ISSN 1884-0760

GRACE TECHNICAL REPORTS

Trace-based Approach to Editability and Correspondence Analysis for Bidirectional Graph Transformations

Soichiro Hidaka Martin Billes Quang Minh Tran
Kazutaka Matsuda

GRACE-TR 2015–04 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/

Trace-based Approach to Editability and Correspondence Analysis for Bidirectional Graph Transformations [⋆]

Soichiro Hidaka[1], Martin Billes[2], Quang Minh Tran[3], and Kazutaka Matsuda[4]

[1] National Institute of Informatics, Japan
`hidaka@nii.ac.jp`
[2] Augsburg University, Germany
`martin.billes@student.uni-augsburg.de`
[3] Daimler Center for IT Innovations, Technical University of Berlin, Germany
`quang.tranminh@dcaiti.com`
[4] The University of Tokyo
`kztk@is.s.u-tokyo.ac.jp`

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, by propagating changes in the artifacts over transformations bidirectionally. However, it is often difficult to understand the correspondence among elements of the artifacts such as to which part in the source an edit on the view is to be propagated. It is equally hard to predict whether an edit is propagable to the source, if the edit affects other parts in the target, or where in the transformation should be changed to accommodate the edit. These issues are critical for more complex transformations. In this paper, we propose an approach to analyzing the above correspondence as well as to classifying edges according to their editability on the target, in a compositional framework of bidirectional graph transformation. These are achieved by augmenting the forward semantics of the transformations with explicit correspondence traces. By leveraging this approach, it is possible to solve the above issues, without executing the entire backward transformation. We demonstrate the effectiveness of our approach via GUI using non-trivial transformations.

Keywords: Bidirectional Graph Transformation, Traceability, Editability

⋆ This is a full version of the paper submitted to the 8th International Conference on Model Transformation (ICMT) 2015. LNCS style file is used to format this article. This is also a revised version of GRACE Technical report GRACE-TR-2015-03 [13]. In particular, the mechanisms for correspondence analysis and editability analysis are unified in the present paper, and whereas the previous correspondence analysis relied on the structured node IDs, the present paper does not.

1 Introduction

Bidirectional transformations has been attracting interdisciplinary studies [7,21], for example as the view-update problem in the database community [3,8] and more recently in the programming language community [11,4]. In model-driven software engineering where artifacts are often refined through compositions of model transformations, bidirectional graph transformation is expected to play an important role, because it enables us to propagate changes in the artifacts over transformations bidirectionally. Other usage of bidirectional model transformation could be distilling a complicated model down to just the features a certain stakeholder is interested in, or converting information to a different platform, such as classes to relational databases. A bidirectional transformation consists of forward ($\mathcal{F}$) and backward ($\mathcal{B}$) transformations [7,21]. $\mathcal{F}$ takes a source model (here a source graph [16,17]) G_S and transforms it into a view graph G_V. $\mathcal{B}$ takes an updated view graph G'_V and returns an updated source graph G'_S, with possibly propagated updates.

$$\begin{tikzcd}
G_S \arrow[r, "\mathcal{F}"] \arrow[d, dashed] & G_V \arrow[d, "\text{edit}"] \\
G'_S & G'_V \arrow[l, "\mathcal{B}"]
\end{tikzcd}$$

In many, especially complex transformations, it is not immediately apparent whether a view edge has its origin in a particular source edge or a part in the transformation, and what that part is. Thus, it is not easy to tell where edits to the view edge are propagated back too. In particular, backward transformation rejects updates 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 for the user to predict whether these edits are accepted by backward transformation. Bidirectional transformations are known for being hard to comprehend and predict [10]. Two features are desirable: 1) Explicit highlighting of correspondence between source, view and transformation 2) Classification of artifacts according to their editability. This way, prohibited edits leading to violation of predefined properties can be recognized by the user early.

Our bidirectional transformation framework called GRoundTram (Graph Roundtrip Transformation for Models) [19,16,17] features compositionality, a user-friendly surface syntax, a tool for validating both models and transformations, and an optimization mechanism. However, the framework also had the above issues. So we have incorporated these features by augmenting the forward semantics with explicit trace information. Our main contribution is to externalize enough trace information through the augmentation and to utilize for correspondence and editability analysis. For the user, corresponding elements in the artifacts are highlighted with different colors according to editability and other properties. For example, the system highlights the corresponding source edge for a view 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. [2] 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 [27] to general bidirectionalization, is also capable of similar classification, while we reserve opportunities to recommend variety of consistent changes for more complex branching conditions. In addition, our approach can trace between nodes of the graph, not just edges.

The rest of the paper is organized as follows: Section 2 overviews our motivation and goal with a running example. The simplicity of the example is for the sake of explanation, and we have more involved examples related to software engineering in our project website mentioned in Section 6. Section 3 summarizes the semantics of our underlying graph data model, core graph language UnCAL [5], and its bidirectional interpretation [14]. Section 4 introduces an augmented semantics of UnCAL to generate trace information for the correspondence and editability analysis. Section 5 explains how the augmented forward semantics can be leveraged to realize correspondence and editability analysis. An implementation is found at the above website. Section 6 describes how the proposed mechanisms in the preceding sections are integrated in GRoundTram.[5] Section 7 discusses related work, and Section 8 concludes the paper with future work.

