, Jun. 1995, pp. 151–163.

[31] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr, "Metamodel-based tool integration with moflon," in *ICSE '08*. ACM, 2008, pp. 807–810.

[32] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software & Systems Modeling*, vol. 8, no. 1, pp. 21–43, 2008.