2 Motivating Example

This section exemplifies the importance of the desireble features mentioned in Sect. 1. Let us consider the source graph in Fig. 1 consisting of fact book information about different countries, and suppose we want to extract the information for European countries as the view graph in Fig. 2. This transformation can be written in UnQL (the surface language of our target language UnCAL) as below.

```
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 1.1. Transformation in UnQL

In the view graph (Fig. 2), three edges have identical labels "German" $(3, \mathrm{German}, 1)$, $(4, \mathrm{German}, 2)$ and $(12, \mathrm{German}, 11)$, but have different origins in the source graph and are produced by different parts in the transformation. For example, the edge $\zeta = (3, \mathrm{German}, 1)$ is the language of Germany and is a

[5] The implementation is based on our earlier approach [13], in which (1) Mechanism for correspondence analysis and editability analysis were separated, while present paper unifies them. (2) Correspondence traces mainly relied on the structured node ID, while current paper does not.

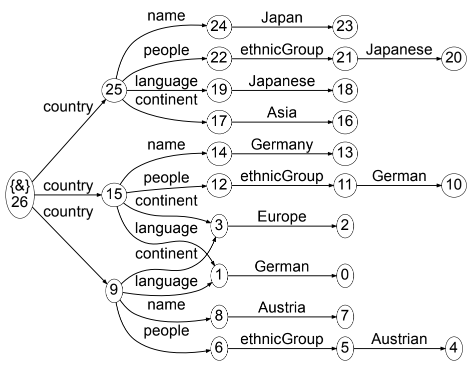


Fig. 1. Example source graph

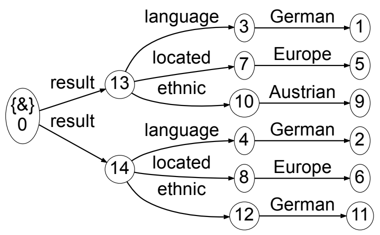


Fig. 2. View graph generated by transformation of the graph in Fig. 1

copy of the edge $(1, \text{German}, 0)$ of the source graph (Fig. 1). 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 much easier for the user to understand, if the system visually highlights 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 in the view. The user may not be aware of this and try to rename "located" to "location", but backward transformation rejects this change. Ideally, the system should make such constant edges easily recognizable to prevent the edits in the view and guide the user to make an edit to the constant in the transformation instead.

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 rejects this modification because these two view edges originate from the language "German" in a single edge of the source graph. The backward propagation of two edits would conflict at the source edge. The user may not realize this until the rejection. Ideally, the system would highlight groups of edges that could cause the conflict, and would prohibit triggering the backward transformation in that case.

Finally, suppose one of the edges labeled "Europe" in the view graph is edited to "Eurasia". Since the transformation depends on the label of this edge, changing it would lead to the selection of another conditional path in the transformation in a subsequent forward transformation. The renaming would cause an empty view after a round-trip of backward and forward transformation. To prevent this, such edits are rejected in our system. Changes in branch behavior are very difficult or impossible to predict, if the transformation is too complex. We can assist the prediction by highlighting the conditional branches involved.

3 Preliminaries

We use the UnCAL (Unstructured CALculus) query language [5]. UnCAL has an SQL-like syntactic sugar called UnQL (Unstructured Query Language) [5]. Listing 1.1 is written in UnQL. Bidirectional execution of graph transformation in UnQL is achieved by desugaring the transformation into UnCAL and then bidirectionally interpreting it [14]. This section explains the graph data model we use, as well as the UnCAL and UnQL languages.

3.1 UnCAL Graph Data Model

UnCAL graphs are multi-rooted and edge-labeled with all information stored in edge labels ranging over $\mathit{Label} \cup \{\varepsilon\}$ ($\mathit{Label}_\varepsilon$), node labels are only used as identifiers. There is no order between outgoing edges of a node. The notion of graph equivalence is defined by bisimulation; so equivalence between the graphs is efficiently determined [5], and graphs can be normalized [17] up to isomorphism.

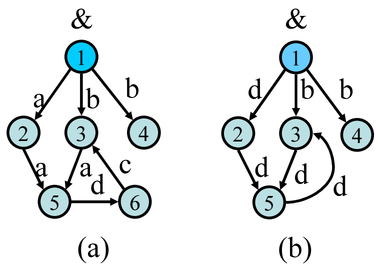


Fig. 3. Cyclic graph examples

Fig. 3 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 $\mathit{Edge}_\varepsilon$, where an edge is represented by a triple of source node, label and destination node. $I : \mathit{Marker} \rightarrow V$ is a function that identifies the roots (called *input nodes*) of a graph. Here, *Marker* is the set of markers of which element is denoted by $\&x$. We may call a marker in $\mathrm{dom}(I)$ (domain of I) an *input marker*. A special marker $\&$ is called the default marker. $O \subseteq V \times \mathit{Marker}$ assigns nodes with markers called *output markers*. If $(v, \&m) \in O$, v is called an *output node*. Intuitively output nodes serve as "exit points" where input nodes serve as "entry points". For example, the graph in Fig. 3 (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.V for V of graph $g = (V, E, I, O)$.

The type of a graphs is defined as the pair of the set of its input markers $\mathcal{X}$ and the set of output markers $\mathcal{Y}$, denoted by $DB^{\mathcal{X}}_{\mathcal{Y}}$. The graph in Fig. 3 (a) has type $DB^{\{\&\}}_{\emptyset}$. The superscript may be omitted, if the set is $\{\&\}$, and the subscript likewise, if the set is empty. The type of this graph is simply denoted by DB.

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

Fig. 4. Core UnCAL Language

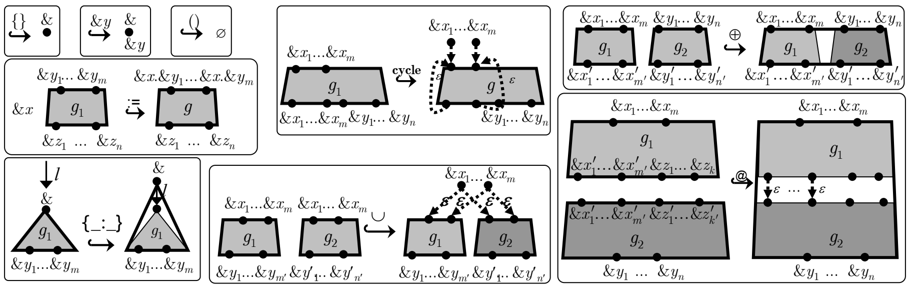


Fig. 5. Graph Constructors of UnCAL

3.2 UnCAL Query Language

Graphs can be created in the UnCAL query language [5], where there are nine graph constructors (Fig. 4) whose semantics is illustrated in Fig. 5. 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 $\mathcal{F}[\![()]\!] \in DB^{\emptyset}$, where $\mathcal{F}[\![e]\!]$ denotes the (forward) evaluation of expression e. 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 with an edge labeled l from the new root to $g.\mathrm{I}(\&)$; thus $\{l : g\} \in DB_{\mathcal{Y}}$. 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 the identical set of input markers $\mathcal{X} = \{\&x_1, \ldots, \&x_m\}$, constructs m new input nodes for each $\&x_i \in \mathcal{X}$, where each node has two ε-edges 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., $(\&x := g) \in DB_{\mathcal{Y}}^{\{\&x\}}$ 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 $\mathcal{X} \cap \mathcal{Y} = \emptyset$, 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. $g_1 @ g_2$ appends $g_1 \in DB_{\mathcal{X}' \cup \mathcal{Z}}^{\mathcal{X}}$ and $g_2 \in DB_{\mathcal{Y}}^{\mathcal{X}' \cup \mathcal{Z}'}$ by connecting the output and input nodes with a matching subset of markers $\mathcal{X}'$, and discards the rest of the markers, thus $g_1 @ g_2 \in DB_{\mathcal{Y}}^{\mathcal{X}}$. 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). The cycle construction $\mathbf{cycle}(g)$ for $g \in DB_{\mathcal{X} \cup \mathcal{Y}}^{\mathcal{X}}$ 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), where the notion of bisimilarity is extended to ε-edges [5].

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 **rec**, whose semantics is explained below by example. The variable binders **let** and **llet** having standard meanings are our extensions used for optimization by rewriting [15].

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

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

If the graph variable $\$db$ is bound to the graph in Fig. 3 (a), the result of the transformation will be the one in Fig. 3 (b). We call the first operand of **rec** the *body* expression and the second operand the *argument* expression. In the above transformation, the body is an **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 bidirectional evaluation (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 $\{\mathtt{d} : \&^1\}^2$ denote the code positions of the graph constructors $\&$ and $\{\mathtt{d} : \&\}$, respectively.

Fig. 6 shows the *bulk* semantics of **rec** for the example. It is “bulk” because the body of **rec** can be evaluated in parallel for each edge and the subgraph

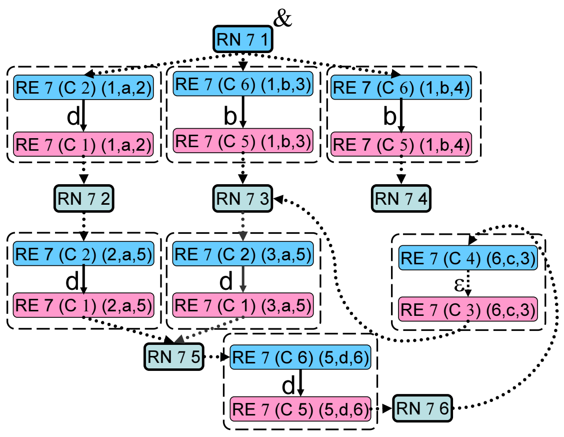


Fig. 6. Bulk semantics by example

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 [14] *StrID*:

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

where the base case (*SrcID*) represents the node identifier in the input graph, $\mathsf{Code}\ p\ \&x$ 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. $\mathsf{RecN}\ \ p\ v\ \&z$ 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. 6, the node $\boxed{\mathsf{RN}\ 7\ 1}$, 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 $(1, \mathsf{a}, 2)$, the result will be $(\{(\mathsf{C}\ 2), (\mathsf{C}\ 1)\}, \{(\mathsf{C}\ 2, \mathsf{d}, \mathsf{C}\ 1)\}, \{\& \mapsto \mathsf{C}\ 2\}, \{(\mathsf{C}\ 2, \&)\})$, with the nodes $\mathsf{C}\ 2$ and $\mathsf{C}\ 1$ 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 $\mathsf{RE}\ p\ v\ \zeta$ for **rec** at position p. These results are surrounded by round squares drawn with dashed lines in Fig. 6. They are then connected together according to the original shape of the graph as depicted in Fig. 6. For example, the input node $\boxed{\mathsf{RE}\ 7\ (\mathsf{C}\ 2)\ (1, \mathsf{a}, 2)}$ is connected with $\boxed{\mathsf{RN\ 7\ 1}}$. After removing the ε-edges and flattening the node IDs, we obtain the result graph in Fig. 3 (b).

Our bidirectional transformation in UnCAL is based on its bidirectional evaluation, whose semantics is given by $\mathcal{F}[\![\_]\!]$ and $\mathcal{B}[\![\_]\!]$ as follows. $\mathcal{F}[\![e]\!]\rho = G$ is the forward semantics applied to UnCAL query e with source variable environment ρ, which includes a global variable binding $\$db$ the input graph. $\mathcal{B}[\![e]\!](\rho, G') = \rho'$ produces the updated source ρ' given the updated view graph G' and the original source ρ.

Bidirectional transformations need to satisfy round-trip properties [7,25], while ours satisfy the GetPut and WPutGet properties [14], which are:

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

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* [9], a weaker notion of PutGet [11] or Correctness [25] or Consistency [3] because of the rather arbitrary variable reference allowed in our language) demands that for a second view graph G''_V which is returned by $\mathcal{F}[\![e]\!]\rho'$, that backward transformation $\mathcal{B}[\![e]\!](\rho, G''_V)$ using this second view graph as well as the original source environment ρ (from the first round of forward transformation) returns ρ' again unchanged.

In the backward evaluation of **rec**, the final ε-elimination to hide them from the user is reversed to restore the shape of Fig. 6, and then the graph is decomposed with the help of the structured 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 $\{\$l : \mathtt{\&}\}$, which produces the binding of $\$l$ updated with `x` and is reflected to the source graph with edge $(1, \mathtt{b}, 3)$, replaced by $(1, \mathtt{x}, 3)$.

UnQL as a Textual Surface Syntax of Bidirectional Graph Transformation We use the surface language UnQL [5] (Fig. 7) 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 [5,20,18] 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.

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

Fig. 7. Syntax of UnQL

$$
\begin{array}{lcl}
 & & \textbf{rec}(\lambda(\$l,\$g).\ \textbf{if}\ \$l = \text{a} \\
\textbf{select}\ \{\text{res:}\$db\} & & \quad \textbf{then rec}(\lambda(\$l',\$g).\ \textbf{if}\ \$l' = \text{b} \\
\textbf{where}\ \{\text{a:}\$g\}\ \textbf{in}\ \$db, & \Rightarrow & \qquad \textbf{then}\ \{\text{res:}\$db\} \\
\qquad \{\text{b:}\$g\}\ \textbf{in}\ \$db & & \qquad \textbf{else}\ \{\})(\$db) \\
 & & \quad \textbf{else}\ \{\})(\$db).
\end{array}
$$

4 Trace-augmented forward semantics of UnCAL

This section describes the forward semantics of UnCAL augmented with explicit correspondence traces. In the trace, every view edge and node is mapped to a corresponding source edge or node or a part of the transformation. The path taken in the transformation is also recorded for the view elements.

The trace information is utilized for (1) correspondence analysis, where a selected source element is contrasted with its corresponding view element(s) by highlighting them, and likewise, a selected view element is contrasted with its corresponding parts in source and transformation, and for (2) editability analysis, classifying edges by origins to (2-1) pre-reject the editing of edges that map to the transformation, (2-2) pre-reject the conflicting editing of view edges whose edit would be propagated to the same source edge, (2-3) warn the edit that could violate WPutGet by changing branching behavior of **if** by highlighting the branch conditions that are affected by the edit.

The augmented forward evaluation $\mathcal{F}[\![\_]\!] : \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Graph} \times \mathit{Trace}$ takes an UnCAL expression and an environment as arguments, and produces a target graph and trace. The trace maps an edge $\in \mathit{Edge}$ (resp. a node $\in \mathit{Node}$) to a source edge (resp. source node) or a code position, preceded by zero or more code positions that represent the corresponding language constructs involved in the transformation that produced the edge (resp. the node). The preceding parts are used for the warning in (2-3) above. Thus

$$\begin{aligned}
&\mathit{Trace} \;\;=\;\; \mathit{Edge} \cup \mathit{Node} \to \mathit{Trace}_{\mathrm{E}} \cup \mathit{Trace}_{\mathrm{V}}\\
&\mathit{Trace}_{\mathrm{E}} ::= \mathit{Pos} : \mathit{Trace}_{\mathrm{E}} \mid [\mathit{Edge} \mid \mathit{Pos}]\\
&\mathit{Trace}_{\mathrm{V}} ::= \mathit{Pos} : \mathit{Trace}_{\mathrm{V}} \mid [\mathit{Node} \mid \mathit{Pos}]
\end{aligned}$$

The environment Env represents bindings of graph variables and label variables. Each graph variable is mapped to a graph with a trace, while each label variable is mapped to a label with a trace that contains only edge mapping. Thus

$$\mathit{Env} = \mathit{Var} \to (\mathit{Graph} \times \mathit{Trace}) \cup (\mathit{Label} \times \mathit{Trace}_{\mathrm{E}}).$$

Given source graph g_s and the corresponding distinct graph variable $\$db$, the top level environment ρ_0 is initialized as follows.

$$\rho_0 = \{\$db \mapsto (g_s, \{\zeta \mapsto [\zeta] \mid \zeta \in g_s.\mathrm{E}\} \cup \{v \mapsto [v] \mid v \in g_s.\mathrm{V}\})\}$$

As idioms used in the following, we introduce two auxiliary functions of type $\mathit{Trace} \to \mathit{Trace}$: prep_p to prepend code position $p \in \mathit{Pos}$ to traces, and $\mathsf{rece}_{p,\zeta}$ to "wrap" the nodes in the domain of traces with RecE constructor to adjust to bulk semantics.

$$\begin{aligned}
\mathsf{prep}_p\, t &= \{\quad x \;\mapsto p{:}\tau \mid (x \mapsto \tau) \in t\}\\
\mathsf{rece}_{p,\zeta}\, t &= \{(f\ x) \mapsto \quad \tau \mid (x \mapsto \tau) \in t\}\\
\textbf{where}\, f x &= \begin{cases} (\mathsf{RecE}\ \ p\ u\ \zeta, l, \mathsf{RecE}\ \ p\ v\ \zeta) & \text{if } x = (u,l,v) \in \mathit{Edge}\\ \mathsf{RecE}\ \ p\ x\ \zeta & \text{if } x \quad\quad\quad\;\; \in \mathit{Node}\end{cases}
\end{aligned}$$

Now we describe the semantics $\mathcal{F}[\![\_]\!]$. The graph component returned by the semantics is the same as that of [14] and recapped in Section 3, so we focus here on the trace parts. The subscripts on the left of the constructor expressions represent the result graph of the constructions.

$$\mathcal{F}[\![\{\}^p]\!]_\rho = ({}_G\{\}^p, \{G.\mathrm{I}(\&) \mapsto [p]\}) \tag{T-EMP}$$

$$\mathcal{F}[\![\&y^p]\!]_\rho = ({}_G\&y^p, \{G.\mathrm{I}(\&) \mapsto [p]\}) \tag{OMRK}$$

$$\mathcal{F}[\![()^p]\!]_\rho = (()^p, \emptyset) \tag{G-EMP}$$

$$\begin{aligned}\mathcal{F}[\![e_1 \cup^p e_2]\!]_\rho &= ({}_G(g_1 \cup^p g_2), (t_1 \cup t_2 \cup \{v \mapsto [p] \mid (\&x \mapsto v) \in G.\mathrm{I}\}))\\ &\quad \textbf{where}\quad ((g_1,t_1),(g_2,t_2)) = (\mathcal{F}[\![e_1]\!]\rho, \mathcal{F}[\![e_2]\!]\rho)\end{aligned} \tag{UNI}$$

$$\begin{aligned}\mathcal{F}[\![e_1 \oplus^p e_2]\!]_\rho &= (g_1 \oplus^p g_2, t_1 \cup t_2)\\ &\quad \textbf{where}\quad ((g_1,t_1),(g_2,t_2)) = (\mathcal{F}[\![e_1]\!]\rho, \mathcal{F}[\![e_2]\!]\rho)\end{aligned} \tag{DUNI}$$

$$\begin{aligned}\mathcal{F}[\![e_1\ @^p\ e_2]\!]_\rho &= (g_1\ @^p\ g_2, t_1 \cup t_2)\\ &\quad \textbf{where}\quad ((g_1,t_1),(g_2,t_2)) = (\mathcal{F}[\![e_1]\!]\rho, \mathcal{F}[\![e_2]\!]\rho)\end{aligned} \tag{APND}$$

$$\begin{aligned}\mathcal{F}[\![\{e_{\mathrm{L}} : e\}^p]\!]_\rho &= ({}_G\{l : g\}^p, \{(v, l, g.\mathrm{I}(\&)) \mapsto \tau, v \mapsto [p]\} \cup t)\\ &\quad \textbf{where}\quad ((l,\tau),(g,t)) = (\mathcal{F}_{\mathrm{L}}[\![e_{\mathrm{L}}]\!]_\rho, \mathcal{F}[\![e]\!]_\rho)\end{aligned} \tag{Edg}$$

$$\begin{aligned}\mathcal{F}[\![(\&x := e)^p]\!]_\rho &= (\&x := g, t)\\ &\quad \textbf{where}\quad (g,t) = \mathcal{F}[\![e]\!]_\rho\end{aligned} \tag{IMRK}$$

$$\begin{aligned}\mathcal{F}[\![\mathbf{cycle}^p(e)]\!]_\rho &= ({}_G\mathbf{cycle}^p(g), t \cup \{v \mapsto [p] \mid (\&x \mapsto v) \in G.\mathrm{I}\})\\ &\quad \textbf{where}\quad (g,t) = \mathcal{F}[\![e]\!]_\rho\end{aligned} \tag{CYC}$$

For the constructor {}, the trace maps the only node ($G.\mathrm{I}(\texttt{\&})$) created, to the code position p of the constructor (T-EMP). The same trace information is generated for the output marker constructor (OMRK). Since neither edge nor node is created by the constructor (), no trace information is generated (G-EMP). The binary graph constructors $\cup$, $\oplus$ and @ returns the traces of both subexpressions, in addition to the trace created by themselves. The graph union's trace maps newly created input nodes to the code position (UNI). Since other two binary constructors do not create any node or non-ε edge, no additional trace is generated (DUNI,APND). For the edge-constructor (Edg), the correspondence between the created edge and its trace created by the label expression e_{L} is established, while the newly created node is mapped to the code position of the constructor. The marker-renaming expression (IMRK) does not add any trace to that of its subexpression, since no additional node or edge is created. The cycle expression (CYC) adds traces that map the roots created by the constructor to the code position of the constructor.

For label expression evaluation $\mathcal{F}_{\mathrm{L}}[\![\_]\!] : \mathit{Expr}_{\mathrm{L}} \to \mathit{Env} \to \mathit{Label} \times \mathit{Trace}_{\mathrm{E}}$, the trace that associates the label to the corresponding edge or code position is accompanied with the resultant label value.

$$\mathcal{F}_{\mathrm{L}}[\![\mathsf{a}^p]\!]_\rho = (\mathsf{a}, [p]) \qquad \text{(LCNST)}$$

$$\begin{aligned}\mathcal{F}_{\mathrm{L}}[\![\$l^p]\!]_\rho &= (l, p : \tau) \\ &\quad \textbf{where } (l, \tau) = \rho(\$l)\end{aligned} \qquad \text{(LVAR)}$$

Label literal expressions (LCNST) record their code positions, while label variable reference expressions (LVAR) add their code positions to the traces that are registered in the environment.

Label variable binding expression (LLET) registers the trace to the environment and passes it to the forward evaluation of the body expression e. Graph variable binding expression (LET) is treated similarly, except it handles graphs and their traces. Graph variable reference (VAR) retrieves traces from the environment and add the code position to it.

$$\begin{aligned}\mathcal{F}[\![\textbf{llet}^p\ \$l = e_{\mathrm{L}}\ \textbf{in}\ e]\!]_\rho &= \mathcal{F}[\![e]\!]_{\rho \cup \{\$l \mapsto (l, p:\tau)\}} \\ &\quad \textbf{where } (l, \tau) = \mathcal{F}_{\mathrm{L}}[\![e_{\mathrm{L}}]\!]_\rho\end{aligned} \qquad \text{(LLET)}$$

$$\begin{aligned}\mathcal{F}[\![\textbf{let}^p\ \$g = e_1\ \textbf{in}\ e_2]\!]_\rho &= \mathcal{F}[\![e_2]\!]_{\rho \cup \{\$g \mapsto (g, \mathsf{prep}_p\, t)\}} \\ &\quad \textbf{where } (g, t) = \mathcal{F}[\![e_1]\!]_\rho\end{aligned} \qquad \text{(LET)}$$

$$\begin{aligned}\mathcal{F}[\![\$g^p]\!]_\rho &= (g, \mathsf{prep}_p\, t) \\ &\quad \textbf{where } (g, t) = \rho(\$g)\end{aligned} \qquad \text{(VAR)}$$

$$\begin{aligned}\mathcal{F}\left[\!\!\left[\begin{array}{l}\textbf{if}^p(e_{\mathrm{L}} = e'_{\mathrm{L}})\ \textbf{then}\ e_{\mathrm{true}} \\ \qquad\qquad\quad\ \textbf{else}\ \ e_{\mathrm{false}}\end{array}\right]\!\!\right]_\rho &= (g, \mathsf{prep}_p\, t) \\ &\quad \textbf{where } ((l, \_), (l', \_)) = (\mathcal{F}_{\mathrm{L}}[\![e_{\mathrm{L}}]\!], \mathcal{F}_{\mathrm{L}}[\![e'_{\mathrm{L}}]\!]) \\ &\qquad\qquad\qquad\qquad\quad b = (l = l') \\ &\qquad\qquad\qquad\quad (g, t) = \mathcal{F}[\![e_b]\!]_\rho\end{aligned} \qquad \text{(IF)}$$

Structural recursion **rec** (REC) introduces new environment for the label ($\$l$) and graph ($\$g$) variable that includes traces inherited from the traces generated by the argument expression e_{a}, augmented with the code position p of **rec**. g_v denotes the subgraph of graph g that is reachable from node v. t_v denotes a trace that are restricted to the subgraph reachable from node v. The function M takes an edge and returns the pair of the graph created by the body expression e_{b} for the edge, and the trace associated with the graph. t_ζ is the trace generated by adjusting the trace returned by M with the node structure introduced by **rec**. $\mathsf{compose}^p_{\mathbf{rec}}$ is the bulk semantics explained in Sect. 3.2 using Fig. 6, for the input graph g and the input/output marker $\mathcal{Z}$ of e_{b}, where V_{RecN} denotes the nodes with structured ID RecN.

$$\mathcal{F}[\![\mathbf{rec}_{\mathcal{Z}}(\lambda(\$l,\$g).e_{\mathrm{b}})(e_{\mathrm{a}})]\!]_\rho = (g', \bigcup_{\zeta \in g.\mathrm{E}} t_\zeta \cup t'_{\mathrm{V}}) \qquad (\textsc{Rec})$$

$$\begin{aligned}
\mathbf{where}\ (g,t) &= \mathcal{F}[\![e_{\mathrm{a}}]\!]\rho \\
M &= \{\zeta \mapsto \mathcal{F}[\![e_{\mathrm{b}}]\!]_{\rho'} \mid \zeta \in g.\mathrm{E}, \zeta \neq \varepsilon, (u,l,v) = \zeta, \\
&\qquad \rho' = \rho \cup \{\$l \mapsto (l, p : t(\zeta)), \$g \mapsto (g_v, \mathsf{prep}_p\, t_v)\}\} \\
g' &= (V_{\mathsf{RecN}} \cup \ldots, \_, \_, \_) = \mathsf{compose}^p_{\mathbf{rec}}(M, g, \mathcal{Z}) \\
t'_{\mathrm{V}} &= \{v \mapsto [p] \mid v \in V_{\mathsf{RecN}}\} \\
t_\zeta &= \mathsf{rece}_{p,\zeta}(\mathsf{prep}_p\, \pi_2(M(\zeta)))
\end{aligned}$$

The above semantics collects all the necessary trace information whose utilization is described in the next section. Even though the tracing mechanisms are defined for UnCAL, they also work straightforwardly for UnQL, based on the observation that 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 {`language` : $\$lang$} and the graph variable $\$lang$ of the UnQL query in Listing 1.1 are transferred to the generated UnCAL query in Listing 1.2.

```
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 1.2. UnCAL expression of UnQL query in Listing 1.1

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

5 Correspondence and Editability Analysis

$$\begin{tikzcd} & \mathit{Query} \arrow[dr, leftrightarrow] & \\ \mathit{Source} \arrow[rr, leftrightarrow] & & \mathit{View} \end{tikzcd}$$

This section elaborates utilization of the traces defined in Sect. 4 for the correspondence and editability analysis motivated in Sect. 2. It allows us to tell the correspondence between elements of the source graph, code positions of the transformation query and elements of the view graph, and editability of the edges in the view graph.

Given transformation e, environment ρ_0, and the corresponding trace t for $(g, t) = \mathcal{F}[\![e]\!]_{\rho_0}$ through semantics given in Sect. 4, the trace for view edge ζ has the following form

$$t(\zeta) = p_1 : p_2 : \ldots : p_n : [x] \quad (n \geq 0)$$

where x is the origin, and $x = \zeta' \in \mathit{Edge}$ if ζ is a copy of ζ' in the source graph, or $x = p \in \mathit{Pos}$ if ζ is created of a label constant at position p in the transformation. $p_1, p_2, \ldots, p_n$ represent code positions of variable definitions/references and conditionals that conveyed ζ.

For view graph g and trace t, define the function $\mathsf{origin} : \mathit{Edge} \to \mathit{Edge} \cup \mathit{Pos}$ and its inverse:

$$\begin{aligned} \mathsf{origin}\ \ \zeta &= \mathsf{last}(t(\zeta)) \\ \mathsf{origin}^{-1}\ x &= \{\zeta \mid \zeta \in g.\mathrm{E}, \mathsf{origin}\,\zeta = x\} \end{aligned}$$

Correspondence is then the relation between the domain and image of trace t, and various individual correspondence can be derived, the most generic one being $R : \mathit{Edge} \cup \mathit{Node} \cup \mathit{Pos} \times \mathit{Edge} \cup \mathit{Node} = \{(x', x) \mid (x \mapsto \tau) \in t, x' \in \tau\}$, meaning that x' and x is related if $(x', x) \in R$. Source-target correspondence being $\{(x', x) \mid (x \mapsto \tau) \in t, x' = \mathsf{last}\,\tau, x' \in (\mathit{Node} \cup \mathit{Edge})\}$. For correspondence analysis in the GUI, ζ' is highlighted in the source graph pane, while language constructs at p for all $p_1, p_2, \ldots, p_n$ are highlighted in the transformation pane. Using origin and origin^{-1}, corresponding source, transformation and view elements can be identified in both directions. When a view element such as the edge $\zeta = (4, \mathrm{German}, 2)$ in Fig. 2 is given, we can find the corresponding source edge $\mathsf{origin}(\zeta) = (1, \mathrm{German}, 0)$, which will be updated if we change ζ. In contrast, given the view edge $(14, \texttt{language}, 4)$, the code position of the label constant `lang` in $\{\texttt{lang} : \$e\}$ of the **select** part in Listing 1.1 is obtained. Given the view edge $\zeta = (3, \texttt{German}, 1)$, the code positions of the the graph variables $\$\mathit{lang}$ of the **select** part and $\$\mathit{db}$ in Listing 1.1 are obtained, utilizing code positions in $p_1, \ldots, p_n$, because $t\ \zeta$ includes such positions. These graph variables copy the source edge $\mathsf{origin}(\zeta) = (1, \mathrm{German}, 0)$ to the view graph. Correspondence also works in reverse direction. For example, when a source edge

$\zeta' = (1, \texttt{German}, 0)$ is given, we can find the set of corresponding view edges $\{(3, \texttt{German}, 1), (4, \texttt{German}, 2)\}$, since $\mathsf{origin}^{-1}\zeta'$ is equal to this set.

For editability analysis, the following notion of equivalence is used. Given the partial function $\mathsf{origin}_{\mathrm{E}} : \mathit{Edge} \to \mathit{Edge}$ defined by $\mathsf{origin}_{\mathrm{E}}\ \zeta = \mathsf{origin}(\zeta)$ if $\mathsf{origin}(\zeta) \in \mathit{Edge}$, or undefined otherwise. Then, view edges ζ_1 and ζ_2 are equivalent, denoted by $\zeta_1 \sim \zeta_2$, if and only if $\mathsf{origin}_{\mathrm{E}}\, \zeta_1 = \mathsf{origin}_{\mathrm{E}}\, \zeta_2$. All edges for which $\mathsf{origin}_{\mathrm{E}}$ is undefined are considered equivalent. The equivalence class for edge ζ is denoted by $[\zeta]_\sim$. Then,

1. An edit on the view edge ζ with $\zeta' = \mathsf{origin}_{\mathrm{E}}\, \zeta$ defined is propagable to ζ' in the source graph by $\mathcal{B}$, when both of the followings hold.
   (a) Other view edges in $[\zeta]_\sim$ are unchanged or consistently updated.
   (b) For every **if** $e_{\mathrm{B}} \ldots$ expression in the backward evaluation path in which e_{B} refers a variable $\$l$ that binds label of edges originated from ζ' (i.e., $\rho(\$l) = (\_, \tau)$ and $\mathsf{last}(\tau) = \zeta'$), applying the edit to the binding does not change the condition.
2. An edit on the view edge ζ with $\mathsf{origin}(\zeta) = p \in \mathit{Pos}$ is not propagable to the source. Editing label constant at p in the transformation would achieve the edits, with possible side effects through other copies of the label constant.

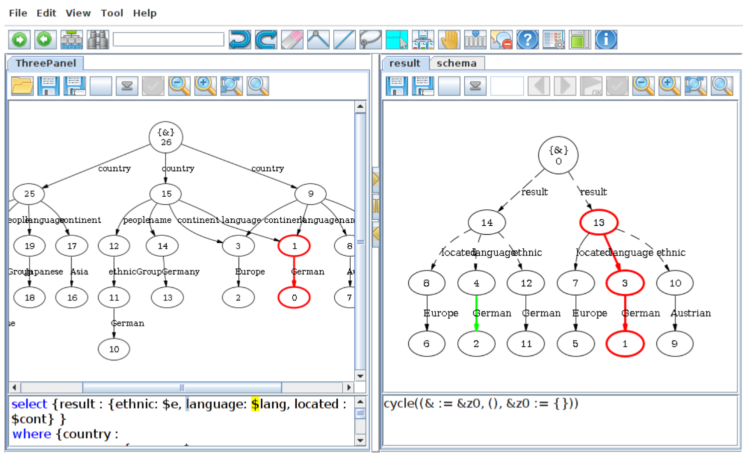
Consider the example in Listing 1.1 with the source graph of Fig. 1 and view graph of Fig. 2. We get four equivalence classes, one each for the source edges $(1, \texttt{German}, 0)$, $(3, \texttt{Europe}, 2)$, $(5, \texttt{Austrian}, 4)$ and $(11, \texttt{German}, 10)$, as well as the class that satisfy condition 2. For view edge $\zeta = (3, \texttt{German}, 1)$, we have $(4, \texttt{German}, 2) \in [\zeta]_\sim$ via $\mathsf{origin}_{\mathrm{E}}\, \zeta = (1, \texttt{German}, 0)$, so these equivalent edges can be selected simultaneously, inconsistent edits on which can be prevented. Direct edits of the view edge $\zeta = (0, \texttt{result}, 14)$ are suppressed since $\mathsf{origin}\, \zeta \in \mathit{Pos}$ (condition 2). As for case 1b, when the view edge $\zeta = (7, \texttt{Europe}, 5)$ is given, all the code positions in $t\ \zeta''$ for $\zeta'' \in \mathsf{origin}_{\mathrm{E}}{}^{-1}(\mathsf{origin}_{\mathrm{E}}\, \zeta)$ are checked if the positions represent conditionals that refer variables, change of those binding would change the conditions and would be rejected by $\mathcal{B}$. We obtain the position for variable reference $\$l$ in the condition $(\$l = \texttt{Europe})$ for warning.

Details on Branch Behavior Change Rejection in the Backward Transformation and its Prevention Here we briefly review how updates in case 1b are rejected by $\mathcal{B}$. Given transformation e and view edge ζ to be updated and the corresponding source edge $\zeta' = \mathsf{origin}_{\mathrm{E}}\, \zeta$, for each $\mathcal{B}[\![\textbf{if}\ \ e_{\mathrm{B}} \ldots]\!]$ invoked from $\mathcal{B}[\![e]\!]$, for all bindings of label variables with the same source edge, in the condition expression e_{B}, label update to ζ' is applied, and e_{B} is reevaluated based on the updated bindings. If the reevaluation result differs from that in the forward direction, the update is rejected.

Next we explain how the case 1b is detected without conducting the entire backward transformation. Suppose $\mathsf{origin}_{\mathrm{E}}{}^{-1}\zeta' = E$ and let E' the set of edges in E that are reachable from the roots of the view graph. For each $\zeta'' \in E'$, $\tau = t\ \zeta''$ denotes all the trace involved for source edge ζ'. Then for all code positions in τ for **if** expression, apply similar propagation of updates for label variables in the condition expression. If any of the conditions change, then the condition 1b is detected and edits of ζ are prevented.

6 Our GRoundTram System

We have integrated the tracing mechanisms in the form of highlighting and editability support described above into our GRoundTram system. The implementation is 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. 8 shows a screenshot of GRoundTram.



```
select {result : {ethnic: $e, language: $lang, located :
$cont} }
where {country :
```

```
cycle((& := &z0, (), &z0 := {}))
```

Fig. 8. Screenshot of the GRoundTram system showing traces between source graph, UnQL transformation and view graph

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. 8, 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 8). The system prevents inconsistent edits from happening.

7 Related Work

Tracing mechanism. Van Amstel et al. proposed a visualization framework for chains of ATL model transformations [2]. Systematic augmentation of the trace-generating capability with model transformations is achieved by higher-order model transformations [26]. Although they also trace between source, transformation and target, we also use the trace for editability analysis. Our own previous work [14] 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 [6] 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 [23] 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 [20,17].

Triple Graph Grammars (TGG) [24] and frameworks based on them are studied extensively and are applied to model-driven engineering [1]. 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. Grammar rules are used to create new elements in source, correspondence and view graphs consistently. By iterating over items in the correspondence graph, incremental TGG approaches can also work with updates and deletions of elements [12]. Our transformation language UnQL is compositional in that it allows arbitrary intermediate graphs to be produced in the transformation, while TGG is not. Our tracing over compositions are achieved by keeping track of variable bindings.

Editability Analysis. Another well-studied bidirectional transformation framework called semantic bidirectionalization [27] generates a table of correspondence between elements in the source and those in the target to guide the reflection of updates over *polymorphic* transformations, without inspecting the forward transformation code (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 and change checking of the branching behaviors 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. We can trace nodes (though we focused on edge tracing in this paper) while theirs cannot.

8 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 is completely provided, rather than (partially) left to the users, so the logic of backward transformation is rather fixed and hidden by the semantics. The prediction is even more difficult for more complex transformations. In this paper, we proposed, within a compositional bidirectional graph transformation framework based on structural recursion, a technique for analyzing the correspondence between source, transformation and target as well as to classifying edges according to their editability. We achieve this by augmenting the forward semantics with explicit correspondence traces. The incorporation of this technique into the GUI enables the user to clearly visualize the correspondence. Moreover, prohibited edits such as changing a constant edge and updating a group of edges inconsistently are disabled. This allows the user to predict violated updates and thus do not attempt them at the first place.

As a future work, we would like to accommodate 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 **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, 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. Leveraging the traces in the forward semantics indicating which edge is involved in the branching behavior, 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".

We also would like 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.

Acknowledgments. The authors would like to thank Zhenjiang Hu and Hiroyuki Kato for their valuable comments. The project was supported by the International Internship Program of the National Institute of Informatics.

References

1. Amelunxen, C., Klar, F., Königs, A., Rötschke, T., Schürr, A.: Metamodel-based tool integration with MOFLON. In: ICSE '08. pp. 807–810. ACM (2008)
2. van Amstel, M.F., van den Brand, M.G.J., Serebrenik, A.: Traceability visualization in model transformations with TraceVis. In: ICMT'12. pp. 152–159 (2012)
3. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. 6(4), 557–575 (1981)
4. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful lenses for string data. In: POPL '08. pp. 407–419 (2008)
5. Buneman, P., Fernandez, M., Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion. The VLDB Journal 9(1), 76–110 (2000)
6. Cheney, J., Acar, U.A., Ahmed, A.: Provenance traces. CoRR abs/0812.0564 (2008)
7. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: ICMT'09. pp. 260–283 (2009)
8. Dayal, U., Bernstein, P.A.: On the correct translation of update operations on relational views. ACM Trans. Database Syst. 7(3), 381–416 (1982)
9. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: The symmetric case. In: MODELS'11, LNCS, vol. 6981, pp. 304–318 (2011)
10. Eramo, R., Pierantonio, A., Rosa, G.: Uncertainty in bidirectional transformations. In: Proc. 6th International Workshop on Modeling in Software Engineering (MiSE). pp. 37–42. ACM (2014)
11. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29(3) (2007)
12. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software & Systems Modeling 8(1), 21–43 (2008)
13. Hidaka, S., Billes, M., Tran, Q.M.: A trace-based approach to increased comprehensibility and predictability of bidirectional graph transformations. Tech. Rep. GRACE-TR-2015-03, GRACE Center, National Institute of Informatics (Sep 2014), `http://www.prg.nii.ac.jp/projects/gtcontrib/cmpbx/cmpbx.pdf`
14. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: ICFP'10. pp. 205–216. ACM (2010)
15. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K., Sasano, I.: Marker-directed optimization of UnCAL graph transformations. In: LOPSTR'11, Revised Selected Papers. LNCS, vol. 7225, pp. 123–138 (Jul 2012)
16. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations (short paper). In: ASE'11. pp. 480–483. IEEE (2011)
17. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. Progress in Informatics (10), 131–148 (March 2013), `http://dx.doi.org/10.2201/NiiPi.2013.10.7`, journal version of [16]
18. Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Towards compostional approach to model transformations for software development. Tech. Rep. GRACE-TR08-01, GRACE Center, National Institute of Informatics (Aug 2008)

19. Hidaka, S., Hu, Z., Kato, H., Nakano, K.: A compositional approach to bidirectional model transformation. In: ICSE New Ideas and Emerging Results track, ICSE Companion. pp. 235–238. IEEE (2009)
20. Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Towards a compositional approach to model transformation for software development. In: Proc. of the 2009 ACM symposium on Applied Computing (SAC). pp. 468–475. ACM (2009)
21. Hidaka, S., Terwilliger, J.F.: Preface to the third international workshop on bidirectional transformations. In: Workshops of the EDBT/ICDT 2014. pp. 61–62. `http://ceur-ws.org/Vol-1133#paper-09`
22. Matsuda, K., Wang, M.: "bidirectionalization for free" for monomorphic transformations. Science of Computer Programming (2014), DOI:10.1016/j.scico.2014.07.008
23. Pombrio, J., Krishnamurthi, S.: Resugaring: Lifting evaluation sequences through syntactic sugar. In: PLDI'14. pp. 361–371. ACM (2014)
24. Schürr, A.: Specification of graph translators with triple graph grammars. In: WG '94. LNCS, vol. 903, pp. 151–163 (Jun 1995)
25. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and System Modeling 9(1), 7–20 (2010)
26. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: ECMDA-FA'09. LNCS, vol. 5562, pp. 18–33 (2009)
27. Voigtländer, J.: Bidirectionalization for free! (pearl). In: POPL '09. pp. 165–176. ACM (2009)